Durable Execution for AI Agents: A Design Pattern for Fault-Tolerant Agent Loops

Stenio de Lima Ferreira
Temporal Technologies Inc.
New York, NY
stenio123@gmail.com

***Abstract*—AI agents—autonomous systems that iteratively invoke tools and make decisions based on accumulated context—are increasingly deployed in mission-critical cloud applications. Yet when these agents fail due to infrastructure crashes, network issues, or service timeouts, they cannot reliably resume, leading to incomplete transactions, duplicated operations, or workflow abandonment.**

We present the Durable AI Agent Loop Architecture, a design pattern addressing this reliability gap through three primitives: deterministic control flow, managed side-effect abstraction, and event-sourced state persistence. Rather than relying on volatile checkpoints, this architecture records every operation as an immutable event before proceeding, ensuring state reconstruction after failures. By isolating non-deterministic operations (LLM calls and tool invocations) from orchestration logic, agents recover without redundant execution or logic drift.

We validate this approach by testing resilience against reproducible failure scenarios—including worker process crashes, external service timeouts, and logic errors—on a reference implementation. Our analysis demonstrates that the architecture maintains state integrity across these failures, and performance measurements show that the durability overhead remains negligible relative to typical LLM inference times. We provide a reproducible open-source reference implementation to enable independent verification and practical adoption.

***Index Terms*—AI agents, fault tolerance, durable execution, event sourcing, distributed systems, cloud computing, LLM**

I. Introduction

Large Language Model (LLM)-based AI agents are increasingly deployed in mission-critical cloud applications, where they employ complex, iterative reasoning patterns—often termed "Agent Loops"—repeatedly invoking external tools, processing results, and making decisions based on accumulated context [1]. These stateful workflows now operate in domains such as autonomous customer service, intelligent process automation, and healthcare, where reliability and consistency are critical.

However, the current landscape of AI agent frameworks presents a critical architectural gap: *the absence of robust fault-tolerance guarantees for stateful agent execution.* When an agent workflow fails mid-execution due to infrastructure faults, external service timeouts, or transient errors, existing frameworks often face irrecoverable state loss. The agent cannot reliably resume from its last known-good state, leading to incomplete transactions or duplicated operations. As noted by Vogels, "Everything fails, all the time" [2], meaning failures are inevitable in distributed systems. This can be illustrated by recent large-scale outages across major cloud providers [3-5].

Existing agentic AI frameworks have emerged to simplify agent development, yet their production readiness for stateful execution remains unclear. Representative frameworks such as LangGraph [6], CrewAI [7], and AutoGen [8] enable rapid prototyping through graph-based abstractions. However, as these agents scale to mission-critical deployments, a fundamental question emerges: *how to guarantee durable execution without state loss or duplicate operations* when infrastructure failures are inevitable.

Two approaches have emerged for managing stateful agent execution:

- **Checkpoint-based**: Persists snapshots of the complete runtime state at specific intervals. This creates a *consistency gap* if a failure occurs after an external tool executes but before the state is saved.
- **Event-sourced**: Records every operation as an immutable event *before* proceeding. This ensures operation-level atomicity but necessitates *deterministic control flow* and *side-effect abstraction* to enable reliable reconstruction during replay.

Here, determinism refers to the reproducibility of the agent's orchestration logic (the sequence of steps the agent executes); non-deterministic operations such as LLM inference, are encapsulated as side-effect abstractions to ensure replayability during recovery.

Fig. 1 illustrates these contrasting recovery models.

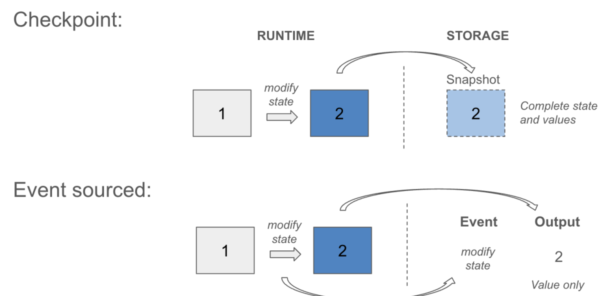

Fig. 1. Comparison of State Recovery Models. Checkpoint-based architectures (top) persist complete snapshots of runtime state. Event-sourced architectures (bottom) persist only the operation and its output, enabling state reconstruction through replay.

We formalize these requirements into the *Durable AI Agent Loop Architecture*, a design pattern defined by three primitives: **Deterministic Control Flow**, **Managed Side-Effect Abstraction**, and **Event-Sourced State Persistence**.Together, they ensure an agent's internal state remains synchronized with its external side effects.

These primitives are encountered in modern workflow orchestration platforms like Azure Durable Functions [9], AWS Step Functions [10], and Temporal [11]. While implementations differ, each provides the necessary building blocks to realize the Durable AI Agent Loop Architecture.

The underlying execution model, formalized as *durable execution* [12], guarantees workflow completion despite infrastructure failures—achieving *crash-proof execution* through deterministic replay against a persisted event history. This goes beyond conventional retry mechanisms that handle transient API failures; durable execution enables workflows to survive process crashes and run for arbitrary durations.

A. Contributions

This work makes three contributions:

- **Architectural Pattern**: We formalize the Durable AI Agent Loop Architecture, a design pattern that applies event sourcing and deterministic replay to AI agent execution.
- **Fault Injection Evaluation**: We test the architecture against three failure scenarios—process crashes, service unavailability, and code bugs—demonstrating recovery without state loss.
- **Reference Implementation**: We provide an open-source implementation using an open source framework, enabling practitioners to reproduce our experiments and adapt the pattern.

II. Related Work

A. Agent Orchestration Frameworks

Contemporary agent frameworks such as LangGraph [6], CrewAI [7], and AutoGen [8] provide abstractions for multi-step reasoning and tool coordination. While these frameworks enable rapid prototyping, they typically rely on in-memory state or periodic checkpointing for persistence. This approach creates consistency gaps where an agent's internal state can diverge from its external actions during failures, as discussed in the Introduction section.

A systematic experimental comparison of fault tolerance across agent frameworks remains beyond the scope of this work. Our contribution focuses on formalizing and validating the architectural pattern itself; comparative benchmarking against LangGraph, CrewAI, and AutoGen under identical fault conditions represents important future work.

B. Durable Execution Platforms

Our work builds on platforms that support durable execution patterns, including Azure Durable Functions [9], AWS Step Functions [10], and Temporal [11]. These platforms provide primitives for event sourcing [13] and deterministic replay to persist state and recover from failures—concepts rooted in decades of distributed systems research [14, 15].

Such platforms also support reliability patterns like the Saga Pattern [16, 17] to coordinate rollback through compensating actions when distributed transactions fail. However, Sagas address *logical* failures (reverting completed operations), not *infrastructure* failures (recovering lost state). Our work focuses on the orthogonal challenge of maintaining execution continuity when workers crash or networks or external services fail, ensuring the agent's reasoning history remains consistent with its external actions.

Traditional workflow orchestration systems like Apache Airflow [18] and Prefect [19] also persist execution state but use checkpoint-based approaches and focus primarily on batch data pipelines rather than interactive agent loops.

We extend the durable execution paradigm to agentic systems, where control flow is dynamically determined by LLM responses. This introduces unique challenges: high-latency non-deterministic operations (LLM calls), interactive state management (user signals), and the need to preserve execution history for restoring state after failures. While durable execution platforms provide the infrastructure primitives, our contribution is formalizing how to apply them specifically to autonomous AI agent architectures.

C. Agent Reliability and Testing

Research on AI agent reliability is still emerging. AgentBench [20] and ToolBench [21] evaluate agent task completion but do not assess fault tolerance and state recovery. LangSmith [22] provides observability for LangChain applications but relies on external logging rather than event-sourced execution history capable of resuming failed processes. Our work complements these efforts by focusing specifically on execution durability and system-level state consistency under failure conditions.

III. The Durable AI Agent Loop Architecture

The Durable AI Agent Loop Architecture ensures reliable execution by defining the interaction between agent logic and fault-tolerant orchestration. Rather than relying on manual or automated checkpointing, this architecture treats the agent's reasoning process as a series of durable state transitions that can be reliably reconstructed after failures.

A. Pattern Overview

The architecture separates an AI agent into three layers: (1) a deterministic orchestrator controlling execution flow, (2) managed side-effect abstractions encapsulating non-deterministic operations, and (3) an event-sourced persistence layer. This enables recovery through deterministic replay: when failure occurs, the orchestrator re-executes from the start, reading from the event log. If an agent completes steps 1-5 but fails on step 6, replay returns recorded results for steps 1-5 and only re-executes step 6.

B. Core Primitives

1) Deterministic Control Flow (The Orchestrator): The orchestrator implements the agent's iterative reasoning loop (e.g., a while loop managing the ReAct [23] pattern). To support reliable replay, its code must be strictly deterministic[1]—it cannot directly perform I/O or invoke external systems. All non-deterministic operations, including external calls, must be delegated to activities. Non-deterministic values like timestamps or random numbers must be obtained from the persistence layer, which provides stable, recorded values that remain consistent during replay. This ensures the agent's execution path can be identically reconstructed across restarts or different compute nodes.

2) Managed Side-Effect Abstraction (Activities): Activities encapsulate all non-deterministic interactions, including LLM inference and tool invocations. Each activity represents an atomic unit of work that can be independently executed, retried on failure, and persisted on completion.

Replay Behavior: Once an activity completes successfully, its result is persisted to the event log. During replay, the orchestrator retrieves stored results instead of re-executing. If an activity fails mid-execution, retries re-run it from the start; thus, activities modifying external state must be idempotent [24].

Failure Handling: If an activity exhausts its retry attempts, the workflow transitions to a failed state while preserving its full execution history. This enables operators to fix logic issues and resume the agent without losing progress.

3) Event-Sourced State Persistence: This layer records every operation as an immutable log entry [13], providing operation-level atomicity: an LLM response is only considered "committed" once it has been successfully logged. This eliminates the dual-write problem where an operation's effects could persist while the state record is lost, ensuring the agent's internal state never diverges from its external actions.

Fig. 2 shows how these components interact.

C. Primitives Across Platforms

The Durable AI Agent Loop is platform-agnostic. Table I shows how these primitives map to existing orchestration platforms: Temporal, Azure Durable Functions, and AWS Step Functions.

TABLE I
PLATFORM INSTANTIATIONS OF DURABLE PRIMITIVES

Primitive	Temporal	Azure	AWS
Deterministic Control Flow	Workflow	Orchestrator	State Machine
Managed Side-Effect Abstraction	Activities	Activity Functions	Lambda Tasks
Event-Sourced State Persistence	Event Sourcing	Checkpoint & Replay	Execution History

[1]As defined in the Introduction, this refers to structural workflow determinism—the requirement that the orchestration logic follows an identical execution path during replay—rather than LLM output determinism.

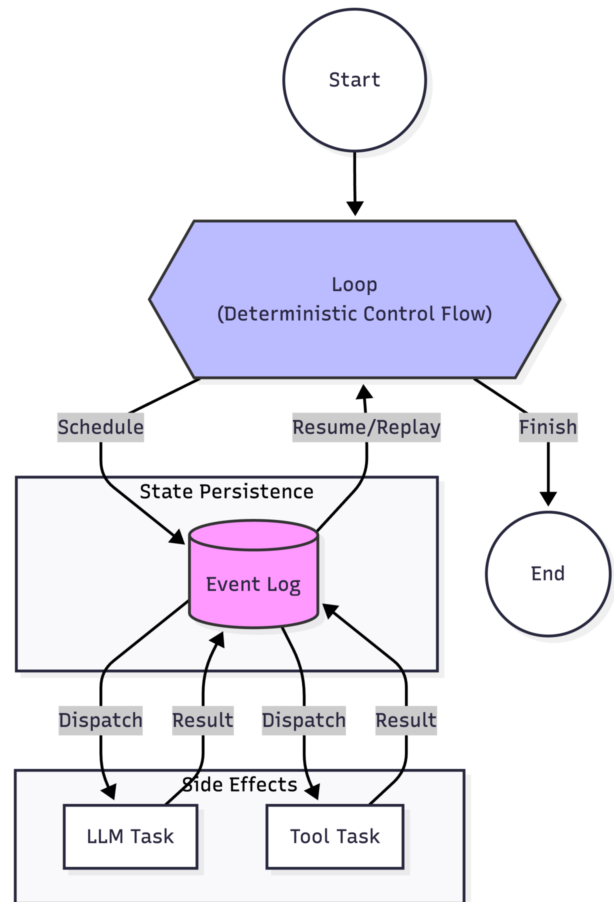

Fig. 2. The Durable AI Agent Loop State Transition Model. The **Deterministic Control Flow** (Loop) transitions state by scheduling **Managed Side-Effects Abstraction** (Side Effects). Execution proceeds only after results are committed to the **Event-Sourced State Persistence** layer (State Persistence). This log-first approach ensures the agent's execution history is always reconstructible from the immutable event log.

IV. Reference Implementation

We implemented the Durable AI Agent Loop Architecture using Temporal with Python. Temporal was selected because it is open-source and self-hostable, enabling full reproducibility without cloud accounts or runtime costs. However, the architectural pattern can be implemented on any platform providing the necessary primitives, including Azure Durable Functions or AWS Step Functions.

A. Temporal Framework Overview

Temporal consists of three components: a *Server* that manages workflow state and event persistence, *Workers* that execute workflow and activity code, and *Clients* that start and interact with running workflows. Each of these components can run on the same or different machines.

Temporal also utilizes two abstractions: *Workflows* and *Activities*. Workflows define the deterministic orchestration logic—in the Durable AI Agent Loop Architecture, this is the iterative reasoning loop. Activities contain non-deterministic logic such as LLM calls and tool execution, allowing them

to be retried independently. Workers execute both Workflow and Activity code, while clients communicate with the server to start workflows and send signals. The server maintains an immutable event log of all workflow actions, enabling workers to reconstruct workflow state by replaying the workflow code against this log when failures occur.

B. Interaction Flow

In our implementation, the user input is passed to the workflow implementing the deterministic loop. The agent's reasoning (via LLM) and tool execution are delegated to non-deterministic activities. LLM calls are abstracted using LiteLLM, allowing the implementation to work with different providers (OpenAI, Anthropic, etc.) via configuration. This serves as the foundation for our experimental evaluation.

Fig. 3 illustrates the sequence of interactions during a single user message.

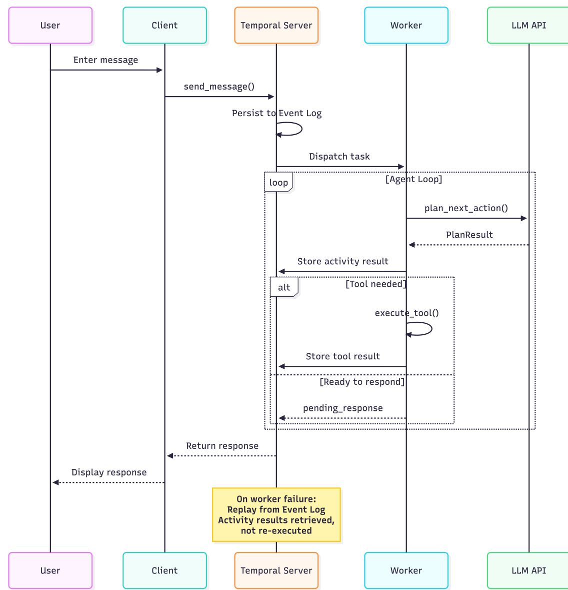

Fig. 3. Sequence Diagram for a Reference Durable AI Agent Loop Implementation. The client sends a message to the Temporal server, which persists it and dispatches to a worker. The worker executes the agent loop—calling the LLM to plan, optionally executing tools, and returning a response. All activity results are stored in the event log, enabling recovery without re-execution.

C. Implementation Structure

The reference implementation consists of several files. Four are central to the architecture:

- `workflow.py`: The deterministic orchestrator (`DurableAgentWorkflow` class) implementing the agent loop. It waits for user input, delegates reasoning to activities, and coordinates tool execution based on LLM decisions.
- `activities.py`: Managed side-effects (`AgentActivities` class) encapsulating LLM inference (`plan_next_action`) and tool execution (`execute_tool`). These are the only components that perform I/O.
- `worker.py`: The Temporal worker process that executes workflow and activity code.
- `start_workflow.py`: The client that initiates workflows and sends user messages.

LLM calls are abstracted using LiteLLM, allowing the implementation to work with different providers (OpenAI, Anthropic, etc.) via configuration. Complete source code is available in the provided repository [25].

D. Recovery and Consistency

The implementation ensures atomicity between tool execution and state updates. If a tool completes but the workflow crashes before processing the result, the Temporal worker re-executes the workflow code from the beginning upon reconnection, reading from the event log. Completed activities return their persisted results immediately rather than re-executing, restoring the workflow to its pre-crash state. This prevents duplicate operations and ensures consistent agent behavior across failures.

Next, we evaluate the resilience of this implementation through controlled fault injection experiments.

V. Experimental Evaluation

A. Test Environment

Experiments ran on an Apple M4 Pro (24GB RAM) with macOS 15.7.3, Python 3.12, and Temporal Server v1.29.1. Performance metrics were averaged over 50 iterations. The evaluation consisted of two components: fault injection tests (`MANUAL_TESTS.md`) to verify correctness under failure conditions, and performance benchmarks (`performance_benchmark.py`) to measure operational overhead introduced by the durability layer.

B. Fault Injection Tests

We subjected the implementation to synthetic faults targeting each core primitive. Our fault model covers three operational scenarios: process crashes (testing infrastructure resilience), external service timeouts (testing retry mechanisms), and logic errors requiring hot fixes (testing state persistence across code deployments). These represent practical failure modes encountered when deploying agent systems in production environments.

1) Scenario 1 - Process Crash:

a) Primitive Tested: Deterministic Control Flow

We simulated a worker crash by sending a SIGKILL signal immediately after a successful LLM inference but before the next operation.

b) Outcome: When the worker restarted, the workflow reconstructed its state from the event log. Because the LLM activity was already recorded, the workflow code replayed up to the failure point and immediately scheduled the next tool activity. The system recovered without re-executing the LLM call, preserving both cost and execution intent.

2) Scenario 2 - External Service Timeouts:

a) Primitive Tested: Managed Side-Effect Abstraction We tested resilience to external service failures by modifying the activity code to raise a ServiceUnavailable error for the first three attempts before succeeding.

b) Outcome: The platform automatically detected the failure and applied exponential backoff between retries. The workflow remained in a running state but did not consume compute resources while waiting. Once the service became available, the agent completed its task without requiring manual error-handling code in the workflow logic.

3) Scenario 3 - Logic Fault and Hot-Fix:

a) Primitive Tested: Event-Sourced State Persistence We introduced a runtime exception in the tool execution logic after the LLM planning activity had already been committed to the event log, causing the workflow to fail repeatedly.

b) Outcome: The workflow entered a retrying state. After deploying a code fix and restarting the worker, the workflow replayed from the event log, retrieved the existing LLM planning result, and successfully executed the corrected logic. This demonstrated that code-level bugs can be fixed without restarting the workflow from the beginning, as long as side-effect results are durably persisted.

C. Performance Benchmarks

To quantify the operational overhead of the Durable AI Agent Loop, we measured end-to-end latency in two configurations: with real LLM calls and with a mock LLM client that returns instantly, both executing without failures. The difference isolates the durability cost from LLM inference time. Measurements represent average latency over $n = 50$ iterations. Table II summarizes the results.

TABLE II
LATENCY MEASUREMENTS

Configuration	Mean	Std Dev	Min	Max
Real LLM	705 ms	116 ms	585 ms	931 ms
Mock LLM	81 ms	38 ms	5 ms	104 ms
Temporal Overhead	**81 ms**	-	-	-

Measurements from reference implementation in local environment. Production metrics will vary based on infrastructure.

The mock LLM configuration isolates the framework overhead—including event persistence, workflow orchestration, state management, and local network round-trips. In our test environment, initial interactions completed in approximately 5 ms, while subsequent interactions averaged 87–104 ms (mean: 81 ms). This variance likely reflects default Temporal server configurations rather than inherent architectural constraints; production deployments typically undergo tuning that can substantially reduce such overhead [26]. Framework overhead represents roughly 11% of the total end-to-end latency when using a real LLM, demonstrating that durability guarantees come at a modest cost relative to typical inference times. These values will vary based on infrastructure configuration, network topology, and server deployment.

D. Analysis

The experimental results reveal both the effectiveness of the architectural primitives and their operational characteristics.

1) Resilience Guarantees: Scenarios 1 and 3 demonstrate how the use of a Deterministic Control Flow enables recovery through state reconstruction. When the worker restarted after a SIGKILL, the workflow resumed within seconds, without requiring re-execution of the LLM call or losing previous execution results.

Scenario 2 validates that the Managed Side-Effect Abstraction handles transient external failures transparently—the workflow logic remained unchanged while the platform managed retries with exponential backoff.

This eliminates side-effect divergence—inconsistent states where an external tool executes successfully but the result is lost due to a crash before state is captured. Without event sourcing [27], such ”ghost executions” create irrecoverable inconsistencies.

2) Observability Benefits: A key finding is the diagnostic value of the immutable event history. The Event-Sourced Persistence layer provides a complete execution log visible in the Temporal UI, enabling deterministic replay of failed agent trajectories. In our experiments, this allowed us to identify the precise sequence of events leading to the logic fault in Scenario 3, significantly reducing debugging time compared to traditional logging approaches.

E. Limitations and Trade-offs

While the architecture provides strong durability guarantees, several trade-offs must be considered:

- Operational Overhead: Requires deployment and maintenance of durable execution infrastructure (Temporal cluster, database for event storage).
- Architectural Rigor: Developers must maintain strict determinism in workflow code. Standard operations like remote calls, reading system clocks, and generating random numbers must be delegated to activities.
- Latency Overhead: In our local environment, the durability layer added approximately 80 ms per interaction. While modest compared to LLM latency, this may accumulate in workflows with extensive state transitions.

VI. Conclusion

This paper formalizes the Durable AI Agent Loop Architecture, an architectural pattern that provides fault-tolerant execution for autonomous AI agents. By combining deterministic control flow, managed side-effect abstraction, and event-sourced state persistence, the architecture addresses the state loss and side-effect divergence problems encountered in checkpoint-based agent frameworks.

Our experimental evaluation demonstrates that the architecture maintains state integrity in the event of infrastructure crashes, service timeouts, and logic errors. The shift from checkpoint-based snapshots to log-based execution ensures that an agent's internal state remains consistent with its external actions. Additionally, the complete execution log

enables deterministic replay, providing significant advantages for debugging and analyzing non-deterministic agent behavior.

Performance measurements show that the durability overhead remains small relative to typical LLM inference times—in our test environment, approximately 11% of total end-to-end latency—making the trade-off favorable for production agent deployments.

A. Future Work

Although this work establishes a foundation for durable single-agent execution, several research directions remain open:

- Dynamic Replay and Execution Branching: The immutable event log creates opportunities for forking agent execution at specific points in history to explore alternative reasoning paths. This would enable comparative analysis of different agent strategies without re-executing the entire conversation history.
- Long-Running Agent State Management: For agents intended to run continuously over extended periods, indefinite log growth presents a performance concern. Future work should investigate state compaction strategies that preserve durability guarantees while bounding replay costs.
- Comparative Framework Analysis: This work does not experimentally compare the Durable AI Agent Loop against existing agent frameworks under failure conditions. A systematic evaluation measuring state recovery, execution consistency, and operational overhead across LangGraph, CrewAI, AutoGen, and durable execution implementations would provide practitioners with quantitative guidance for framework selection.

By providing both a formal pattern definition and an open-source reference implementation, this work aims to help transition AI agents from experimental prototypes to production-ready systems with enterprise-grade reliability guarantees.

References

[1] Q. Wu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," 2023, *arXiv:2308.08155*. [Online]. Available: https://arxiv.org/abs/2308.08155

[2] W. Vogels, "Everything fails all the time," *Commun. ACM*, vol. 59, no. 3, pp. 44–46, Mar. 2016. [Online]. Available: https://cacm.acm.org/opinion/everything-fails-all-the-time/

[3] Cloudflare, "Understanding how Facebook disappeared from the Internet," Oct. 2021. [Online]. Available: https://blog.cloudflare.com/october-2021-facebook-outage/. Accessed: Jan. 2026.

[4] ThousandEyes, "Microsoft Outage Analysis: January 25, 2023," Jan. 2023. [Online]. Available: https://www.thousandeyes.com/blog/microsoft-outage-analysis-january-25-2023. Accessed: Jan. 2026.

[5] AWS, "AWS Post-Event Summaries," n.d. [Online]. Available: https://aws.amazon.com/premiumsupport/technology/pes/. Accessed: Jan. 2026.

[6] LangChain, "LangGraph," n.d. [Online]. Available: https://www.langchain.com/langgraph. Accessed: Jan. 2026.

[7] CrewAI, "Multi-AI agent systems," n.d. [Online]. Available: https://www.crewai.com. Accessed: Jan. 2026.

[8] Microsoft, "AutoGen documentation," n.d. [Online]. Available: https://microsoft.github.io/autogen/stable/index.html. Accessed: Jan. 2026.

[9] Microsoft Azure, "Durable Functions documentation," n.d. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/durable/. Accessed: Jan. 2026.

[10] Amazon Web Services, "AWS Step Functions documentation," n.d. [Online]. Available: https://docs.aws.amazon.com/step-functions/. Accessed: Jan. 2026.

[11] Temporal Technologies, "Temporal documentation," n.d. [Online]. Available: https://temporal.io. Accessed: Jan. 2026.

[12] S. Burckhardt *et al.*, "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021, Art. no. 133, doi: 10.1145/3485510.

[13] M. Fowler, "Event sourcing," Dec. 2005. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html. Accessed: Jan. 2026.

[14] J. Gray, "Why do computers stop and what can be done about it?," in *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, Erlangen, 1985. [Online]. Available: https://jimgray.azurewebsites.net/papers/TandemTR85.7_WhyDoComputersStop.pdf. Accessed: Jan. 2026.

[15] D. Yuan *et al.*, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proc. 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 249–265.

[16] H. Garcia-Molina and K. Salem, "Sagas," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1987, pp. 249–259.

[17] C. Richardson, "Pattern: Saga," in *Microservices Patterns*, Shelter Island, NY: Manning Publications, 2018, ch. 4.

[18] Apache Airflow, "Apache Airflow documentation," n.d. [Online]. Available: https://airflow.apache.org/. Accessed: Jan. 2026.

[19] Prefect, "Prefect: The path of least resistance," n.d. [Online]. Available: https://prefect.io/. Accessed: Jan. 2026.

[20] X. Liu *et al.*, "AgentBench: Evaluating LLMs as agents," 2023, *arXiv:2308.03688*.

[21] Q. Zuo *et al.*, "On the tool manipulation capability of open-source large language models," 2023, *arXiv:2305.16504*.

[22] LangSmith, "LangSmith observability," n.d. [Online]. Available: https://www.langchain.com/langsmith/observability. Accessed: Jan. 2026.

[23] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *Proc. 11th Int. Conf. on Learning Representations (ICLR)*, May 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[24] P. Helland, "Idempotence is not a medical condition," *Commun. ACM*, vol. 55, no. 5, pp. 56–65, May 2012.

[25] S. de Lima Ferreira, "Durable AI Agent Loop reference implementation," 2024. [Online]. Available: https://github.com/stenio123/durable-agent-loop-reference-implementation. Accessed: Jan. 2026.

[26] Temporal Technologies, "Worker performance and scalability," n.d. [Online]. Available: https://docs.temporal.io/develop/worker-performance. Accessed: Jan. 2026.

[27] M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.