

Architecture of a Durable Agent

Runtime: Reliable Loops, Fault Tolerance, and Recovery

1. Foundations of Reliable Agent Execution

The transition of artificial intelligence agents from experimental scripts to production systems reveals a critical vulnerability: the fragility of in-memory reasoning loops. While large language models (LLMs) provide advanced planning and synthesis, treating an agent invocation as a synchronous, indivisible network call guarantees failure in distributed environments.¹ To survive network partitions, server crashes, and provider outages, agent execution must be architected as a durable, observable, stateful workflow system.²

Definitional Taxonomy

To establish a precise architectural vocabulary, the following operational definitions are required:

Term	Definition
Agent	An autonomous software entity combining a language model, tools, and a control loop to achieve an objective.
Agent invocation	A single request initiating an agent's processing cycle, potentially spanning multiple asynchronous steps.
Agent loop	The iterative, cyclic process of observing state, deciding on an action, executing it, and evaluating results.
Workflow	A durable, stateful sequence of tasks orchestrating one or more agents toward a comprehensive business goal.
Task	A distinct unit of assignable work within a

	broader workflow.
Step	A discrete, checkpointable operation within a task, such as a reasoning phase or external tool invocation.
Attempt	A specific execution instance of a step, allowing for retries upon failure.
Action	An intent generated by an agent to mutate an external environment or request new information.
Observation	The structured result of an action, integrated back into the agent's context window.
Decision	The reasoning phase where the agent determines its next action based on current state.
State transition	The atomic movement of a workflow, task, or step from one defined status to another.
Checkpoint	A durable representation of resumable progress saved to persistent storage. ¹
Snapshot	A point-in-time copy of the entire workflow state, enabling time-travel debugging. ³
Event	An immutable record of a state change or action within the runtime.
Side effect	A mutation made to an external system (e.g., sending an email, charging a card).
Retry	The re-execution of a failed operation, typically mediated by an exponential backoff policy.

Replay	Re-running historical events through deterministic workflow code to rebuild state in-memory. ⁴
Resume	Continuing a paused or crashed workflow exactly from its latest valid checkpoint. ¹
Restart	Beginning a task entirely from scratch, permanently discarding previous progress.
Recovery	The systemic process of restoring an interrupted runtime to a healthy, processing state.
Rollback	Reverting local database changes within a single atomic transactional boundary.
Compensation	Executing a new, semantically opposite action to reverse a previously completed external side effect. ⁵
Cancellation	The intentional, cooperative termination of a workflow before its natural completion.
Timeout	A defined threshold after which an operation is forcefully considered degraded or failed.
Failure	The systemic inability of a component to perform its required function.
Error	A specific, often catchable programmatic issue that may precipitate a failure.
Exception	A runtime anomaly interrupting the standard execution flow.
Dead letter	A task that cannot be processed successfully and is isolated in a queue for human review.

Heartbeat	A periodic signal emitted by a worker proving it is alive and actively processing a lease.
Lease	A time-bound, exclusive right granted to a worker to process a specific task. ⁶
Watchdog	An independent monitoring process that observes agent loops for stagnation or policy breaches.
Circuit breaker	A state machine that halts requests to a repeatedly failing downstream service to prevent cascading exhaustion. ⁷
Idempotency	The property of an operation guaranteeing that multiple executions yield the exact same result as a single execution. ⁸

Treating an agent invocation as an indivisible model call traps side effects, contextual memory, and human approvals within an ephemeral process.⁴ If the process dies, the context vanishes. Therefore, execution must be decomposed into strict layers, each possessing distinct failure domains and recovery mechanisms.

Execution Layers and Failure Domains

1. **Workflow Layer:** Manages the directed acyclic graph (DAG) of the broader objective. Failures here indicate structural logic errors. State must be durably event-sourced.⁴ Recovery involves workflow version migration or human intervention.
2. **Task Layer:** Tracks specific assignments. It stores inputs, constraints, and completion status. Failures often stem from budget exhaustion or timeout. Recovery requires resuming from task checkpoints.
3. **Agent-Loop Layer:** Manages iterative cognition. It stores reasoning outputs and intermediate plans. Blind retries here create infinite reasoning loops; recovery mandates either context reloading or strategy replanning.
4. **Model-Invocation Layer:** Interfaces directly with LLM providers. It stores prompt hashes and parameter configurations. It fails via rate limits or 503 errors.² Recovery is achieved via transient retries with exponential backoff.
5. **Tool-Execution Layer:** Handles local API interactions. It stores tool inputs and deterministic outputs. Recovery depends on the tool's specific error payload.
6. **External Side-Effect Layer:** Interacts with the real world. It stores idempotency keys and receipts.⁴ Blind retries duplicate real-world actions. Recovery requires idempotent wrappers or compensating sagas.⁸

7. **Persistence Layer:** The database storing checkpoints. Failures here stall the entire runtime. Recovery involves database failover mechanisms.
8. **Observability Layer:** Emits traces. It fails silently. Recovery requires resilient telemetry sidecars.

Recovering a model call is a simple network operation. Recovering an agent step involves reloading a prompt context. Recovering a task involves identifying a lease expiration and resuming from a checkpoint.⁶ Recovering a workflow requires orchestrating distributed sagas across multiple tasks.⁵ Recovering an external business action requires executing a logical compensation (e.g., a refund).⁹

2. Observe–Think–Act Loops

The foundational architecture of an agent is the Observe–Think–Act loop. This pattern dictates how an agent interprets its environment, reasons about its objectives, and selects tools to manipulate external state.

Core Components

- **Observation:** The ingestion of environment state, tool receipts, and external events.
- **Internal state:** The derived memory representing the agent's current understanding of the task.
- **Reasoning (Decision) phase:** The LLM invocation analyzing the observation against the objective.
- **Action selection:** The formulation of a structured intent (e.g., a JSON tool call).
- **Tool execution:** The physical execution of the selected action.
- **Environment update:** The mutation resulting from the tool.
- **Result evaluation:** Assessing whether the action satisfied the immediate requirement.

Canonical Loop Architecture

Code snippet

flowchart TD

```
A --> B
B --> C
C --> D{Validate Proposed Action}
D -- Invalid --> E
E --> C
D -- Valid --> F[Execute Action]
F --> G
G --> H
H --> I{Evaluate Completion}
I -- Incomplete --> A
```

I -- Complete --> J
I -- Blocked --> K[Escalate / Pause]

Variants of this loop include ReAct (interleaving reasoning traces with explicit tool requests), tool-use loops (relying on native LLM function calling), event-driven loops (dormant until triggered by webhooks), and hierarchical supervisor-worker networks.³

However, these loops exhibit unique failure modes. **Repeated observations** occur when an agent repeatedly polls an unchanged environment. **Reasoning detached from evidence** manifests as hallucination, where the agent proposes actions based on unverified premises.

Acting before validating results in malformed API calls. **Unbounded iteration** occurs when an agent fails to recognize completion and loops infinitely.¹⁰

To achieve durability, every meaningful transition must be persisted. *Before* an action, the runtime must record the exact proposed tool input and intent to an action ledger. *After* the action, the raw observation and tool receipt must be checkpointed before the agent's context window is updated.²

3. Plan–Execute–Verify Loops

For deterministic, multi-step workflows, the reactive Observe–Think–Act loop is insufficient. The Plan–Execute–Verify architecture imposes structured rigor, separating the formulation of strategy from physical execution and subsequent quality control.

The **planning phase** decomposes a high-level objective into a dependency graph of sub-tasks.

The **execution phase** sequentially or concurrently processes these steps. The **verification phase** applies deterministic or model-based heuristics to evaluate the output. If verification fails, a **replanning phase** is triggered to adjust the strategy based on the new context.

Architectures vary significantly. *Plan once, then execute* is brittle; if step two fails, the remainder of the plan is useless. *Iterative planning* adapts dynamically but consumes massive token budgets. *Planner-executor-critic* trios distribute cognitive load among specialized models, preventing a single model from falling into a localized reasoning rut.

Plans inevitably become stale. The environment may shift, a dependency API may go offline, or a human may alter requirements mid-flight. A durable runtime requires strict plan invalidation rules. Plans must be versioned in the state database. If an assumption proves false, the specific step and all dependent downstream steps must be marked Invalidated.

Crucially, verification must not rely on asking the exact same model instance if its own work is correct; LLMs exhibit high sycophancy and will confidently validate their own hallucinations.

Robust verification mechanisms include deterministic schema validation, static code analysis, invariant data checks, unit tests, and completely independent model review (e.g., using a separate reasoning model with a zero-temperature strict grading prompt).

4. State Machines for Agent Behaviour

Inferring the status of a workflow by passing the entire conversation history to an LLM and asking "Is the task done?" is a catastrophic anti-pattern. Explicit state management via a finite-state machine (FSM) is significantly more reliable and computationally efficient.²

A standard agent-task state machine establishes rigid guardrails around execution.

Task State Definition Table

State	Meaning	Entry Transitions	Exit Transitions	Timeout / Retry Policy
Created	Task defined, awaiting validation.	(Initial)	Validating, Cancelled	Absolute deadline applies.
Ready	Validated, waiting for capacity.	Created, Paused	Queued, Cancelled	No timeout; moves to DLQ if un-queued.
Queued	In broker, awaiting worker.	Ready, Retrying	Claimed, Cancelled	Queue visibility timeout applies.
Claimed	Worker acquired exclusive lease.	Queued	Initialising, Failed	Lease expiration triggers re-queue. ⁶
Executing	Agent loop is actively running.	Initialising	Waiting for Tool, Completed	Step timeout limits LLM generation.
Waiting for Tool	Blocked on external execution.	Executing	Executing, Timed Out	Tool SLA timeout applies.
Waiting for Human	Blocked on user input.	Executing	Executing, Escalated	Approval timeout triggers escalation.
Paused	Operator halted execution.	Executing	Ready, Cancelled	Indefinite; preserves lease state.

Checkpointed	Safe boundary saved to disk.	Executing	Executing, Resumed	Represents a durable rest state.
Completed	Acceptance criteria strictly met.	Executing, Verifying	Archived	N/A
Failed	Terminal error encountered.	Executing, Claimed	Retrying, Dead-lettered	N/A
Dead-lettered	Unrecoverable without human fix.	Failed	Compensating, Archived	Indefinite retention for forensics.
Compensating	Reversing external side effects.	Dead-lettered	Rolled back, Failed	Strict compensation deadline applies.

It is vital to differentiate between states. *Task state* tracks the holistic objective. *Step state* tracks the current FSM node. *Attempt state* tracks retries. *Lease state* tracks worker ownership. Invalid transitions must be strictly blocked by the database. Transitioning from Completed directly to Executing is invalid. Transitioning from Waiting for Human while maintaining an active exclusive lease is an anti-pattern that exhausts worker capacity; the lease must be released and reacquired upon human response.

Code snippet

stateDiagram-v2

```

[*] --> Created
Created --> Ready
Ready --> Queued
Queued --> Claimed
Claimed --> Executing
Executing --> WaitingForTool
WaitingForTool --> Executing
Executing --> WaitingForHuman
WaitingForHuman --> Checkpointed
Checkpointed --> Queued : Human Replied

```

Executing --> Failed
Failed --> Retrying
Retrying --> Queued
Failed --> DeadLettered
DeadLettered --> Compensating
Compensating --> RolledBack
Executing --> Verifying
Verifying --> Executing : Replanning
Verifying --> Completed
Completed --> [*]
RolledBack --> [*]

5. Durable Execution and Event Sourcing

Durable execution is the architectural guarantee that a long-running workflow will survive process crashes, scale-in events, and infrastructure failures without losing progress.¹ When a basic agent crashes mid-task, its in-memory context disappears. Durable execution externalizes this state.

Persistence approaches vary. *Mutable workflow rows* update a single database record, which is fast but destroys historical context. *Periodic snapshots* save the full state at intervals. *Event sourcing* appends immutable state changes to a log, reconstructing the current state via deterministic replay.⁴ Agent runtimes function best using a hybrid: step-boundary snapshots (checkpoints) to avoid massive replay overhead, combined with append-only write-ahead logs for intra-step durability.³

In a durable system, the database is the authoritative state; the agent's memory is merely derived state.¹¹ If a crash occurs, the worker fetches the latest checkpoint and resumes. Because non-deterministic operations (like an LLM generating prose) cannot be safely replayed, their outputs must be explicitly checkpointed so the resumed workflow simply loads the historical output rather than asking the LLM to generate it again.¹²

Durable Execution Record

A robust execution record must contain:

- Workflow ID, Task ID, Step ID, Attempt ID
- Current state and State version (for optimistic concurrency)
- Assigned agent and Lease owner
- Input/Output references (pointers to blob storage³)
- Checkpoint reference, Retry count, Error history
- Budget usage and Deadline

6. Maximum Iteration Limits

Unbounded agent loops pose severe financial and computational risks. Implementing a single

universal iteration limit (e.g., "Max 10 steps") is inadequate, as simple queries require few steps while autonomous coding tasks require dozens.

Limits must be explicitly modeled. *Maximum model invocations* constrain LLM API calls.

Maximum elapsed time enforces SLAs. *Maximum financial cost* tracks exact token pricing.

Maximum delegation depth prevents multi-agent architectures from spawning infinite sub-agents.

These limits must be tiered. *Soft limits* (warning thresholds) trigger a watchdog alert or force a replanning phase. *Hard limits* trigger immediate interception. When a hard limit is reached, the runtime must not terminate violently. It must execute a shielded stop: save the current checkpoint, package all partial outputs, mark the task as Budget Exhausted, and route it to a human supervisor or dead-letter queue.

7. Timeouts and Cancellation

In distributed systems, a timeout does not guarantee that an operation has stopped; it only indicates that the client has stopped waiting.¹ If an agent calls a payment API and the connection times out, the provider's server may still process the charge.

A durable runtime must differentiate timeouts. A *model-call timeout* applies to a single LLM HTTP request. A *step timeout* applies to the reasoning phase. A *queue visibility timeout* dictates how long a worker can hide a task before the broker assumes the worker died.⁶ An *idle timeout* triggers when an agent holds a lease but emits no progress metrics.

Cancellation semantics must be cooperative. When cancellation is requested, the runtime transitions the state to Cancellation in progress. It issues a cancellation token to the worker. The worker must intercept this signal, safely close file handles, record pending receipts to the action ledger, and gracefully exit. If the agent had already initiated a non-idempotent external write (e.g., a database mutation), the cancellation handler must automatically trigger a compensating transaction to reverse it.⁵

8. Retry Policies

Delegating retry logic to the LLM (e.g., passing a stack trace and prompting the model to "try again") is a massive anti-pattern that leads to rapid retry storms and token exhaustion.² Retries must be strictly controlled by the deterministic runtime.

Errors must be rigorously categorized:

- **Transient:** Rate limits (429), network timeouts, 502/503 provider outages. *Action: Retry with delay.*
- **Permanent:** Authorization (401/403), Resource Not Found (404), Invalid Schema. *Action: Fail permanently, route to DLQ.*
- **Policy Violation:** Security filter triggers. *Action: Escalate to human.*

A retry policy record defines the Maximum attempts, Initial delay, Backoff multiplier, Jitter policy, and a Fallback target. If an LLM provider returns a 429 Rate Limit containing a Retry-After header, the runtime must parse and respect it, overriding the default backoff calculation.

9. Exponential Backoff and Jitter

When a major LLM provider experiences an outage, thousands of deployed agents will fail simultaneously. If they retry at fixed intervals, they create a thundering herd that repeatedly overloads the recovering provider API.

Exponential backoff increases the delay multiplicatively, but without jitter, the herd remains synchronized. Decorrelated jitter provides the optimal mathematical distribution to smooth the load curve.

Decorrelated Jitter Backoff Mathematics

The delay for attempt N without jitter is: $T_{\text{delay}} = \min(T_{\text{max}}, T_{\text{base}} \cdot 2^N)$ With decorrelated jitter, the wait time is dynamically calculated based on the previous delay:

$$T_{\text{wait}} = \text{Uniform}(T_{\text{base}}, \min(T_{\text{max}}, T_{\text{previous_wait}} \cdot 3))$$

Pseudocode Implementation

```
Python
import random

def get_decorrelated_jitter_delay(attempt, base_delay, max_delay, last_delay):
    if attempt == 1:
        return base_delay
    upper_bound = min(max_delay, last_delay * 3)
    sleep_time = random.uniform(base_delay, upper_bound)
    return sleep_time
```

Crucially, backoff logic must interact with task leases. If a worker calculates a 120-second backoff delay, but its queue visibility timeout is 60 seconds, the task will be reassigned to a new worker mid-sleep, causing duplicate execution.⁶ The sleeping worker must actively heartbeat to renew its lease while waiting.

10. Idempotent Actions

Idempotency is the mathematical property ensuring that executing an operation multiple times produces the exact same result as executing it once.⁸ This is the only architectural defense

against the distributed crash window: the exact microsecond where an external action succeeds, but the worker crashes before persisting the local receipt.

An *idempotent request* relies on a unique idempotency key. Deduplication tables at the receiver end ensure the operation is processed only once. Operations like reading a file or searching the web are naturally idempotent. Generating a draft, writing a file, charging a card, or deploying code are inherently non-idempotent and require strict wrappers.⁴

Action Ledger Design

Field	Purpose
Idempotency key	SHA-256 hash of (Task ID + Step ID + Action Payload).
Action type	Enumerated action class (e.g., ChargeCard).
Execution status	Pending, Success, Failed, Compensated.
External system ref	Transaction ID returned by the downstream API.
First execution time	Timestamp of initial attempt.
Result reference	Pointer to the blob store containing the exact response.

The Required Sequence:

1. The agent proposes an intent. The runtime hashes it to create the idempotency key.
2. The runtime checks the Action Ledger. If the key exists with Success, it immediately returns the Result reference to the agent without network calls.⁴
3. If absent, the runtime reserves the key (writes Pending).
4. The runtime executes the external API, passing the key in the headers.
5. Upon success, the runtime updates the ledger to Success and stores the receipt.

11. Exactly-Once vs. At-Least-Once Execution

Strict end-to-end exactly-once execution across disparate distributed systems is functionally impossible due to network unreliability. Production runtimes instead combine *at-least-once message delivery* with *effectively-once business behavior*.⁴

If a broker delivers a task to a worker, and the worker crashes immediately after completing the

work but before acknowledging the message, the broker will redeliver the task (at-least-once). The new worker will attempt to execute the exact same external action. Because the runtime utilizes idempotency keys and an action ledger, the duplicate execution is caught locally or at the external API, preventing duplicate side effects (effectively-once).

Realistic execution guarantees for an agent runtime:

- **Queue delivery:** At-least-once.
- **Model invocation:** At-least-once (retries are safe).
- **Artifact creation:** Exactly-once (enforced via versioned object storage).
- **External API side effects:** Effectively-once (enforced via idempotency keys).⁴

12. Checkpointing

A checkpoint is a durable representation of resumable task progress.¹ Rather than relying on transient memory, frameworks like LangGraph utilize PostgreSQL to serialize the state graph at defined intervals.³

Checkpoints must be created strategically: after planning, before external actions (to save intent), after external actions (to save receipts), and before pausing for human input.⁴

Checkpoint Schema Design

A robust schema must separate primitive tracking data from massive LLM context blobs.³

- **Core Data:** Checkpoint ID, Task ID, Sequence, Current state, Remaining budget.
- **Artifact References:** Pointers to Amazon S3 or blob storage holding heavy context manifests, retrieved PDFs, or generated images.¹⁴
- **Ledger References:** Pointers to the Action Ledger for completed tools.
- **Integrity Hash:** A cryptographic hash of the state to detect manual database tampering.

Checkpoint consistency relies on relational atomicity. Updating the checkpoint, rotating the fencing token, and updating the lease must occur within a single SQL transaction.

13. Workflow Resumption

Workflow resumption is the process of recovering a crashed task.¹ It differs fundamentally from a *retry*, which re-attempts a failed step. Resumption reconstructs the exact environment prior to the crash.

The runtime initiates resumption by loading the latest valid checkpoint and verifying its integrity hash.² It checks workflow-definition compatibility to ensure the underlying agent code has not changed structurally. It reconciles completed external actions by cross-referencing the action ledger. It revalidates current context freshness, issues a new fencing token to the claiming worker⁶, and restores pending child-task relationships.

If the workflow code, prompt templates, or tool schemas have changed during the downtime, old checkpoints may become incompatible.³ The runtime must apply automated migration policies, or flag the task for human review to map the old schema to the new state architecture.

14. Partial-Result Preservation

When an agent encounters a permanent failure or exhausts its budget, discarding the progress is highly inefficient. Agents must preserve partial results.

Arbitrary "percentage-complete" metrics are useless. Progress must be quantified by completed plan steps, evidence coverage, and artifact generation. A partial-result record contains the Validation status, Known limitations, and Reusability index. For example, if a research agent crashes after gathering 50 PDFs but before summarizing them, the runtime preserves the vector index. When a replacement agent is dispatched, it mounts the existing index rather than executing 50 duplicate search queries, saving massive token and compute costs.²

15. Dead-Letter Queues

A Dead-Letter Queue (DLQ) isolates tasks that cannot be processed successfully, protecting the runtime from infinite retry loops on poisoned data.

Tasks are dead-lettered due to retry exhaustion, malformed protocol inputs, missing critical dependencies, unresolvable merge conflicts, or repeated loop detection. The dead-letter record preserves the exact state of the failure: the Original task payload, Failure category, Last checkpoint, and a Suggested operator action.

Operators inspecting the DLQ can repair the payload, override a security block, manually resolve a conflict, or archive the task. DLQs require strict alerting; a rapidly growing DLQ indicates a systemic failure in the workflow definition or a permanent provider schema change.

16. Circuit Breakers

While retry policies protect individual tasks, circuit breakers protect the entire runtime ecosystem from cascading failures.⁷

If a specific model provider experiences severe latency, hundreds of agents will hang, exhausting connection pools and memory. A circuit breaker monitors this provider. When the failure rate exceeds a threshold, the breaker trips from Closed to Open. All subsequent agent requests to this provider instantly fail-fast without attempting a network connection.

After a cool-down period, the breaker enters Half-open, allowing a small percentage of requests through as a health probe. If they succeed, it closes; if they fail, it re-opens. Circuit breaker state must be stored in a centralized, low-latency datastore (e.g., Redis) so all distributed workers share a unified view of provider health.

17. Heartbeats and PING/PONG Health Checks

Liveness monitoring is complex in agent systems because a process can be alive and consuming CPU while its task is completely stuck (e.g., trapped in a regex parsing loop or an LLM hallucination loop). The runtime must distinguish between process liveness, agent liveness, and workflow progress.

Workers emit periodic heartbeats (e.g., every 5 seconds). The payload must contain the Worker

ID, Task ID, Timestamp, and critically, the Last progress time and Last completed action. False failure detections occur due to heavy LLM streaming load, network delays, or garbage collection pauses.⁸ Therefore, declaring an agent "Dead" requires missing multiple consecutive heartbeats.

- **Healthy:** Heartbeats arriving, progress metrics updating.
- **Degraded:** Heartbeats arriving, progress stalled.
- **Suspected Dead:** One heartbeat missed.
- **Dead:** Three consecutive heartbeats missed; lease forcefully expired.

18. Watchdog Agents and Supervisory Monitors

Watchdogs observe workflows to enforce policy. They operate on two distinct axes: deterministic and model-assisted.

Deterministic watchdogs are high-speed rule engines. They detect missed heartbeats, expired leases, budget breaches, and exact action repetitions.⁶ They operate instantly without LLM cost.

Model-assisted supervisors analyze semantic context. They read the agent's recent history to detect semantic stagnation, contradictory intermediate results, or task drift. Because model-assisted watchdogs introduce latency and token costs, they must be triggered sparsely—only when deterministic progress metrics stall. A watchdog policy defines the Detection rule, Observation window, and Automatic action (e.g., forcing a strategy change or escalating to a human).

19. Detecting Loops and Stagnation

Agent loops are hidden execution cycles that consume resources without explicitly failing.¹⁰ Detecting them is paramount for cost control.

Loops manifest as repeated tool calls, repeated plan revisions, or circular agent-to-agent conversations. Exact repetition detection (comparing call stacks) is computationally cheap but easily evaded by LLMs that subtly rephrase identical intents.

Semantic Loop Detection solves this via vector embeddings.¹⁰ The runtime converts the agent's recent outputs into embeddings. It calculates the cosine similarity:

$$\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \|B\|}$$

If the cosine similarity between consecutive reasoning steps exceeds a threshold (e.g., 0.95), the agent is semantically looping.¹⁵

Upon detection, the response escalates:

1. Inject a warning into the agent's prompt.
2. Force a strategy change via a supervisor agent.
3. Block the repeated action entirely.

4. Escalate to a human operator.¹⁷

20. Detecting Low-Confidence Output

Model self-reported confidence is notoriously uncalibrated. Runtimes must rely on independent verification to detect low-confidence or hallucinated output.

Signals include missing JSON fields, unsupported claims, failure to satisfy test cases, or excessive hedging ("As an AI language model...").

A result-quality record assesses Requirement coverage and Evidence coverage (can the output be explicitly traced back to the retrieved context?). If the validation pipeline detects low quality, the runtime can automatically request a revision, route the output to a separate specialized model for independent verification, or escalate to a human.

21. Invalid and Malformed Results

When an agent produces malformed output (e.g., truncated JSON, hallucinated tool names), blind retries are inefficient.

The validation pipeline applies syntax, schema, and policy validation. If it fails, the runtime attempts **deterministic repair** (e.g., using regex to extract JSON from Markdown code blocks). If deterministic repair fails, it uses **model-based repair**: passing the broken output and the specific validation error to a smaller, faster model (like Llama 3 8B) instructed purely to fix the syntax. Repairing an output is vastly cheaper than re-running a massive reasoning chain. The runtime must enforce provenance, recording exactly how the original output was modified.

22. Rollback and Compensation Actions

Database transactions can be rolled back; external side effects cannot. An email sent cannot be un-sent. A supplier order submitted via API cannot be rolled back via SQL.

Agent runtimes manage distributed side effects using the **Saga pattern**.⁵ A saga breaks a distributed transaction into a sequence of local transactions. If step 3 fails, the saga executes **compensating transactions** to reverse steps 1 and 2.⁵

A compensation logically reverses a business operation. It does not restore the database to its exact previous state.⁵

- *Original action*: Charge Customer. *Compensation*: Issue Refund.
- *Original action*: Send Confirmation Email. *Compensation*: Send Cancellation Email.
- *Original action*: Deploy Code. *Compensation*: Rollback Deployment.

High-risk compensations (like deleting infrastructure or issuing large refunds) must require explicit human authorization before execution.

23. Provider Outage Fallback

When a primary model provider goes offline, the runtime must failover seamlessly. Fallback must be **capability-based**, not static string matching.

If a task requires 128k context, structured JSON output, and image vision, the runtime queries its model registry for an available fallback satisfying those precise constraints. Switching from

GPT-4o to a local open-source model fundamentally alters behavior. The runtime must recompile the prompt to match the new model's specific template requirements (e.g., ChatML), adjust context limits, and record the failover in the audit trail. Failover must be prohibited for tasks processing sensitive data restricted by regional compliance laws (e.g., failing over from a localized Azure EU endpoint to a generic US provider endpoint).

24. Recovery After Server Restarts

Server restarts stress-test durable execution. Graceful shutdown hooks—which attempt to extend leases and flush logs—are helpful but insufficient against kernel panics or hard power losses.

Upon startup, the Durable Agent Runtime connects to the authoritative database (PostgreSQL) and loads the workflow state.² It identifies tasks marked Executing. It cross-references the lease table. Remote workers will have fresh heartbeats; tasks held by the dead server will have expired leases.⁶ The orchestrator reconciles in-progress external actions against the Action Ledger, rebuilds circuit breaker state, and allows available workers to claim the orphaned tasks using new fencing tokens, resuming execution from the last valid checkpoint.¹

25. Leases, Locks, and Task Claiming

When a distributed queue delivers a task, the worker must hold an exclusive lease.⁶ A catastrophic race condition occurs if Worker A acquires a lease, experiences a 30-second garbage collection pause, and its lease expires.⁸ The queue assigns the task to Worker B. Worker A wakes up and continues processing. Both workers commit data, causing a split-brain corruption.⁸

This is solved using **Fencing Tokens**.⁶

1. The lease manager issues a strictly monotonically increasing token upon acquisition. Worker A receives token 33.
2. Worker A pauses; its lease expires.
3. Worker B acquires the lease and receives token 34.
4. Both workers attempt to write checkpoints to the storage layer.
5. The storage layer checks the token. It accepts Worker B's write (34). When Worker A attempts to write (33), the storage layer sees 33 is older than the current high-water mark and strictly rejects it.⁶ Data corruption is mathematically prevented.⁶

26. Concurrency and Race Conditions

Agent workflows suffer from complex race conditions: two agents updating the same plan, or a retry starting before the original attempt cleanly terminates.

Concurrency controls mandate atomic state transitions. **Optimistic locking** via state-version checks is highly effective. When loading a checkpoint, the worker notes the `state_version` (e.g., 5). When writing the updated checkpoint, it executes a Compare-and-Swap (CAS) query: `UPDATE checkpoints SET data = X, version = 6 WHERE version = 5`. If another agent already updated the state to 6, the query affects 0 rows. The worker detects the stale state, discards its

processing, reloads the fresh state, and recalculates.

27. Human-in-the-Loop Recovery

Deterministic automation cannot resolve ambiguity. A Human-in-the-Loop (HITL) recovery interface is essential for failed, uncertain, or high-risk tasks.

The interface displays the current state, last successful checkpoint, failure reason, partial artifacts, and available compensation options. Operators can approve a retry, supply missing context, edit the prompt, resolve a merge conflict, or approve a high-risk compensation saga. Every human intervention must be cryptographically signed and logged. Decisions regarding policy overrides or un-bounded financial retries must never be delegated to an automated agent.

28. Recovery Policies by Task Type

Different objectives require tailored recovery semantics.

- **Research Tasks:** High tolerance for interruption. Recovery must preserve vector indexes and evidence coverage maps to avoid repeating expensive document retrieval.
- **Coding Tasks:** Highly stateful. Recovery must preserve git branches and test logs. The runtime must re-verify the working tree to ensure human developers haven't pushed conflicting commits during downtime.
- **Data-Processing Tasks:** Requires strict partition checkpoints (like Kafka offsets) to deduplicate output records.
- **External-Action Tasks:** Zero tolerance for duplicates. Recovery mandates idempotency reconciliation before any retry is permitted.⁵
- **Human-Approval Workflows:** Persist the approval request to avoid spamming users with duplicate notifications after a restart.³

29. Recovery Policy Engine

Hardcoding recovery logic creates brittle systems. A rules-based policy engine dynamically determines post-failure behavior.

Inputs include the Failure category, Task type, Attempt count, Remaining budget, and Provider health.

Decision Matrix Pseudocode:

Python

```
def evaluate_recovery(failure, context, budget):
    if failure.category == "AuthError":
        return Action.DEAD_LETTER
    if failure.category == "RateLimit" and budget.attempts < MAX_RETRIES:
        return Action.RETRY_WITH_JITTER
```

```
if failure.category == "ModelTimeout" and circuit_breaker.is_open():
    return Action.FALLBACK_PROVIDER
if context.side_effects_pending:
    return Action.ESCALATE_TO_HUMAN
if budget.cost >= budget.max_limit:
    return Action.SAVE_PARTIAL_AND_PAUSE
```

While LLMs can be utilized to parse obscure error strings, the final routing decision must remain strictly deterministic to ensure systemic safety.

30. Observability for Reliable Agent Runtimes

Observability must map directly to distributed tracing.²² A workflow is a Trace. A task is a Span. Agent loops and tool executions are Child Spans.

Operators must have instant visibility into the last successful step, active providers, and pending compensations. Metrics must track the First-attempt success rate, Mean time to recovery (MTTR), False-positive watchdog rate, and Partial-result salvage rate.

Integration with OpenTelemetry is mandatory.²² Spans must be enriched with GenAI semantic conventions, logging token consumption, model IDs, latency, and the thread_id to correlate backend infrastructure events with the specific agent reasoning loop that triggered them.

31. Audit Logs and Forensic Reconstruction

An immutable audit trail is required for compliance and forensics. The system must record every state transition, tool call, and human approval.

If an incident occurs (e.g., a duplicate email is sent), forensics rely on the audit log. The log will reconstruct the exact microsecond the intent was recorded, the issuance of the idempotency key, the network timeout, the subsequent lease expiration, the fencing token acquisition by the second worker⁶, and exactly why the deduplication check failed. Without an append-only log, diagnosing race conditions across asynchronous agents is impossible. PII must be redacted prior to persistence.

32. Reliability Budgets

Treating LLM invocations as infinite commodities leads to massive cloud overruns. Agent systems require strict reliability budgets.

Budgets define Maximum attempts, Maximum token use, Maximum provider failures, and Maximum cost (e.g., £50 per workflow). Budgets interact dynamically with task priority. If a £10 data-extraction task hits a provider outage, the runtime pauses it. If a £500 critical trading task hits the same outage, the runtime expends its higher budget to instantly route to an expensive, high-availability fallback provider. Budget exhaustion triggers a graceful shutdown, salvaging partial results.

33. Agent-Runtime Data Model

The persistence layer requires a normalized schema for concurrency and isolation.³

Entity	Primary Key	Important Fields	Concurrency Controls
Workflows	id (UUID)	objective, status, budget_limit	Parent lock
Tasks	id (UUID)	workflow_id, status, deadline	Row lock
Leases	id (UUID)	task_id, worker_id, fencing_token	Row lock, Expiry
Checkpoints	id (UUID)	thread_id, state_jsonb, version	Compare-and-Swap
Action Ledger	key (Hash)	task_id, receipt, status	Unique Index
Dead Letters	id (UUID)	task_id, failure_reason, payload	Append-only

34. Durable Agent Runtime Protocol (DARP) Specification

The Durable Agent Runtime Protocol (DARP) establishes a provider-independent architecture, cleanly separating deterministic runtime control from non-deterministic LLM reasoning.

Runtime Boundaries:

- **Deterministic Control:** Manages state transitions, leases, timeouts, and idempotency.
- **Model Reasoning:** Stateless; consumes prompts and returns structured intent.
- **Persistent Storage:** Abstracted interfaces (e.g., PostgreSQL Checkpointer).³

Execution Contract: A task requires a globally unique Task ID, Objective, Tools, Budgets, and Deadlines. The result must output a Completion status, Structured result, Artifacts references, and an External actions ledger.

35. Checkpoint Format Specification

Checkpoints must follow a strictly versioned format to ensure backward compatibility across

updates.³

JSON Schema Representation:

JSON

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "DARP_Checkpoint",
  "type": "object",
  "properties": {
    "checkpoint_id": { "type": "string", "format": "uuid" },
    "thread_id": { "type": "string", "format": "uuid" },
    "sequence": { "type": "integer" },
    "version": { "type": "string" },
    "agent_state": {
      "type": "object",
      "properties": {
        "status": { "type": "string" },
        "budget_used_gbp": { "type": "number" },
        "pending_approvals": { "type": "array" }
      }
    },
    "artifact_refs": { "type": "array", "items": { "type": "string", "format": "uri" } },
    "integrity_hash": { "type": "string" }
  },
  "required": ["checkpoint_id", "thread_id", "sequence", "agent_state"]
}
```

36. Loop-Controller Specification

The Loop Controller orchestrates the lifecycle deterministically.

Durable Loop Pseudocode:

Python

```
def durable_agent_loop(task_id):
    # 1. State Loading & Lease Acquisition
    state, fencing_token = acquire_lease_and_load_checkpoint(task_id)
    if not state: return
```

```

while not state.is_terminal():
    # 2. Watchdog & Limits
    if check_limits(state) or detect_loops_cosine(state):
        state = escalate_or_pause(state)
        break

    # 3. Model Invocation (Circuit Breakers)
    try:
        intent = invoke_model_with_fallback(state.context)
    except TransientError:
        schedule_retry_with_jitter(task_id)
        break

    # 4. Action Ledger & Idempotency Execution
    if intent.is_external_action():
        if not check_idempotency_ledger(intent.hash):
            receipt = execute_tool(intent)
            write_action_ledger(intent.hash, receipt)
        else:
            receipt = get_cached_receipt(intent.hash)

    # 5. State Update & Checkpointing
    state = update_state(state, receipt)
    commit_checkpoint(task_id, state, fencing_token)

    # 6. Heartbeat & Renewal
    renew_lease(task_id, fencing_token)

release_lease(task_id, fencing_token)
return state.result

```

37. Required Worked Examples

1. **Research task interrupted by server restart:** An agent gathers 60% of required PDFs when the pod is evicted. The heartbeat ceases, and the lease expires.⁶ Upon restart, a new worker claims the task. It loads the checkpoint, linking to the blob store holding the PDFs. The agent resumes precisely at 60%, avoiding duplicate searches.¹
2. **Coding agent crashes after creating a patch:** The code patch is generated, but tests crash. The checkpoint saved *before* testing holds the patch. Upon resumption, the runtime reads the patch, restores repository state, and immediately proceeds to unit testing.²
3. **Duplicate email risk:** An email request times out after reaching SendGrid. The worker

retries. Because the intent was logged with an idempotency key prior to the first attempt, SendGrid receives the exact same key, recognizes the duplicate, drops the second email, and returns the original success receipt.⁸

4. **Provider outage:** GPT-4o experiences a major outage. The circuit breaker opens. The task routes to Claude 3.5 Sonnet. The runtime recompiles the prompt context to match Anthropic's schema, validates the output, and completes the task. The audit log flags the capability fallback.
5. **Agent loop:** An agent searches for "Q3 revenue", fails, and searches for "Revenue third quarter". The cosine similarity loop detector scores the vector embeddings at 0.99.¹⁰ It intercepts the third attempt, injects a system warning, and forces the agent to search for annual reports instead.
6. **Invalid structured output:** A local model outputs JSON with a missing bracket. The parser fails. The system passes the broken string to a fast validator model with the error trace. It returns the repaired JSON. Execution continues without regenerating the expensive reasoning chain.
7. **Human approval delay:** A deployment workflow halts, writing an approval request. The server restarts 12 hours later. The state loads as Waiting for Human. The runtime does not resend the Slack notification, as the idempotency key proves it was already sent.³
8. **Compensation:** An automated supplier order is placed. Later, validation discovers the inventory count was hallucinated. The runtime executes a compensating transaction: calling the supplier's cancel_order endpoint referencing the original transaction ID, logically reversing the error.⁵
9. **Two workers claim the same task:** Worker A hangs. Worker B gets the lease with fencing token 42. Worker B completes the step and writes to the DB. Worker A wakes up and tries to write with token 41. The DB strictly rejects token 41. Data corruption is prevented.⁶
10. **Old checkpoint incompatibility:** An agent workflow is upgraded to v2. A paused v1 checkpoint is resumed. The runtime detects the version mismatch in the checkpoint_migrations table.³ It pauses the task for human review to safely map the v1 state into the v2 schema.

38. Failure-Mode Catalogue (FMEA)

Failure Description	Detection Method	Severity	Retryability	Recovery Action
Model timeout	HTTP Exception	Low	Yes (Backoff)	Retry same provider.
Provider	Circuit Breaker	High	No (Switch)	Fallback to

outage	open			secondary provider.
Invalid artifact hash	Integrity validation	Critical	No	Dead-letter, human intervention.
Worker crash	Missed heartbeat	Medium	Yes	Expire lease, new worker resumes. ⁶
Stale worker	Fencing token rejected	High	No	Terminate stale process immediately. ⁶
Stagnation	No metric progress	Medium	Yes	Supervisor forces strategy shift.
Budget exhaustion	Cost limit breach	High	No	Save partial state, halt workflow.
Compensation failure	Exception during Saga	Critical	No	Immediate escalation to operator. ⁵

39. Technology Comparison

Technology	Reliability Model	Operational Complexity	Suitable for	Restart Recovery
Temporal	Event Sourcing / Replay	Very High	Massive distributed scale	Excellent (deterministic replay)
LangGraph (Postgres)	State Checkpointing	Moderate	Python Agent ecosystems	High (loads last snapshot) ³

Kafka (Sagas)	Event stream choreography	High	High-throughput events	Excellent (offset replays) ⁵
Redis Queues	At-least-once delivery	Low	Simple local workers	Poor (volatile state loss)
AWS Step Functions	Managed state machine	Low (Managed)	Serverless orchestration	Excellent (fully managed)

Evaluation: LangGraph provides a robust, Python-native checkpointing paradigm tailored for LLM states³, but Temporal remains the gold standard for orchestrating massive, cross-language deterministic workflows.

40. Testing Strategy

Rigorous Chaos testing separates prototypes from production runtimes.

- **State-machine tests:** Assert invalid transitions throw exceptions.
- **Fencing-token tests:** Deliberately pause a worker, spawn a second, and verify the first worker's DB write is rejected.²¹
- **Crash window tests:** Kill the worker process milliseconds *after* an external API returns 200 OK, but *before* the local DB commit. Ensure the subsequent retry utilizes the idempotency key and recovers correctly.⁸
- **Compensation tests:** Inject a failure late in a Saga workflow and verify all preceding steps successfully execute their compensating transactions.⁵

41. Reliability Invariants

Invariants are absolute systemic truths:

1. **A completed step is never silently reverted.**
2. **Every external action must possess an idempotency record.**
3. **A worker cannot commit state after losing its lease.** (Enforced via fencing tokens in SQL: UPDATE... WHERE token >= X).
4. **A dead-lettered task cannot restart without explicit human intervention.**
5. **Human approval cannot be reused if the payload materially changes.**

42. Anti-Patterns

Anti-pattern	Risk	Safer Alternative
State only in LLM context	Crashes erase all progress.	Durable execution

		checkpoints. ¹
Retrying every error	Infinite loops, billing spikes.	Classify errors; only retry transient ones.
Locks without fencing tokens	Zombie workers corrupt DB.	Fencing tokens checked at storage. ⁶
Deleting failed tasks	Loss of forensic data.	Route to Dead-Letter Queue (DLQ).
Asking LLM to fix syntax loop	Context window exhaustion.	Deterministic parsers or specialized models.

43. Decision Frameworks

Should this failure be retried?

1. Is the error code in the transient list (e.g., 429, 502, 503, Network Timeout)? -> Yes.
2. Is the error code terminal (e.g., 401, 403, 404, Validation)? -> No.
3. Has the maximum retry count been exceeded? -> No.

Is compensation required?

1. Did the task fail permanently? -> Yes.
2. Did any previous step in this workflow execute an external side effect? -> Yes.
3. Therefore, *Compensation is required.*⁵

44. Required Architecture Diagrams

High-Level Durable Agent Architecture

Code snippet

flowchart TD

subgraph Orchestration Layer

A --> B

A --> C

end

subgraph Execution Layer

B --> D[Agent Loop Controller]

D --> E[LLM Provider]

D --> F

```

end
subgraph Persistence Layer
  D <--> G
  G --> H[Checkpoints]
  G --> I[Action Ledger]
  G --> J
end

```

45. Implementation Blueprint

Level 1: Restart-safe Local Runtime

A single orchestrator using SQLite or PostgreSQL. It utilizes explicit task states and step-boundary checkpoints.³ If the server is restarted, the agent does not repeat completed steps.

Level 2: Reliable Single-server Fleet

Introduces multiple local workers. Requires task leases, heartbeats, exponential backoff, and an action ledger for idempotency. Dead-letter queues protect against poisoned inputs.

Level 3: Distributed Runtime

Spans multiple servers. Requires a durable message broker, shared object storage for artifacts, OpenTelemetry tracing²², and crucial *fencing tokens* to prevent split-brain processing.⁶

Level 4: Production-grade Multi-provider Runtime

Adds capability-based provider routing, automated checkpoint migration, global circuit breakers, rigorous Saga-pattern compensation workflows⁵, and cosine-similarity loop detection.¹⁰

46. Recommended Version 1.0 Architecture

For a small but serious agent fleet, prioritize robust persistence over complex distributed choreography.**Architecture:**

- **Database:** PostgreSQL serves as the authoritative state, storing Workflows, Tasks, Leases, Checkpoints, and the Action Ledger.³
- **Broker:** PostgreSQL-backed job queue to avoid managing a separate Kafka cluster.
- **Checkpoints:** Step-boundary checkpoints saving intermediate JSON state.⁴
- **Concurrency:** Task leases backed by database row-locks and strictly enforced fencing tokens.⁶
- **Storage:** Amazon S3 for immutable artifact storage (files, code, massive context windows) to keep PostgreSQL lean.³

Avoid multi-region replication, complex event-driven choreography, and custom workflow

engines until transaction volume absolutely necessitates them.

Required Conclusions

- **What is the safest basic loop for a production agent?** The Plan-Execute-Verify loop, underpinned by step-boundary checkpointing and deterministic validation.
- **Should agent state live in prompts, memory, a database, or all three?** State must live authoritatively in the database (Durable Execution). Memory and prompts are strictly derived views of that database state.²
- **What must be persisted after every agent iteration?** The reasoning intent, the tool input, the external receipt, and the budget consumption.
- **How often should checkpoints be created?** At step boundaries: before and after non-deterministic external operations, and before waiting for human input.³
- **What is the difference between retrying and resuming?** Retrying attempts to execute a failed step again. Resuming reconstructs the environment from a saved checkpoint after a system crash to continue execution.¹
- **How should a task recover after a server restart?** By identifying orphaned leases, claiming the task with a new fencing token, loading the latest checkpoint, reconciling the action ledger, and continuing.¹
- **Which errors should never be retried automatically?** Policy violations, 401/403 Auth errors, poisoned inputs, and unrecoverable schema validation failures.
- **How should external actions be made idempotent?** By generating a deterministic hash of the intent, checking it against an action ledger, and passing it as an idempotency key to the external API.⁸
- **Is exactly-once task execution realistically achievable?** No. Network failures make it impossible.
- **What guarantee should an agent runtime promise instead?** At-least-once task delivery combined with effectively-once business side effects via idempotency and deduplication.⁴
- **How should loops and stagnation be detected?** By monitoring heartbeats for lack of metric progress, and using cosine similarity on embeddings of recent actions to detect semantic repetition.¹⁵
- **How should low-confidence results be handled?** Through deterministic validation, deterministic repair, fallback to stronger models, or escalation to human review.
- **What should happen when a provider is unavailable?** The circuit breaker opens, and the task routes to a fallback model possessing the exact same required capabilities, automatically recompiling the prompt format.
- **When should partial work be returned?** Whenever budget is exhausted, deadlines expire, or permanent failures occur, to ensure invested compute yields salvageable artifacts.
- **When should a task enter a dead-letter queue?** Upon exhaustion of retry budgets, detection of infinite loops, or encountering un-parseable poisoned inputs.
- **When is rollback possible, and when is compensation required?** Rollback is only

possible for local, uncommitted database transactions. Compensation (Sagas) is strictly required to logically reverse any completed external side effect.⁵

- **What is the role of heartbeats, leases, and fencing tokens?** Heartbeats prove a worker is alive. Leases grant exclusive processing rights. Fencing tokens strictly prevent crashed-then-revived workers from corrupting state after their lease expires.⁶
- **Which recovery decisions must remain deterministic?** Retry loops, backoff math, budget enforcement, lease management, idempotency checks, and state transitions.
- **Which recovery decisions may use an AI model?** Semantic loop detection, syntactic JSON repair, and dynamic replanning after an evaluated failure.
- **What is the smallest reliable runtime a new agent fleet should build?** A single-server PostgreSQL-backed workflow engine utilizing explicit states, step-checkpoints, leases with fencing tokens, and an idempotency ledger.³

Works cited

1. Durable Execution for AI Agents | blog - inference.sh, accessed July 25, 2026, <https://inference.sh/blog/agent-runtime/durable-execution>
2. Durable Task for AI Agents - Azure | Microsoft Learn, accessed July 25, 2026, <https://learn.microsoft.com/en-us/azure/durable-task/sdks/durable-task-for-ai-agents>
3. Scaling LangGraph's Postgres Checkpointer in Production • Tadeo ..., accessed July 25, 2026, <https://tadeodonegana.com/posts/scaling-langgraph-postgres-checkpointer/>
4. Durable Execution for AI Agent Runtimes: Checkpointing, Replay ..., accessed July 25, 2026, <https://zylos.ai/research/2026-04-24-durable-execution-agent-runtimes/>
5. Saga Pattern for Microservices Explained - Conduktor, accessed July 25, 2026, <https://www.conduktor.io/glossary/saga-pattern-for-distributed-transactions>
6. System Design: Distributed Lock Service | Algoroq, accessed July 25, 2026, <https://algoroq.io/system-design/design-distributed-lock/>
7. Saga Design Pattern - Azure Architecture Center | Microsoft Learn, accessed July 25, 2026, <https://learn.microsoft.com/en-us/azure/architecture/patterns/saga>
8. Distributed Locks: Redlock, Fencing Tokens & Advisory — AppScale ..., accessed July 25, 2026, <https://appscale.blog/en/blog/distributed-locks-redlock-fencing-tokens-postgres-advisory-2026>
9. Compensating Transactions -SAGA - Medium, accessed July 25, 2026, <https://medium.com/@surezms/compensating-transactions-saga-5d28d49b2f0c>
10. Unsupervised Cycle Detection in Agentic Applications - arXiv, accessed July 25, 2026, <https://arxiv.org/pdf/2511.10650>
11. Agent workflow reliability is not a checkpointing problem ... - Moltbook, accessed July 25, 2026, <https://www.moltbook.com/post/92ad5dec-9264-4d04-a723-d84e902e8ef8>
12. Durable Agents - Restate Docs, accessed July 25, 2026,

- <https://docs.restate.dev/ai/patterns/durable-agents>
13. LangGraph State Management: Complete 2026 Guide | Dailyaiworld, accessed July 25, 2026, <https://dailyaiworld.com/blogs/langgraph-state-management-2026>
 14. LangGraph State Management: Checkpointing & Recovery | ActiveWizards, accessed July 25, 2026, <https://activewizards.com/blog/langgraph-state-management-checkpointing-recovery-and-the-persistence-layer-decision/>
 15. Navigating the Labyrinth of Agent-to-Agent Communication – Preventing Infinite Loops in Multi-Agent Systems - NeuralTrust AI, accessed July 25, 2026, <https://neuraltrust.ai/blog/a2a-loop>
 16. What is cosine similarity and why is it used in semantic search? - Milvus, accessed July 25, 2026, <https://milvus.io/ai-quick-reference/what-is-cosine-similarity-and-why-is-it-used-in-semantic-search>
 17. Stop the Loop! How to Prevent Infinite Conversations in Your AI Agents - DEV Community, accessed July 25, 2026, https://dev.to/alessandro_pignati/stop-the-loop-how-to-prevent-infinite-conversations-in-your-ai-agents-ekj
 18. Building Distributed Resilience: The SAGA Pattern for Financial Applications, accessed July 25, 2026, <https://dev.to/rpi1337/building-distributed-resilience-the-saga-pattern-for-financial-applications-51cf>
 19. Orchestration SAGA - Compensating Transactions. Introduction. - YouTube, accessed July 25, 2026, <https://www.youtube.com/watch?v=SM20SAb3JOE>
 20. Pattern: Saga - Microservices.io, accessed July 25, 2026, <https://microservices.io/patterns/data/saga.html>
 21. How to do distributed locking — Martin Kleppmann's blog, accessed July 25, 2026, <https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html>
 22. OpenTelemetry for AI Observability: What It Covers and Where It Stops | Fiddler AI Blog, accessed July 25, 2026, <https://www.fiddler.ai/blog/opentelemetry-ai-observability-guide>