AI Agents Need Operating Systems, Not Just Better Models

Why the Next Frontier of AI Reliability Is a Software Engineering Problem, Not a Modeling One

By Nishil Pathak

The problems faced in deploying AI agents in production environments largely stem from engineering issues due to the lack of an underlying sound runtime environment, as opposed to model capacity limitations. This paper argues that the current generation of agent frameworks have solved the problem of composition but do not provide a runtime environment, and insufficiently address process isolation, resource management, error handling, state durability, and provenance. Building upon concepts drawn from operating system design, saga, runtime validation, and transactions, this paper describes six reliability properties that must be fulfilled by a sophisticated agent runtime environment. It then examines how the current generation of agents including LangChain and orchestration tools such as Temporal and Airflow fall short, before concluding with actionable recommendations to deploy AI agents in a reliable manner.

Key Takeaways

AI agent reliability issues in production are a result of problems in software engineering due to the lack of adequate runtime infrastructure, and not the lack of capabilities of models. Solving such problems requires systems engineering methods, not better prompts or bigger models.

Frameworks and runtimes solve different problems. Framework defines conventions on composing agent behavior, while runtime manages agent life cycle, resource governance, faults isolation and recovery capabilities. Modern agent frameworks work at the framework level and do not affect the underlying runtime layer.

Operating systems provide abstractions that correspond to agent execution and include process isolation, resource governance, structured error hierarchy, state persistence and observability of execution. Modern agent infrastructure does not have these features by default.

Long horizon tasks require agent's ability to use the saga pattern for compensation of partial failure based on known checkpoints and therefore avoid restarting the whole task from the start.

Structured provenance that represents a queryable causal relationship between inputs, steps of agent reasoning and its output is necessary for agents working in regulated industries or making decisions in consequence. It should be provided by the runtime.

There is a strong assumption in the field of AI agent development that reliability of agents can be achieved solely through the development of better foundation models which means parameterizing them more, giving them more training data and aligning them better. This assumption is not only wrong but architecturally naïve and leads to production problems caused by lack of software infrastructure rather than quality of the models.

Developing companies that build production agent infrastructure have to learn again the lessons which were learned by the operating systems and distributed systems community decades ago. Software that makes decisions, performs actions, manages resources and interacts with other systems in a dynamic environment needs not just a smart central part but also resource

management, faults isolation, recovery capabilities, provenance tracking and observability. It needs the operating system-like substrate to run the agent on top of.

This gap in agent engineering is not the modeling gap but the systems gap, and it should be closed through careful consideration of operating systems design, distributed systems theory and runtime verification, not waiting until the new generation of models internalizes these features into their parameters.

The Agent Deployment Reality

Disregarding benchmark leaderboards, this article is about the behavior of agents in production. A serious engineering deployment does not consist merely of a chatbot with added tools; it perceives the environment, holds state or memory, chooses from a set of available actions, performs them with observable external side-effects, and repeats this cycle over an extended period of time under minimal supervision. This description echoes well with the definition of a long-running process in a constrained and multi-tenant environment used by operating systems folks.

Engineering challenges faced are not particularly esoteric – process isolation, resource management, scheduling, error handling and recovery. The novelty is in the fact that the decision maker at the heart of it all is a probabilistic model rather than a deterministic program, which turns reliability considerations into something for which existing infrastructure is ill-equipped.

The situations described below are composite examples – patterns seen repeatedly in production deployments of agents, reconstructed to demonstrate mechanisms of failure rather than to point fingers at any one particular incident. As an example, let’s consider a deployment of an agent that needs to audit a company's software dependencies, find vulnerable packages, and submit remediation tickets. An agent works for several minutes, querying the package databases, parsing manifests, doing the reasoning about transitive dependencies, and writing tickets to an external system. In the middle of its execution, the network partition produces a malformed response from the dependency query. Being unable to check if this is a valid response due to lack of a contract for its tool calls, the agent treats the malformed data as a valid input, and continues to work. This leads to incorrect tickets being submitted – non-existent vulnerabilities found; real ones ignored.

The reason behind this failure had nothing to do with the probabilistic model itself – it performed correctly, doing reasoning it was trained to do. The problem was with a lack of a runtime environment, which would detect the anomalies in tool responses, enforce the data contracts, handle partial failures, and stop making further decisions until the agent gets a clean slate. This is a software engineering failure, and it needs a software engineering solution.

It's important to note that not all deployments start with the agent exposing itself to as much risk right away – short-horizon agents, with limited tool capabilities, and with human review of their actions before making any external writes are considerably safer. It's not suggested that all agents need the runtime layer from day one. The point is that as agents get longer horizons, more powerful toolsets, and less human supervision, the cost of omission of runtime infrastructure keeps getting higher, and mistakes harder to unwind.

- **Practical Engineering Takeaways:** Signs that your infrastructure has outgrown its current design are agents performing writes to external systems without pre-execution checks, incidents where reconstruction of the events requires cross-service log analysis, and failed runs that have no recovery options apart from starting afresh. Common mistake is in

confusing prototypes running reliably on pristine test inputs and production-ready agents – an agent performing well on the latter tends to misbehave when tool responses become unreliable and environmental state unpredictable.

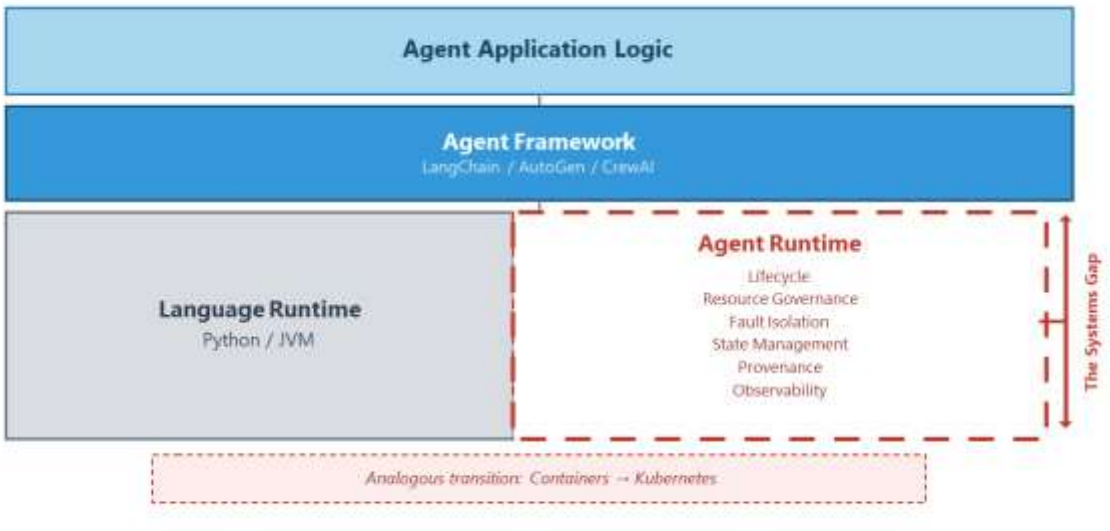

Figure 1. Today's AI agent stacks offer the framework layer for agents' composition. However, there is no equivalent of a runtime layer. The reliability gap is between the framework offerings and production execution environment requirements, similar to how the reliability gap existed in container deployment world before the emergence of Kubernetes.

Why Frameworks Aren't Enough

Initially, the natural response from many teams would be pointing out the growing list of agent frameworks such as LangChain, LlamaIndex, AutoGen, CrewAI, Semantic Kernel which solve the problems of behavior composition, tool calls management, and multi-agent workflow orchestration. Even though these frameworks provide great benefits, they are frameworks, not runtimes, and this fact has major production consequences.

Framework can be defined as a convention that imposes certain code structure and interfaces. Runtime, on the other hand, can be seen as an execution environment that manages program life cycles, enforces resource constraints, catches system level exceptions, separates concurrent workloads, and ensures process execution behavior. This distinction resembles one between web framework and an operating system. For instance, Django defines the structure of a web application, while Linux decides whether an application executes, how long does it run, what resources are allocated, and what happens in case of crash.

During production, there are at least three failure scenarios, all pointing at the lack of a proper runtime:

Tool call might get stuck in an infinite loop with no infrastructure-level timeouts;

Agent can grow its memory usage without limit without having any resource governors;

Auditability of actions will be impossible with no provenance logging in case of an incident that takes weeks to resolve;

These frameworks are quite sufficient in solving composition issues but it's not enough to assume that it's enough for production use.

The Kubernetes comparison is valid here. Before the Kubernetes era, teams managed container lifecycles with shell scripts and custom orchestration tools. While containers worked just fine, their lifecycle management – health checks, resource enforcement, scheduling, and recovery – was far from perfect. Kubernetes didn't make containers smarter; it just added the runtime layer allowing scalable operability. Agents need a similar treatment.

- **Practical Engineering Takeaways:** Before inventing a custom runtime, try to analyze how much benefit you can get from four targeted controls, which can be put above your existing framework: output validation on each tool call, hard step-count ceiling enforced by the orchestrator, circuit breaker for external API calls, and a dry run before executing any writes. None of these replaces a runtime layer, but rather a first step towards having one. Introducing the runtime layer adds real operational complexity and for low-consequence internal tools with high human oversight the overhead might be too much to handle. For agents with external write access or compliance requirements, however, the opposite tends to be true.

The Missing Abstractions

Every abstraction used by operating systems for controlling traditional processes has its counterpart in agents' execution. Comparing the two will demonstrate the existing gap in modernity.

- **Process isolation and sandboxing.** An operating system protects one misbehaving process from affecting the other processes' resources while the counterpart in agent world is action isolation. For example, a support agent and a billing agent both using a shared memory backend may have a buggy write in the support agent that corrupts the billing context for unrelated sessions. While container runtimes use namespace isolation, there is no default way of handling this in agents.
- **Resource management.** Long running agents consume significant number of tokens, compute and API quotas. Running one agent on a shared platform and making it parse a large document corpus will drain all API quotas of an organization and will leave all other agents disabled. While Kubernetes controls that with resource requests and limits, agents need similar control mechanism.
- **State management and persistence.** When an agent analyzing database migration crashes during some operation it will restart it from scratch losing tokens for the same actions and making unnecessary calls to the tools that already succeeded. Agent state needs filesystem abstractions that operating systems provide.
- **Error handling and recovery.** Current agent systems don't differentiate between transient errors (such as HTTP 429 requiring backoff and retry), persistent authorization failures (such as 403 that requires escalation) and semantic errors of the agent reasoning which require re-planning. This conflation leads either to agents that retry on unrecoverable errors until budget exhaustion or to silent ignoring of the errors which should stop the whole operation.
- **Observability and auditability.** When a production agent makes a mistake in interpretation of a security alert and escalates an incident response autonomously to critical, discovering why this happened means going through logs and making inferences what

window of information was seen by the agent. This is not debugging anymore, but an archaeology of the agent.

- **Practical Engineering Takeaways:** If only one part can be addressed currently, then error classification should be the priority. Give at least three error classes to every tool — retryable, non-retryable and requires-human-review — and implement this classification in the tool wrapper and not the prompt. The biggest problem in this area is the conflation of retrying with fault tolerance. Retrying solves one failure mode, but doesn't help with partial success cases, semantic mistakes in the tool output and state corruption after crash.

Figure 2. Every operating system abstraction has a direct counterpart in agent execution. Right column lists features that have to be implemented in the agent infrastructure explicitly; nothing is given for free by any framework currently.

Why the Reliability Challenge Is Structurally Different

A traditional program execution is deterministic due to its inputs and code. Decisions made by an agent are based on probabilistic inference among billions of parameters and therefore, identical inputs will lead to a different result at different times. Such non-determinism allows us to explain the agent's flexibility and why static verification approaches don't apply here.

In this situation, we have a direct application of runtime verification [1] – we don't analyze code of an agent but check the behavior of a system against certain predefined properties and intervene if any deviation happens. In case of an agent, such property is to define invariant of actions performed, pre- and post-conditions of tool invocation and output constraints and then verify during execution. Any violation will result in an intervention – pausing execution, asking for human approval, reverting performed actions or restarting a plan from a known good state.

Long-horizon tasks reveal a secondary structural problem where the saga pattern will help [2]. Saga breaks a long operation into local operations with corresponding compensation actions, which means that if there was a failure at step five out of twelve, then it will be rolled back in reverse order, instead of having some inconsistent state that should be cleaned up manually. In

case of an agent that performs multiple stages of code review, test generation and pull-request creation, it can also define the compensatory actions on each step.

Temporal's durability and activity retry policy cover one of the parts of it [3]. The problem is not an execution engine but Temporal's approach to the workflow graph that should be statically defined. Dynamic agent behavior can be modeled by making planning loop an activity that generates sub-activities dynamically; however, such approach requires quite a lot of engineering work and results in harder-to-analyze workflows than traditional ones.

- **Practical Engineering Takeaways:** When microservices are orchestrated with Temporal, agent workflows are the natural next step. The primary thing to decide on is the frequency of state saving: saving after every LLM call minimizes progress loss but consumes more memory. The reasonable compromise is saving state after completion of each tool call, without the trouble of saving after every request but still having a state to roll back from. Verification adds additional cost and someone will have to define those and keep them up-to-date. Start with a basic verification regime that denies only obviously unsafe actions and gradually expand verifications as agent gets productive tasks.

Where Do Workflow Tools Fall Short?

Using agents inside orchestration engines like Apache Airflow, Temporal, or Prefect is superior to relying solely on application-level code. However, there are architectural limitations here because such engines assume that the control flow graph is known before the execution; meanwhile, agent's execution graph emerges in the course of its execution as it understands what to do next. Airflow expects the whole task graph beforehand; accordingly, when an agent fails as part of an Airflow task, Airflow signals a task failure but cannot determine where exactly did the agent fail, whether the failure happened at some specific reasoning step, or the output of some tool call was incorrect, etc.

Temporal offers better support of dynamic execution but was designed for human-authored workflow code rather than workflows in which what comes next depends on the output of a probabilistic model. Debugging by replaying a Temporal workflow assumes that the same series of actions produces the same result, which is untrue for a case where step six depends on the output of the language model at step five unless additional measures are applied.

- **Practical Engineering Takeaways:** Temporal should be used for outer orchestration, while the agent's reasoning loop should be represented as one activity with its own checkpointing, saving of the agent's state to the external storage after each tool call. Each extra layer of orchestration brings in additional overhead and failure points. An agent with human review in the end may not need Temporal; however, a multi-step unattended DevOps agent with write access to production definitely needs Temporal.

The Provenance Problem

Provenance is a comprehensive structured auditable record of why the agent made certain decisions, not just what decisions were made. This difference is crucial here because, while logs are a chronologically ordered set of events, provenance establishes the causal relation between inputs, reasoning steps, and outputs allowing to reconstruct the entire decision-making process at a later point in time [4, 5].

Imagine a security remediation agent that fixes misconfigured IAM policies. During two hours it makes 47 changes; afterwards, an audit finds out that three of these changes have caused new problems rather than mitigated old ones. Without provenance, finding out which inputs affected

these specific decisions would require manual reconstruction of the agent's context at that moment, which is tedious and unreliable. With a provenance graph, one could directly query which tool outputs corresponded to incorrect decisions, thus reducing the amount of effort for root cause analysis from days to hours.

In regulated industries, the stakes are higher: financial institutions using agents to make automated credit decisions or to produce regulatory reports need to justify automated decisions to auditors; similar rules apply to health care companies. In this context, provenance is mandatory because it is a compliance requirement and its absence means legal risk that most companies do not know how to quantify yet.

Flat log of tool calls is not a provenance record. For a record to be a legitimate one, it should establish a causal relationship between each action and the preceding reasoning step, tool outputs and input context that informed that decision.

- **Practical Engineering Takeaways:** Without a comprehensive provenance graph, two small changes will bring a lot of value: give each agent run its own execution id that gets attached to all the tool calls, LLM invocations and external writes and record the input context of the agent at each decision point, not only the final output. If incident postmortem talks about something "probably seen by the agent", this is a problem that can be prevented, not fixed.

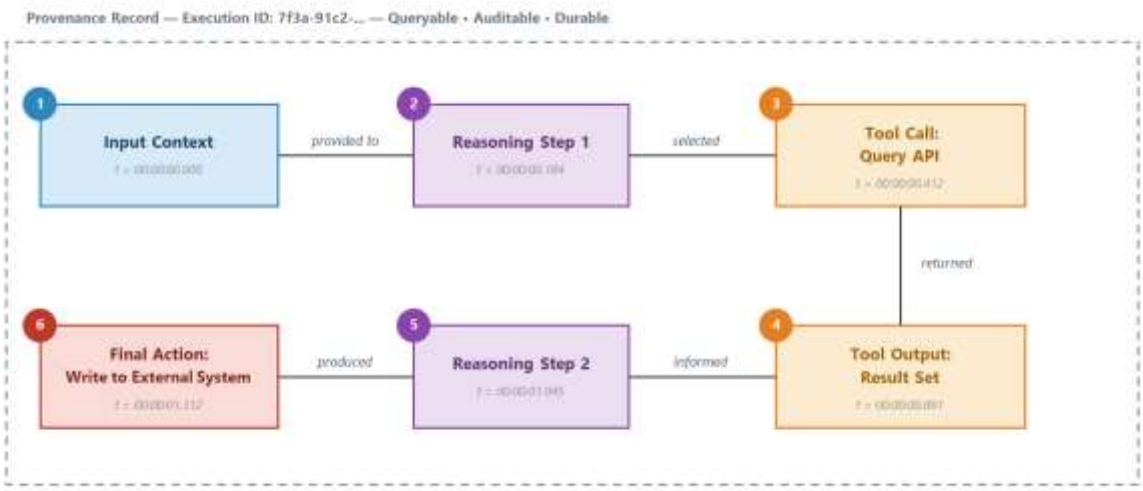

Figure 3. Provenance graph of a single decision cycle of an agent includes the links between input context, reasoning steps, tool calls, tool outputs and final actions.

Multi-Agent Coordination and Its Engineering Challenges

Since multi-agent systems are distributed in nature, they inherit all the possible failure scenarios that are typical for distributed systems [6] plus two more: message loss and out-of-order delivery. Any agent can fail during its execution, taking away partial results, thus leaving the other agents to work with the subset of information that was available at their disposal. Shared memory backends are prone to race conditions, which require some sort of coordination.

An example of a pattern that is quite common in the code review pipeline implemented with multi-agent systems is the following: the planner agent breaks down the code change and assigns its parts to particular reviewers, while an aggregator generates the overall review from the results of these individual reviewers. In case one of the reviewers failed during the process of

generating its reviews but had started writing them into the shared memory, the aggregator might see an inconsistent state and will produce the review summary without the problems that were found by the failed agent. As a result, the code change goes forward with approval even though there were plenty of issues in it. This scenario is called dirty reads in databases and is solved there by using transactions that write to an isolated memory space and commit only after their successful completion, thus allowing to be seen only after committing [7]. The same approach should be applied to agents, that is, the writes should be treated as transactions for concurrency control and the state generated by the agent should become visible only after being committed.

For example, Redis guarantees the transactional consistency for individual keys but does not provide such guarantee across multiple keys, unless the operations are wrapped in the MULTI/EXEC block. It is up to the application to make sure this is enough for its access pattern, just like in the case with agents.

- **Practical Engineering Takeaways:** Before implementing any multi-agent system that uses shared state, dedicate at least half an hour to defining what consistency guarantees the agents need and how it compares to the guarantees provided by the storage layer. Otherwise, one would get agents that work correctly sequentially but exhibit distributed-system bugs when executed concurrently. Just reaching the statelessness of individual operations would not help if the overall system state is stored in the shared memory, queues or a storage backend.

Production Reliability Requirements

The mature agent execution environment requires six requirements of reliability. Put simply, these requirements represent the degree of alignment between existing infrastructure and production needs:

- **Safety checks:** Checking of suggested actions against defined safety and correctness rules before executing, with blocking or logging of any violation.
- **Failure-aware execution with semantic classification:** The runtime recognizes transient failures, persistent failures, and reasoning semantic errors, applying the right policy to restore from each type of the failure. A universal retry policy won't do as a default solution.
- **Recovery-capable execution state:** Persisted state in transactionally consistent tool-call-boundary scope that allows recovery of the execution to the last checkpoint after any failure.
- **Provenance structured capture:** Queryable causality record for each run connecting input context, reasoning, tool choice, tool output, and action.
- **Resource governance with enforcement:** Limited tokens, API calls, memory, and external quotas, and pre-defined behavior when these limits are close to being reached.
- **Observability structured:** Telemetry compatible with the observability stack currently used by ops.

No one of these requirements requires a better foundation model. All of them, however, rely on software engineering infrastructure currently rebuilt by every team independently without established common standards and patterns.

Practical Steps Engineering Teams Can Take Today

Currently, there is no agent runtime ready for production; therefore, all necessary features should be implemented on top of the current set of tools. The approach does not involve platform rework, but it does require engineering effort to get right.

- **Check against a contract all responses from the tool:** before using any value provided by the tool, verify that it fits some anticipated schema. If it doesn't, the tool call should be considered failed. Malformed data that is "almost valid" will be much harder to debug in the future.

Minimalistic implementation example is provided below:

```
from pydantic import BaseModel, ValidationError
class PackageQueryResult(BaseModel):
    package_name: str
    version: str
    known_vulnerabilities: list[str] = []
def call_tool_safely(tool_fn, *args, **kwargs):
    raw = tool_fn(*args, **kwargs)
    try:
        return PackageQueryResult.model_validate(raw)
    except ValidationError as e:
        # Fail closed: expose a structured and retryable error
        # rather than letting the agent operate on the malformed data.
        raise ToolContractError(tool_fn.__name__, errors=e.errors())
```

- **Assign a unique correlation ID for every execution:** assign a unique ID to every agent execution and include it into all log lines. It is easy to implement and greatly simplifies debugging failures in cross-service contexts.
- **Store a checkpoint after every tool call:** after every successful call, store a checkpoint in a durable storage (e.g., Redis, database, or alike). It enables resuming from the middle of the run, speeds up the development process and allows for resuming for long-term execution in the production.
- **Classify failures before attempting to recover:** different types of failures call for different approaches to recovery. Network timeout might be the case to retry while permissions error calls for manual intervention. Classification of the failures should be done before any attempt to recover.
- **Limit the number of steps and explain why the agent stopped:** agents should stop when they get close to exhausting their allocated budget of steps, with a description of why the execution stopped and if possible, a suggestion for human involvement. This helps identify cases of low efficiency sooner.
- **Require human confirmation for all irreversible operations:** classify all operations as read, reversible writes and irreversible writes. Every irreversible operation needs human confirmation to be executed – this becomes a cheap yet effective safety measure in production.

- **Monitor resource consumption for every run:** track tokens spent, API calls and execution duration of every run, with alerts for anomalous resource consumption compared to the baseline.

All of this makes agent systems good enough to be used in production without a special runtime, and implementable on top of the current infrastructure. Implementation of such features is significantly easier now compared to their implementation in the future when more infrastructure changes would be needed.

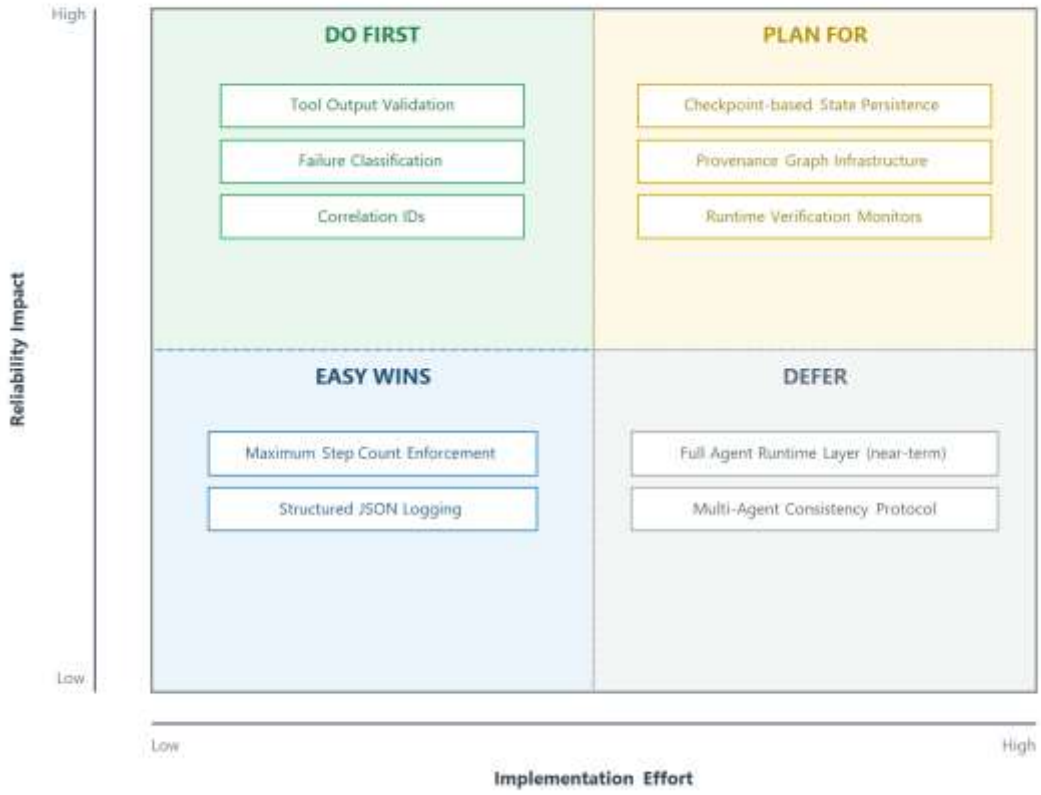

Figure 4. Reliability practices for agent deployments, divided by implementation difficulty and the value added to the agent infrastructure.

The Path Forward Is Systems Engineering

Based on experience with multiple paradigm shifts in the field we can see that the usual pattern is being repeated: innovation followed by reliability challenges, and then a lengthy period of making it reliable in the field. Web services, microservices, and other technologies went through the same path and so will the agents and their wrappers.

No new fundamental mechanisms need to be discovered; decades of scientific and practical researches proved how to make infrastructural software reliable. The creation of TCP/IP protocols introduced network reliability, while transaction theory introduced database reliability. In both cases, systems thinking was applied to the domain; this kind of thinking is exactly what is needed for agents. Particularly, people who would pay attention to the reliability of agent-based infrastructure would gain huge advantages for the industry downstream.

Closing: Recommendations, Open Questions, and a Challenge

Three pieces of advice to teams building agents in modern times:

- **Consider wrapper tooling to be part of the core infrastructure instead of just a glue component:** there is always a chance of failure, resource inefficiency, or data exfiltration at any point of interoperation between two agents. This means that in the development of the wrappers, the focus should be on such things as schema validation, error classification, retries, and telemetry instead of them being optional.
- **Define the reliability standard before the deployment:** define and reach a consensus about the reliability design specification of any agent participating in production-related decision-making with regards to observability needs, resource limitations, and failure semantics of both the agent and the surrounding system. Though developing the spec requires a serious effort, it is much easier than dealing with incidents caused by already deployed wrapper.
- **Invest into provenance in proportion to the consequence of any mistakes that may occur:** some agents require nothing more than a set of correlation identifiers, but the others, especially those who interact with safety-critical systems, need a more thorough auditing and tracing to comply with their trust and safety obligations.

Three questions that need to be solved:

- What kinds of failure-handling design patterns should be implemented at the level of agents themselves and which tasks should be left for wrappers? Now it is up to each team to solve this problem.

- What consistency model should be applied to agents sharing memory space and how dependent it will be on interaction patterns? Right now, there is no clear advice about this.

- How can provenance be standardized so that cross-agent comparison becomes possible? Without the standard schema it is impossible to compare provenance information from different frameworks, making fleet-level audits unfeasible.

A forward-looking question to start conversations around production agents: If a software agent responsible for executing the seventh out of twelve steps of the process fails and cannot continue working further because of it, how will the process have to change? How will this failure become evident for the human operator, how will the process be returned to a consistent state, and what will be the timeframe for it? Until all of this is sorted out with precision, no matter how good the metrics, the agent under discussion cannot be considered production-safe. Correctness and reliability are properties of whole systems, not separate components—and in the case of agents, it is still at the very beginning of its system-level journey.

References

[1] M. Leucker and C. Schallhart, A brief account of runtime verification (2009), Journal of Logic and Algebraic Programming, 78(5), 293–303

[2] H. Garcia-Molina and K. Salem, Sagas (1987), ACM SIGMOD Record, 16(3), 249–259

[3] Temporal Technologies, Temporal workflow documentation (2024), https://docs.temporal.io

[4] L. Moreau et al., PROV-DM: The PROV Data Model (2013), W3C Recommendation, https://www.w3.org/TR/prov-dm/

[5] W3C PROV Working Group, PROV Overview (2013), https://www.w3.org/TR/prov-overview/

[6] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, 2nd ed. (2007), Prentice Hall

[7] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques (1992), Morgan Kaufmann

[8] V. G. Cerf and R. E. Kahn, A protocol for packet network intercommunication (1974), IEEE Transactions on Communications, 22(5), 637–648

About the Author

Nishil Pathak is an independent software engineering researcher interested in reliability of production AI agent systems. He uses principles of operating systems design, distributed systems, and runtime verification to safely and reliably operate autonomous agents. The scenarios described in this article are composite and based on patterns seen during multiple production agent deployments.

Figure Attribution: All images in this article are made by the author in Inkscape specifically for this article. No material has been captured, borrowed or lifted from anywhere else—the author had to do no permissions or rights talks. They were developed alongside the writing process and will be reused in the future as well.