Proceedings of the Paris Institute
for Advanced Study

A Science of LLM-Agent Ecosystems

Siddharth, Suri [1]

[1] *Microsoft Research*

TO CITE

Siddharth, S. (2026). A Science of LLM-Agent Ecosystems. In *Proceedings of the Paris Institute for Advanced Study* (Vol. 27). https://paris.pias.science/article/a-science-of-llm-agent-ecosystems

PUBLICATION DATE

31/08/2026

ABSTRACT

LLM agents make a new form of software delegation possible. Where conventional software follows fixed steps written out in advance, an LLM agent hands some of those steps to a language model that interprets context, forms plans, chooses tools, updates memory, generates outputs, and decides when to stop. Because such systems take consequential actions on behalf of users, their trajectories matter. Correct agent behavior depends on what agents inspect, which tools they call, what state they update, how they recover from failure, and when they stop. As agentic workflows become easier to build and deploy, many such agents will run at once, and their trajectories will increasingly intersect through shared tools, documents, memories, workflows, and the same underlying AI models. This creates an ecosystem-level problem in which local trajectory failures can become shared state, propagate across agents, and sometimes amplify through feedback loops, provenance erosion, resource contention, dense handoffs, and correlated model failures. Meeting this challenge calls for a science of agentic coordination, one that makes trajectories observable, controls how errors propagate, and stress-tests how agents interact, so it can predict when a local failure stays local and when it tips into a systemic cascade.

Contents

Introduction 3
What is an LLM agent? 5
Single-agent consequences of LLM calls 7
From single-agent failures to ecosystem propagation 9
Amplification through spread and laundering 11
Correlation through shared foundation models 12
Better models are not enough 13
Better checking is not enough either 14
Building a research program for agentic coordination 15
Acknowledgements 18

Introduction

LLM agents are becoming salient now for two reasons. The first is technical. They make a new form of software delegation possible. Conventional software follows fixed, predictable steps written out in advance; an LLM agent hands some of those steps to a language model that interprets the situation and decides what to do. One call to the model might interpret context, form a plan, choose a tool, update the agent's memory, produce output, or decide when to stop (Park et al., 2023; Schick, 2023; Yao et al., 2022). As a result, instead of specifying every step of a workflow, users can operate at a higher level of abstraction by specifying goals. The tradeoff is that users gain speed, adaptability, and the ability to delegate a wider range of work across tools and systems. In exchange, they give up some deterministic control, transparency, predictability, and ease of debugging. The second reason is socio-technical. Agentic systems are becoming easier to build through natural-language, graphical, low-code, and no-code platforms [Microsoft Copilot Studio]. LLMs also help build the agents themselves by scaffolding code, writing prompts, connecting APIs, and debugging workflows. Building agents is getting easier. Getting an ecosystem of them to work together robustly is not.

The payoff for a robust ecosystem of agents is large because much important work is multi-step, requires judgment along the way, and is spread across many tools and institutions. This is already underway in software engineering, the leading edge of agentic AI, where agents are most mature and most actively developed, and already act on shared repositories, pull requests, and test suites (Jimenez et al., 2024; Jimenez et al., 2024). That makes it both a preview of the coordination problems other domains will face and the natural first place to study agent ecosystems. Looking forward, in science, agents could help search literature, clean data, run simulations, and coordinate collaborators. In medicine, they could help patients and clinicians navigate records and guidelines, book referrals, and coordinate follow-up. In public services, they could help people navigate forms and eligibility, submit applications, and track deadlines and case updates. In all of these domains, agents are unlikely to remain solitary. As they become useful and cheap to build, many people and organizations will each run many agents, and those agents will compose into chains where one agent's output becomes another's input. The value then depends not on a single capable agent, but on many agents working together robustly through shared tools, documents, memory, and state.

A conventional LLM response can often be evaluated mainly by its final artifact. An agent is different because it acts on the user's behalf by doing consequential work for them. Agents may modify code, update tickets, change records, send

messages, spend money, trigger downstream workflows, or hand the task to another agent. An agent's trajectory includes the contexts interpreted, plans formed, tools called, results observed, state updated, and stopping decisions made. Success therefore cannot be read off a final output. Often there is none, and when there is, it does not capture the actions taken along the way. For LLM agents, the process is part of the behavior being evaluated (Kapoor et al., 2024). Furthermore, because these systems are partly nondeterministic, the same task can produce different trajectories across runs, so reliability must be judged over a distribution of runs, not a single success (Jimenez et al., 2024).

An LLM-agent ecosystem is a collection of such agents that interact through the same digital surroundings, the same tools, documents, code, memories, and workflows, and often the same underlying AI models. In such an ecosystem, one agent's mistake need not stay contained. It can become shared state, part of the environment that other agents later sense and act on. A bad summary, a stale memory, a leaked piece of context, a mislabeled record, or an unlogged recovery step can pass from tool to tool and agent to agent. The same shared resources that make these ecosystems valuable also open new paths for failure.

An ecosystem of LLM agents consists of many general-purpose, language model-driven agents pursuing underspecified goals through shared tools, documents, memories, and the same underlying AI models. Existing sciences each study it at a single scale, from the individual agent up to the whole system, and each makes assumptions that LLM-agent ecosystems violate. Game theory and mechanism design model how agents reason about one another, but only where the players, payoffs, and set of moves are well specified; an LLM agent has none of these, pursuing a goal stated in natural language through whatever action its tools allow. When agents must coordinate, multi-agent and distributed systems offer ways to coordinate and to keep working when parts fail, but their safeguards, structural checks, and agreement among redundant nodes assume failures are either malformed or independent, while an LLM's are fluent, well-formed, and correlated across agents built on the same model. At larger scale, collective intelligence and agent-based modeling study how populations produce aggregate behavior, but usually populations of simple agents with fixed rules, not general-purpose ones that reason and improvise. At the level of the whole, complex systems explain how small local events can produce outsized macro effects, but as a general theory of emergence, not an account of how a language-mediated error becomes the next agent's premise. What no field studies is all of it at once, agents that are individually complex, collectively numerous, and interacting

through open-ended language over shared information they can all change. No established science was built to reason about that combination.

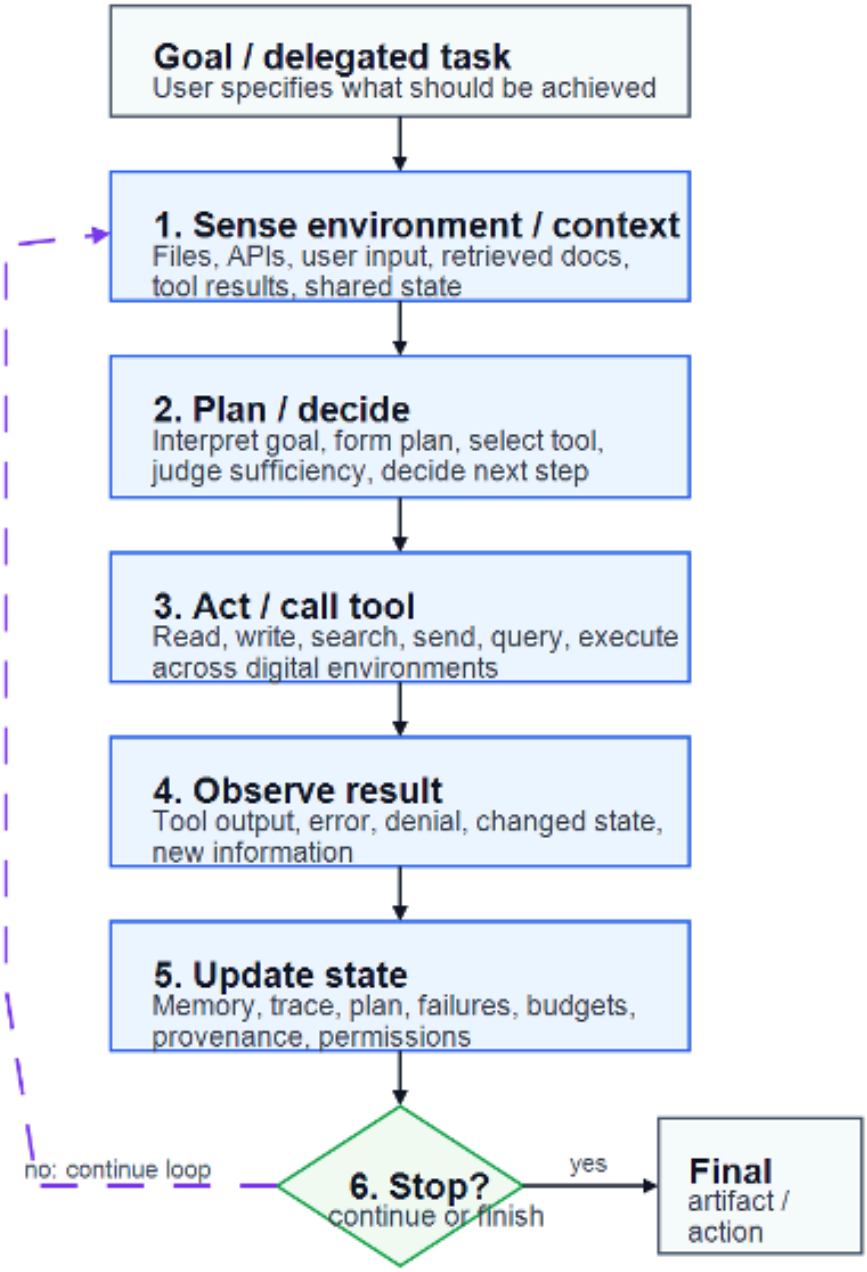
The key research question is when such failures remain local, or spread linearly or super-linearly. Feedback loops, provenance erosion (losing track of where information came from), resource contention, dense handoffs, and correlated model failures, where many agents fail the same way because they are built on the same model (Bommasani, 2021), are all mechanisms that could drive that super-linear scaling. Understanding which conditions keep failures local, and which let them cascade, is central to building agent ecosystems that are robust rather than fragile. Realizing the benefits of agent ecosystems while avoiding these cascades will require a science of agentic coordination that integrates existing work rather than starting from scratch, one that makes trajectories observable, controls how errors propagate, and stress-tests how agents interact so that local failures stay local.

What is an LLM agent?

An agent is given a goal, senses its environment or context, plans or decides what to do next, acts by calling a tool or taking an action, observes the result, updates its state, and then either stops or iterates over the loop again (Figure 1). The key aspect of this loop, which is defined in (Franklin & Graesser, 1997; Russell & Norvig, 2021; Wooldridge & Jennings, 1995), is that the agent acts over time, in pursuit of its goals with some degree of autonomy.

Figure 1: The agent senses its context, plans, acts, observes the result, and updates its state, iterating until it stops and returns a final artifact or action.

Figure 1. The LLM-agent loop



A bare LLM is not an agent. It becomes part of an LLM agent when it is placed inside the loop, so that calls to the model perform one or more of the loop's functions or supply one or more of its control decisions. These calls may interpret context, form plans, choose tools, summarize observations, update memory, generate outputs, or decide when to stop (Huang et al., 2024; Li, 2025; Park et al., 2023; Schick, 2023; Yao et al., 2022). Even when the model performs several of these functions, it does not replace the rest of the system. The agent still needs non-model scaffolding such as goals, state, tools, memory, environment

access, permissions, and stopping logic. In this paper, we focus on LLM agents that operate in digital environments, rather than on robotics or embodied agents.

The output of an LLM call can change the agent's state, and because the agent runs this loop over time, earlier steps shape later ones through that changing state. A trajectory is the sequence of contexts interpreted, plans formed, tools called, results observed, state updated, and stopping decisions made.

Single-agent consequences of LLM calls

Allowing a single agent to call an LLM results in a tradeoff. LLM calls give users speed, adaptability, a higher level of abstraction, and broader delegation across tools and systems. However, the same calls introduce nondeterminism, opacity, context sensitivity, cost, and new complexity in controlling the loop.

The clearest of these gains is a shift in the user's role from operator, executing every step of a workflow, to architect, designing, supervising, evaluating, and revising it, thereby allowing the user to work at a higher level of abstraction. Because the model can interpret context and adapt steps to the situation without every branch being specified in advance, the user can state goals and constraints instead of procedures. This makes the system more adaptable, since the model can adjust steps to the situation without every branch being written in advance. In addition, it widens what can be delegated, letting the user hand off multi-step work across tools, files, APIs, documents, and other digital systems.

This flexibility traces back to a property that sets LLM agents apart. LLMs are controlled with natural language. Their goals, the documents and tool outputs they read, the messages they receive from users, and their own system instructions all arrive as natural language, and the agent acts by interpreting it. This is what lets an agent operate in underspecified settings, where the next step cannot be fixed in advance and has to be inferred from context. But the same property is also a liability. The agent's behavior depends on how the model interprets that language, which is often ambiguous, so the same instruction can be read in more than one way.

Operating at a higher level of abstraction relocates, but does not necessarily remove, complexity. The work the user no longer does, specifying each step, is replaced by the work of controlling the loop, which means specifying goals, monitoring trajectories, recovering from failures, deciding what the agent remembers and forgets, and judging when the goal is met. Controlling the loop is hard because many of these decisions now run through the model. A single LLM call can shape what the agent attends to, how it breaks a goal into subgoals, which tools it calls, in what order it acts, how it interprets a tool result, what it stores in memory, and when it decides the task is done. Thus, validating an agent involves more than determining whether the code runs or produces the right

output (if any). Because these decisions are context-sensitive and partly nondeterministic, they cannot be judged by reading the code or inspecting a final result alone. What has to be examined instead is the trajectory itself.

That trajectory has to be judged in two ways. The first is workflow control, whether the agent used the right context, recovered from errors, preserved the right state, and stopped well. The second is accountability. Because the agent acts on the user's behalf, changing records, sending messages, and spending resources, what matters is not only whether it succeeded but whether it stayed within the goals, permissions, and constraints it was given. Those actions stand whether or not a final answer is correct, and whether or not there is one at all.

Both purposes are hard to serve because a clean-looking result can hide a messy trajectory. An LLM can often recover from invalid state, a failed tool call, stale memory, or bad context and still produce an acceptable result. The run looks like success, but it is recovered success rather than clean success, and the difference is invisible if only the end state is checked. Telling the two apart requires logging the trajectory in enough detail to inspect its contexts, model calls, tool calls, tool results, state updates, guardrail hits, retries, approvals, failures, and stopping decisions. This logged record is the trace, the concrete artifact that trajectory evaluation works on. Such evaluation is already emerging for individual agents (Kapoor et al., 2024; Liu, 2024; Ma et al., 2024.

Because some of an LLM agent's steps run through an LLM, its behavior is probabilistic and context-sensitive, more like a machine-learning system than a conventional program. The same task, run again or worded slightly differently, can take a different path or reach a different result, so a single run, even a clean one, is only evidence that a good outcome can be achieved, not that it can be achieved reliably. Thorough testing means characterizing the distribution of trajectories and results across repeated runs, varied contexts, and model versions (Jimenez et al., 2024). Beyond whether a single run produced an acceptable result and a clean trajectory, what matters is whether those properties hold across the whole distribution (Kapoor et al., 2024). The distributional view has limits of its own. For rare, high-stakes, or irreversible actions, the distribution is hard to estimate, and the average is little comfort, because a single failure is not offset by many good runs.

This variability is a reason not to route the whole loop through the model. Some of an agent's work is brittle mechanics that must come out the same way every time, such as discovering the right files, searching them, checking a tool's preconditions, validating state, enforcing permissions, and halting on step or budget limits. Leaving these to the model means having the model figure them out again on every run, with the variability that brings. They are better handled by deterministic scaffolding, so they hold regardless of how the model behaves. The model earns its place elsewhere, where interpretation, relevance judgment,

adaptation, and synthesis are what the task needs. The design problem for an LLM agent is where to draw that line between steps that follow fixed rules and steps better left to a language-model call, and drawing it well takes experience and expertise.

An agent is a partially observable, nondeterministic control system, one you cannot fully see into or fully predict, with memory, failure recovery, stopping rules, guardrails, and behavior that has to be judged over a distribution of runs. Wiring a language model to some tools in a loop can still demo well, because a capable model handles the easy cases, but it does not by itself make a reliable agent. The failures show up in the harder cases, in the trajectories and the tails rather than the head of the distribution. The fundamental concern is not that people cannot build agents but that they can build them so easily, before they have reckoned with the control problems they have taken on. Furthermore, few of these agents will run alone. What happens when many of their trajectories meet, through shared state and shared environments, is where the harder problem begins.

From single-agent failures to ecosystem propagation

We have argued that an LLM agent must be judged by the distribution of its trajectories. The ecosystem problem begins when those trajectories stop being isolated, when agents read and write the same documents, tickets, code, databases, and memories. This does not require badly built agents; even careful ones produce local failures, and shallow recipes only make them more common. The complexity explosion is not merely that there will be many agents. It is that many nondeterministic, language-mediated agents will interact via a complex network of trajectories intersecting through shared tools, memories, documents, workflows, and foundation models. In this way, behavior that is tolerable in one trajectory can produce a worse outcome once many cross, a micro-to-macro effect long studied in complex systems (Anderson, 1972; Schelling et al., 1978).

A failure in one trajectory leaves an error in shared state, and when a later agent reads it, it may act on that error, making it an incorrect premise. The premise need not be a false claim. It need only be an input that looks sound but is not, and it can be unsound in two ways. First, its substance may be defective. It may be wrong, an incorrect fact or a bad data value. It may be stale, correct once and no longer. It may be incomplete, missing a part or detail that changes what it means. Or it may be out of scope, sound for one task and misleading in another. The second way a premise can be unsound is when its warning signs are stripped, so it reads as sounder than it is. It may be overstated, its uncertainty gone so a hedge reads as settled, or unsourced, its provenance and caveats fallen

away. Not every failure plants a claim at all; some remove a safeguard, as a disabled test does. What unifies these failures is not falsity but that a later agent acts on an unsound premise as if it were sound, and that crossing from one agent's output into another's input is what turns a local failure into an ecosystem one.

Erroneous data, whether in prose or in a structured format, is difficult for an agent to detect. Its wrongness resides in its meaning, not its shape, so you cannot catch it by inspecting the message itself. An agent would have to compare it against something outside the message. That external validation can come from three places:

- Reason: derive the claim from axioms by truth-preserving steps, or check it against a precise criterion. This is a formal specification.
- The world: observe or measure the claim, or consult a source that did. This is an external oracle.
- Other minds: pool independent judgments and trust the consensus. This is aggregation.

Unfortunately, in an open ecosystem each of these three methods is usually out of reach. A formal specification rarely exists for open-ended language, and the types or schemas that do exist pin down form, not meaning. An external oracle, a test, a tool, or ground-truth data, is missing for most open tasks, and the domains that have one, like code with its tests, are just where agents are already most reliable. Aggregation fails when the judgments are not independent, which is the rule once agents share models, prompts, and training, or when one model checks its own work (Huang et al., 2023). So a wrong premise travels as an ordinary-looking claim.

We illustrate these arguments with a concrete example from the domain of software engineering, which currently has the most mature agent ecosystem. Coding agents already resolve issues and run the project's automated tests, the checks meant to confirm the code works, and they read each other's build results, the green-or-red summary of whether those tests passed, as evidence that a change is sound. Suppose an agent faces a test that fails intermittently for no clear reason and, judging it flaky, disables the test to force the build green. The disabled test is persistent state, and the green check is the artifact every later agent reads. To them, that part of the code looks tested, so they build on it, restructure the code around it, and approve changes that touch it, all on the belief that a passing build means working code. The green build is the false premise that spreads widely and is widely believed, and the disabled test is exactly the external oracle that would have caught it, switched off. Meanwhile, the code the

test used to guard continues to be edited and drifts untested, accumulating real defects that nothing is watching for. The mistake is not only the one silenced test; it is that a whole part of the system now rests on a green signal that no longer means what everyone thought it meant. When the system finally breaks, the failure surfaces far from the agent that hid it, in code that relied on a safety net that was no longer there. One might ask why a team of human engineers does not fail this way. Sometimes it does, an intern silences a flaky test, and the same story can unfold. But the ecosystem version is worse because agents act at a volume and speed that bury one silenced test under a flood of other plausible changes, so the warning signal is drowned rather than caught.

Amplification through spread and laundering

When one agent's error becomes another's input, it spreads from agent to agent, forming a cascade whose size is the number of agents that act on the erroneous information (Watts, 2002). The cascade grows multiplicatively when each agent that inherits the error hands it to more than one other, so its reach depends on the branching structure of the network of agent interaction.

Spread and laundering amplify different quantities. Spread widens how far an error reaches; laundering boosts how true it looks. A premise can arrive with its warning signs already stripped, but interaction amplifies the same effect as the error travels. Repetition gives it credibility, since each pass tends to polish the artifact, turning a hedged finding into a clean summary until a claim that survived a few retellings reads as settled fact. Endorsement adds credence from a different direction. A single approval, sign-off, or citation gets treated as if it meant the claim was actually checked, so later agents lean on that stamp instead of re-examining the underlying evidence. The two are distinct, but compounding, and each can make the other worse; whether that compounding grows harm super-linearly is an open empirical question, its mechanism clear but its magnitude not.

The costs of a cascade can be amplified through feedback loops. Every agent that acts on the error spends real tool calls, queue slots, rate limits, and budget on work a correct premise would not have required. When that work fails or collides, agents retry, and retries against already-strained services can drive a feedback loop, where the response to failure becomes a new source of load (Ulrich, 2016). Human review is a relatively slow and scarce resource, unable to keep up as the number of things needing review climbs. There are remedies to the symptom of increased load, such as backoff and graceful degradation (Ulrich,

2016), but solving the cause, that the premise driving the load was wrong, is the deeper problem.

Correlation through shared foundation models

A shared foundation model makes agents' errors correlated rather than independent, and it does so in two ways. When the agents are operating, those that share a model, prompt, tools, or evaluation criteria inherit its failure modes directly (Bommasani, 2021). An input that fools one agent tends to fool similarly built agents. Thus, the spreading error lands on a population unlikely to catch it, and the cascade stops being merely large and becomes consensus. Correlation also enters when the agents are built, because language models increasingly write the agent code. Agents generated from the same models and the same popular templates inherit the same blind spots, such as a missing check, even when they later run on different models. One consequence of that correlation is that errors need not travel to spread. Where propagation carries one agent's mistake into another's input, correlation has many agents reach the same mistake independently.

The deeper consequence is that correlation breaks the two most common safeguards, redundancy and checking. Adding agents, whether they redo the work and vote or check one another's output, buys reliability only when their errors are independent. That much is not new; even programs written independently still failed on the same inputs, especially the corner cases, long before LLMs (Knight & Leveson, 1986), and a shared decision rule correlates outcomes across everyone who adopts it (Kleinberg & Raghavan, 2021). What is new is not the correlation but the safeguard it removes. A semantic error, recall, carries no structural check. The checks that could catch one are the same three, a formal specification, an external oracle, or the aggregation of independent judgments, and with the first two usually absent, independent judgment is the last line, and that is what a shared foundation model removes. A second reviewer built on the same model as the first misses what the first missed, and a checker that shares the generator's training waves its errors through. An ecosystem can then run the full apparatus of redundancy and review while adding apparent assurance but little real scrutiny, so a single shared weakness becomes an ecosystem-wide fault line rather than a local bug.

Seen one agent at a time, each of these failures can look tolerable. A single disabled test, a single confident summary, a single reviewer that misses what the last one missed will pass its local check and would not, on its own, bring a system down. The trouble is that the same failures rarely stay one agent at a time. A local error can persist as shared state, propagate through the environments agents read and write, amplify as more agents act on it and believe it, and correlate across agents built and run on the same foundation model. Each

mechanism carries the failure further from the trajectory that produced it, until the macro-level outcome can be qualitatively worse than the sum of the individual defects, in a way no single-agent inspection would have revealed. That is what makes an LLM-agent ecosystem more than a collection of agents. It is a system with failure modes of its own, and the question that matters is which of these mechanisms keep a failure local and which let it become systemic.

Better models are not enough

The key technical challenge in constructing a robust ecosystem of agents is that an agent cannot easily tell whether what it reads from shared state, or what it writes to shared state, is actually correct. If that validation were feasible, the propagation, laundering, and correlation problems would be solved, because each premise could be checked at the handoff, preventing wrong ones from spreading. One might hope that a more capable model will eventually solve this validation problem. Better models could certainly help, by lowering the base rate of error, calibrating their uncertainty, and, when asked, carrying provenance and caveats forward more faithfully. But each of these improves the output at the moment it is generated; none guarantees it survives the handoff to the next agent, which is where propagation, laundering, and correlation do their work.

A better model, given current architectures, is not a new way to certify truth. A language model is a prior over plausible text, and training it on more data and more compute sharpens that prior, lowering the rate at which it emits a false claim. A lower error rate is not a certificate. The model still has no specification to check against, no contact with the world to measure against, and when it grades its own output or a peer's, it is not an independent judge but the same prior consulted twice, aggregation with its independence removed. As with redundancy across correlated agents, a check that shares the generator's blind spots waves the generator's errors through. Validating truth is a grounding problem rather than a capability problem, and grounding is the contact with something outside the model that scaling the model does not add: a specification to check against, the world or a test to measure against, or ground-truth data to consult. Importantly, better models raise the typical trajectory but do not remove the tail, and in an ecosystem, tail risk matters because a single rare failure that reaches shared state or a high-consequence workflow can seed a cascade that many clean runs do not undo.

Furthermore, reliability is a property of the whole agent system, not of the model at its center. Whether a local failure turns systemic depends on four things: how often an agent errs, how far an error spreads, how much it launders as it travels, and how correlated the agents are; how much damage it then does depends on the cost of the actions it reaches. A better model mitigates only the first. Spread

depends on how frequently agents hand work to one another through shared memory and state, which the model does not set. Laundering depends on whether provenance and caveats survive each handoff, which the system's design governs more than the model's capability. Correlation rises, not falls, as more agents come to depend on the same capable model. And cost does not fall but tends to rise, since the more capable an agent appears, the more authority it is given, broader permissions, and more irreversible actions within reach. These are structural choices made around the model, not capabilities inside it, so a stronger model dropped into a system still produces trajectories that propagate into shared state, amplify as other agents act, and correlate with agents built the same way. A lower error rate multiplied by wider reach, more laundering, tighter correlation, and higher stakes need not be a lower risk.

Better models can even worsen the risk they are supposed to mitigate. The patterns they learn tend to raise capability while saying little about monitoring, recovery, provenance, memory scope, or stress testing, so more capable agents can ship with the same thin instrumentation as before. Those patterns also circulate, as agent code and practices written by models are published, become training data, and are reproduced by later models, so a weak way of building agents can entrench rather than wash out. And the more agents the same models write from the same recipes, the more their designs and failure modes correlate at build time, which is the very correlation that removes independent judgment. Better models therefore tend to make agent ecosystems more capable and more widespread, raising the need for observability, propagation control, and stress testing rather than removing it.

There is one genuine way better models improve validation, and it proves the point. A model can be wired to call a test, query ground truth, carry provenance forward, or escalate when it is unsure, and each of these does supply the grounding the model lacks. But the grounding comes from the test, the data source, the provenance channel, and the escalation path, that is, from instrumentation the system provides, not from the model's raw capability. The claim that better models will solve the ecosystem problem becomes the claim that a better-instrumented ecosystem will, which is not a rebuttal to the argument of this paper but a statement of its program.

Better checking is not enough either

Suppose the ecosystem designer instruments every agent to run a second, independent verifier, a simpler model from a different family, that checks the information passing into and out of the primary at each handoff. This would help because a different model family uses different training data, architecture, and therefore has different failure modes. Thus, it does not share all of the primary

model's blind spots, so an error idiosyncratic to one family has a better chance of being caught by the other. It is also exactly the kind of propagation control this paper recommends, an independent judgment inserted where shared models had removed it.

This approach mitigates rather than solves systemic risk, for four reasons. First, independence lowers correlated error, but does not remove it entirely. An independent verifier is still a judgment, not grounding. For any claim whose truth depends on an external fact, the verifier is guessing too, with a differently shaped prior. Two models that disagree flag uncertainty; two that agree do not certify truth, since both can be confidently and agreeingly wrong about a fact neither can consult. Second, the independence is partial and shrinking. Different families train on overlapping internet-scale corpora, increasingly including each other's output, toward similar objectives. Independently written programs still failed on the same inputs long before LLMs, because the hard cases are hard for everyone (Knight & Leveson, 1986), and cross-family verifiers will likewise miss in correlated fashion on exactly the subtle cases that matter. Third, the verifier is itself a fallible, nondeterministic agent in the ecosystem. When it wrongly approves, it adds an endorsement that launders the error further; when it wrongly rejects, it adds load, and at ecosystem scale a checker tuned to catch subtle errors floods the scarce human review it leans on, while one tuned to stay quiet misses them. Fourth, verification reaches only part of the risk. Of the four factors that set whether a local failure turns systemic, how often an agent errs, how far the error spreads, how much it launders, and how correlated the agents are, cross-family checking lowers the error rate somewhat and correlation partially, does nothing about spread, and can worsen laundering rather than reduce it. Nor does it lower the cost of the actions an error reaches.

The verifier, then, is a real improvement that leaves the core problem standing. That it is one of the better instruments in this paper's own program and still cannot close the gap is the point. Grounding is what is missing, and no amount of independent opinion manufactures it. Whether cross-family verification shifts the threshold between local and systemic failure at all is an empirical question, one that would require measuring the residual correlation between families, the catch rate on laundered versus raw errors, and the load the checks impose, which is the measurement the next section calls for.

Building a research program for agentic coordination

The central question is: under what conditions does a failure stay local, scale linearly with the number of agents, or tip into a super-linear cascade? Answering it is part engineering, part science, and both are necessary. The engineering builds

the instrumentation to observe what agents do, to keep one agent's output from silently becoming another's incorrect premise, and to bound what an erring agent can reach. The science uses that instrumentation to measure whether an ecosystem is safe and to predict when it will not be. Four measurable factors determine whether a local failure turns systemic: how often an agent errs, how far an error spreads, how much it launders, and how correlated the agents are. A fifth quantity, the cost of the actions an erring agent can reach, sets how much damage a systemic failure does. Defining these, measuring them, and learning to predict the threshold is the science in the program.

Each of these factors is a property of a trajectory, and a trajectory is invisible by default, so observability comes first. The engineering is logging the whole agent loop, the model calls, the tool calls and their results, the state updates, the approvals, the retries, the failures and recovery paths, and the stopping decision, kept as traces inspectable enough for debugging, accountability, and monitoring across the ecosystem. The science is in gleaning the insights from those traces. A trace should separate a clean success from one reached only after recovering from an error, since the two say very different things about reliability (Holmstrom, 1979). And because a language-mediated agent is nondeterministic, no single run certifies it, and reliability becomes a distributional property. One logged success is weak evidence (Kapoor et al., 2024), and what matters is the distribution of trajectories and outcomes across repeated runs, varied contexts, model versions, prompts, and memory states, the distribution from which the base rate and the other factors are estimated (Jimenez et al., 2024).

The harder task is keeping what happened in one trajectory from silently becoming an incorrect premise for the next agent. Memory and retrieval can be scoped by task, user, role, domain, and time, so a note written for one job does not resurface as authority in an unrelated one, which bounds spread. When information is summarized, retrieved, or handed off, its provenance, uncertainty, and caveats should travel with it rather than being laundered away, so a hedged finding does not arrive downstream as settled fact (Sankararaman et al., 2024). Artifacts should not become action-guiding context without validation, and an agent that meets contaminated or out-of-scope input can fail fast and escalate rather than quietly repair it and press on. In the running example, treating a newly green build as a claim to be checked rather than as ground truth would have caught the disabled test before anything was built on it.

There are several direct controls that bound the cost of an erring agent, such as permissions that scope what an agent may touch, budgets that cap what it may spend, guardrails that block unacceptable actions outright, and a preference for reversible actions, so a wrong premise cannot cause irreversible harm before anything catches it. Human judgment is the scarcest control of all, so it should be reserved for the actions whose stakes justify it, those that are high-consequence,

irreversible, ambiguous, or privacy-sensitive, and conserved everywhere else. A different kind of cost arises where many agents share a tool or an API, where the danger is aggregate load rather than any single action; there the controls built for cascading failure apply almost directly, circuit breakers, backoff, and rate limits that keep one agent's overload from pulling down shared infrastructure (Ulrich, 2016).

Each of these controls protects a single agent or shared service in isolation, but the failures that matter emerge in the interactions between agents, which testing one agent at a time cannot reveal. Ecosystem stress tests should exercise multi-agent handoffs, feedback loops, resource contention, shared-memory contamination, laundering, and correlated failure, asking of each which local failures stay local and which turn systemic. They are also where the empirical unknowns get measured, the residual correlation between agents built on different model families, the catch rate on laundered versus raw errors, and the load that added checking imposes. They should span both ordinary conditions, such as workload growth as more agents deploy, and adversarial or distribution-shifted ones, since an ecosystem that holds under normal load can still tip under a hostile input.

One exposure in particular that slips past per-agent testing is the correlation due to shared foundation models, an ecosystem-level factor no single agent can reveal. Measuring and governing it means tracking which agents share the same model, prompts, scaffolds, tools, retrieval systems, training data, and evaluation criteria, because that shared inheritance is what quietly turns many independent-looking agents into correlated ones (Bommasani, 2021). Redundancy and cross-checking, the usual ways to buy reliability, protect much less when the redundant agents carry correlated failure modes (Kleinberg & Raghavan, 2021), so ecosystem monitoring should watch for them across agents, not only for defects inside one. Without that view, an ecosystem could pass every local check while a single shared weakness moved through all of it at once.

None of this is an argument against building agent ecosystems. Making trajectories observable, controlling propagation, spending human judgment where it counts, stress-testing interactions, and governing shared foundation models are what would make delegated, model-mediated action legible, auditable, and robust enough to trust with consequential work. The reason to study these systems is not that they are bound to fail, but that they could become an important foundation for coordinated action across otherwise fragmented digital systems, in science, the economy, public services, and everyday life. Getting that coordination right is

what turns a collection of capable but fragile agents into an ecosystem worth having.

Acknowledgements

This work was carried out while I was a visiting fellow at the Paris Institute for Advanced Study (Institut d'études avancées de Paris). I thank Mohammed Alsobay and Scott Counts for many useful comments. I used Claude Opus 4.8 as a research assistant for this paper. I am responsible for all content.

Bibliography

Anderson, P. W. (1972). More Is Different. *Science*, *177*(4047), 393–396. https://doi.org/10.1126/science.177.4047.393

Bommasani, R. (2021). *On the Opportunities and Risks of Foundation Models* (Vol. 2108). https://arxiv.org/abs/2108.07258

Franklin, S., & Graesser, A. (1997). Is It an Agent, or Just a Program? A Taxonomy for Autonomous Agents. In *Intelligent Agents III (ATAL 1996), LNCS 1193* (pp. 21–35). https://doi.org/10.1007/BFb0013570

Holmstrom, B. (1979). Moral Hazard and Observability. *The Bell Journal of Economics*, *10*(1), 74–91. https://doi.org/10.2307/3003320

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2024). Large Language Models Cannot Self-Correct Reasoning Yet. *ICLR*, *2310*(01798). https://arxiv.org/abs/2310.01798

Huang, X. (2024). *Understanding the Planning of LLM Agents: A Survey* (Vol. 2402). https://arxiv.org/abs/2402.02716

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *ICLR*, *2310*(06770). https://arxiv.org/abs/2310.06770

Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., & Narayanan, A. (2024). *AI Agents That Matter* (Vol. 2407). https://arxiv.org/abs/2407.01502

Kleinberg, J., & Raghavan, M. (2021). Algorithmic Monoculture and Social Welfare. *Proceedings of the National Academy of Sciences*, *118*(22). https://doi.org/10.1073/pnas.2018340118

Knight, J. C., & Leveson, N. G. (1986). An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, *SE-12(1):96-109*. https://doi.org/10.1109/TSE.1986.6312924

Li, X. (2025). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use. *Planning (Including RAG), and Feedback Learning. Proceedings of COLING 2025*, 9760–9779. https://aclanthology.org/2025.coling-main.652/

Liu, X. (2024). AgentBench: Evaluating LLMs as Agents. *ICLR*, *2308*(03688). https://arxiv.org/abs/2308.03688

Ma, C. (2024). AgentBoard: An Analytical Evaluation Board of Multi-turn LLM Agents. *NeurIPS*, *2401*(13178). https://arxiv.org/abs/2401.13178

Microsoft. (n.d.). *Copilot Studio product page*. https://www.microsoft.com/en-us/microsoft-365-copilot/microsoft-copilot-studio/

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). *Generative Agents: Interactive Simulacra of Human Behavior* (Vol. 2304). https://arxiv.org/abs/2304.03442

Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Sankararaman, H., Yasin, M. N., Sorensen, T., Bari, A., & Stolcke, A. (2024). Provenance: A Light-weight Fact-checker for Retrieval Augmented LLM Generation Output. *EMNLP*. https://arxiv.org/abs/2411.01022

Schelling, T. C. M., Norton, M. W. W., & Company. (1978).

Schick, T. (2023). *Toolformer: Language Models Can Teach Themselves to Use Tools* (Vol. 2302). https://arxiv.org/abs/2302.04761

Ulrich, M. (n.d.). Addressing Cascading Failures. In B. Beyer, C. Jones, J. Petoff, & N. R. Murphy (Eds.), *Site Reliability Engineering: How Google Runs Production Systems*. https://sre.google/sre-book/addressing-cascading-failures/

Watts, D. J. (2002). A Simple Model of Global Cascades on Random Networks. *Proceedings of the National Academy of Sciences*, *99*(9), 5766–5771. https://doi.org/10.1073/pnas.082090499

Wooldridge, M., & Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, *10*(2), 115–152. https://doi.org/10.1017/S0269888900008122

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *NeurIPS*, *2405*(15793). https://arxiv.org/abs/2405.15793

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*, *2210*(03629). https://arxiv.org/abs/2210.03629

Yao, S., Shinn, N., Razavi, P., & Narasimhan, K. tau-bench. (2024). *A Benchmark for Tool-Agent-User Interaction in Real-World Domains* (Vol. 2406). https://arxiv.org/abs/2406.12045