Agents, Frameworks, Feedback, and A2A

Where I Think Multi-Agent Systems Are Going, and What That Demands of a Platform

Pavan Kumar TV

***Abstract*—The hard problems in multi-agent systems are not model problems: they are organisational problems in an engineering medium, and organisations have been failing at them for long enough to have left notes. I set out the vocabulary that makes an agent a distinct category, the platform obligations that follow, the feedback write path where agents actually fail, the A2A extension I published to describe that path, and the three ideas I think matter most over the next few years: hierarchy and boundaries, careers made legible, and shared context that distributes at the speed of a write rather than the speed of a career. It argues throughout that teaching has to run in both directions: a learnable is a surface people and agents share, not a port pointing one way, and the extension needs one revision before it can carry the return leg. The closing sections are the ones to read adversarially: what I argued against, what I hold loosely with its falsification condition stated, and the questions I cannot yet answer. No effort, cost or schedule figures appear anywhere, deliberately.**

I. The Vocabulary Problem

One word doing five jobs, and the test that separates them

Gartner expects more than 40% of agentic AI projects to be cancelled by the end of 2027.[1] Read the reasons before the number: escalating costs, unclear business value, inadequate risk controls. Not one of those is a model limitation. Not one is fixed by a better benchmark score.

They are the reasons projects have always been cancelled, and they show up here because the word "agent" is doing five jobs at once. That is expensive in a way that is easy to miss. It is not a labelling argument. Each of the things below has a different failure mode, a different testing strategy, and a different answer to "who is accountable when this is wrong." Calling all five the same thing means picking the wrong one of each.

An engine is a deterministic transformation. Rules in, result out. No identity, no memory, and that is the feature: the same input a year from now produces the same output.
A library is an engine linked into your process. You own the lifecycle; it forgets you the instant it returns.
An API is an engine behind a boundary with somebody else's team on the other side. Its defining property is that the caller needs to know nothing about it.
A model is a probabilistic engine. Same input, roughly the same output, no memory of the last call. Non-determinism is not personality.
A workflow is a graph of those. Orchestration in the edges, every node dumb.

[1]Reported in *Forbes*, July 2026.

TABLE I
THE SIX CATEGORIES, BY THE PROPERTIES THAT ACTUALLY DIFFER

	engine	api	workflow	agent
holds context	caller	caller	the graph	itself
state after a call	none	none	discarded	permanent
100th call better	no	no	no	yes
can it refuse	no	error	no	yes, with a reason
can it delegate	no	no	pre-wired	decides who owns it
who answers for it	author	owner	author	itself, then its owner

An agent is the only thing on the list that is supposed to be different tomorrow because of what happened today.

That last line is the dividing line, and it beats any capability argument. Everything above it is designed to be unchanged by use. An agent is designed to be changed by use.

A. The test

Run the same interaction one hundred times. Does the hundredth go better than the first?

If no, it is a service. Services are supposed to be stateless, a good job, honestly named. If yes, everything in the rest of this note applies and none of it is optional.

B. Three things that invert

Take the definition seriously and three certainties turn upside down.

Context flips direction. An API is engineered so the caller carries all the knowledge. An agent is engineered to accumulate knowledge of the caller. Wrap a model in a REST route and call it an agent and you get neither: it cannot hold state because the route is stateless, and it cannot be trusted because the model is not deterministic.

Version means the opposite thing. An API version is a promise not to change. An agent version is a record of having changed. Version an agent the way you version an API and you get one number that never moves while the thing behind it drifts every week.

Idempotency becomes a defect. We spent years ensuring the same request twice produces the same result. Ask a colleague the same question twice and the right answer is *"you asked me this on Tuesday, here is what we decided."* An agent that answers identically the second time has told you it was not listening the first time.

That third inversion breaks the testing strategy, which is worth sitting with. An engine fails loudly and identically, which

is exactly why unit tests work. An agent fails the way a new hire fails: plausibly, once, in a way that looked fine at the time and turns out to have been wrong six weeks later. You do not catch that with assertions. You catch it with read-back, review, escalation, and a record of who decided what.

II. Statelessness as an Inherited Virtue

Every rule we learned is a rule about forgetting

Look at what the last twenty years of distributed-systems practice taught, listed plainly:

- REST: no session on the server.
- Twelve-factor: processes are disposable, share nothing.
- Horizontal scaling: any instance serves any request, only true if no instance remembers anything.
- Idempotency: same call twice, same result.
- Pure functions, immutable infrastructure, cattle not pets.

Every one of those is a rule about forgetting. They were right. State was what broke at scale, and a system that remembers nothing can be restarted, replicated and reasoned about at three in the morning.

Then a model went into the middle of that stack and the result was called an agent. Now the most valuable property of the system is the exact thing the stack was built to eliminate.

This is why so many agent frameworks feel thin. They model the agent as a while-loop around a model call. The loop runs, the task completes, the process exits, and everything learned inside it dies at exit because the design has nowhere to put it. A bigger context window does not fix it. A context window is a longer breath; it is not a memory.

The research side reached the same place independently. The ICLR 2026 workshop on agent memory landed on memory as the central unsolved infrastructure problem in agentic AI, with staleness, not storage, as the hard part, and separate teams converging on the same three-layer split of global, role-scoped and private memory. Everyone is discovering the same hole.

A. What the hole looks like from inside one portfolio

Four products here each hit the wall and each patched it the same way without consulting the others: a table that remembers what a person said, consulted before asking again. They were built at different times by different teams. Each accreted its own guard list after its own incident; one exists because a notification vendor's domain once auto-created itself as a customer and misattributed shipments.

Independent teams converging on the same table is not a coincidence and not a coordination failure. It is the shape of the hole. Everything that follows is an argument about what belongs in it.

III. What a Platform Owes an Agent

Engines, agents, loops, and the three obligations underneath

The architecture question people ask is "what boxes do I draw." I do not think there is an agent architecture worth drawing. There is a division of labour, and it is closer to an org design than a block diagram.

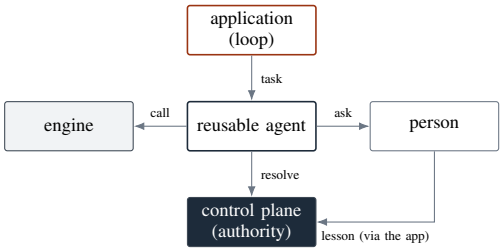

Fig. 1. The division of labour. The agent asks; the application decides whether to teach; the authority records. The agent never writes its own memory (Section V is why.

Engines do deterministic work and issue a receipt. Document extraction, rate evaluation, party lookup. Either tenant-agnostic compute with a tenant-attributed receipt, or a tenant-keyed answer, and that distinction has to be written down, because whether a replay is reproducible depends on it.

Agents hold judgment and memory. They ask a human when they cannot decide. They are reusable across applications.

Loops are the application-specific part: exception handling owned by exactly one product. A loop that served every application would be an agent, not a loop, and that distinction is worth enforcing at task-submit time rather than by convention. An envelope naming a different application than the loop's owner should be refused before the skill lookup runs, deliberately before, so that a misrouted task can neither enumerate skills by the difference between a 404 and a 403 nor consume an idempotency key on its way to being rejected.

A. The three obligations

Three platform mechanisms carry the weight, and none of them are interesting to look at.

A contract package. One shared definition of the task envelope, the task lifecycle, the artifact and the receipt, pinned by tag rather than tracked on a branch, and resolved at image-build time from that tag. Anything that resolves a dependency from a branch tip has made its builds a function of what somebody else pushed this morning.

Content-addressed definitions. What an agent knows is a bundle identified by the hash of its contents. Activating a bundle for a tenant is an audited row. Keep publishing separate from activating and rollback is free.

Task pinning. A task carries the definition-bundle hash it started with. A task that began under one version of an agent's knowledge finishes under that version even if a newer one goes live mid-flight.

That third one is the least glamorous requirement in agent design and the first one anybody wishes they had. Without it, "the agent learns" and "the agent changes unpredictably underneath running work" describe the same behaviour and neither can be debugged.

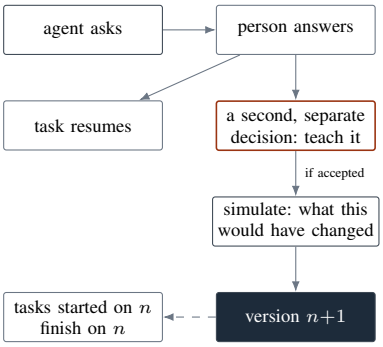

Fig. 2. Answering is not teaching. The answer unblocks one task; turning it into a rule is a separate, explicit decision, nothing is learned that was not simulated first, and nothing already running changes version underneath itself.

B. Tenancy, stated honestly

Tenancy is where this is weakest and it would be dishonest to skip it. The contract requires a tenant on every envelope, forbids delegation across tenants, and keys every runtime table on it. The mapping from an application's own identifier to a platform tenant belongs in the authority and should be written only by an operator command, never by an agent.

Underneath that, the products differ sharply: one enforces tenancy at the database with row-level security, others enforce it in application code by hand, and one is a document store where an unscoped read fails silently rather than loudly. Wherever the database does not enforce it, a hand-off's tenant becomes the sole authority for downstream writes and nothing catches a wrong one. That is a risk to carry deliberately and name in the design, not a solved problem to describe optimistically.

IV. Feedback: The Write Path

"Saved" is not a result

Everyone doing agentic work hits this failure mode. You ask for something. It reports success. You have no way to check the report except by doing the work yourself.

People solved this a long time ago with show-back. Ask a colleague to draft an email and they show you the draft, not the words "email drafted." You fix one thing. Next time it is already applied and you never say it twice.

Three mechanisms hide inside that ordinary exchange:

Template 1 — the three parts of an ordinary correction

```
read-back    here is what I understood, in the
             shape of the actual output
correction   a channel for you to fix it, cheaply,
             in the moment
durability   a write path that makes the correction
             survive the session
```

Most implementations have the first, sort of. Almost none have the third. If you build nothing else from this note, build the third. An agent that reports the success of its own write, instead of showing you what it now makes of your input, has told you nothing.

A. What happens to the answer

When an agent stops and asks a human, the answer is not merely an unblocking token. It is evidence about the world, held by the one party who had it, delivered at the exact moment it mattered. There are three things done with it:

Template 2 — the three options in the wild

```
discard      the same question tomorrow, forever
redeploy     write it into a prompt or config,
             ship it Thursday
self-modify  fast, and nobody can say what the
             agent knew when it decided
```

The first is what most systems do. The second is what serious teams do, and it means the knowledge reaches production days after the person who held it walked past the screen. The third is what the industry currently calls "agents that learn."

An agent writing to its own memory with no author, no version and no undo is not learning. It is an agent editing its own source in production while nobody watches.

This is the sentence the whole feedback design exists to avoid.

B. One layer, not one per product

The position I hold: no product keeps its own learn-on-confirm store. Each one's table becomes seed data for a shared layer and then stops being read. Permanent dual writes with no consumer are a liability nobody ever cleans up, and rollback during a transition is better served by a flag than by a reconciliation.

What a unified layer has to carry, so that nothing is quietly lost, is specific, and enumerating it is most of the design work:

- every product's guard list, each of which exists because of a distinct incident, merged into one audited rule set;
- negative signals: "this is not ours" is a fact worth keeping;
- a mandatory counterfactual preview before teaching: show what this lesson would have done to the last thirty days before it is allowed to affect the next thirty;
- a signal vocabulary that is the *union* of what the products use, not the intersection.

One behaviour I would subtract rather than move: auto-creating a party record on first sight of a corporate domain. The clerk proposes a new party; the application decides. Convenience there is exactly how a notification vendor becomes a customer.

V. A2A and the Learnables Extension

The protocol specifies the moment a human is asked, and stops there

A2A does a lot of things well. An agent publishes a card saying what it can do. When it hits something it cannot decide, it moves the task to `input-required`, a human answers, and the task resumes. That hand-off is specified carefully and it works.

What is not specified is the moment after, which is exactly when the only person who knew the answer walks away. So I

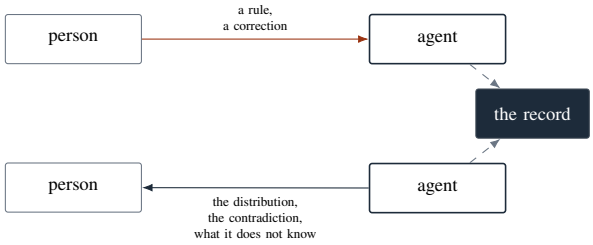

Fig. 3. Both directions are the same machinery: a proposal carrying evidence, a named party who decides, one record. Designs routinely wire the upper path and leave the lower one out.

wrote an extension for it and published it as a draft under a URI I control.[2]

A. Three roles

The agent proposes, the client decides, the authority records.

The whole extension is this sentence with the edge cases written down.

The **agent** publishes what it can be taught, in the same place it publishes what it can do: which collections exist, what shape an answer must take, how it decides today in prose, and whether that decision is made by rules or by a model. That last field is required because it changes what a client may honestly promise. Under rules, a taught mark takes effect deterministically and the interface may say "this will match." Under a model it is one example among many. Same button, entirely different sentence beside it.

The **client** renders those fields blind. It reads the label off the card and draws the input. A client encountering a field kind it has never heard of must render it as a text box; never drop it, never fail. This is the load-bearing rule of the whole extension: a client rendering blind needs no change when an agent that reads spreadsheets asks to be taught a column mapping instead of a text pattern, whereas a client that hard-codes one agent's fields must be edited and released.

The **authority** absorbs the lesson. It validates nothing itself; it hands the value to the learnable and asks. It publishes a new immutable version. It records who taught it and why.

B. Teaching runs both ways, or there is no point

A learnable is a shared surface, not a one-way port.

If teaching only ever flows from person to agent, the person's attention is a pure cost and the agent is an apprentice forever.

Everything above describes a human teaching an agent. Read it back and the asymmetry is embarrassing. A person spends months correcting the same thing and receives, in exchange, an agent that stops asking. That is a reduction in annoyance. It is not a return.

The return has to be that the agent teaches the person. It is the only party in the arrangement that has seen the whole distribution. A person sees the exceptions that reached them. The agent has seen every message, every correction, and every case that quietly resolved itself, at a volume nobody on the team can hold. That is not a lesser kind of knowledge. It is a different one, and it is the half nobody is shipping.

What the reverse direction carries:

- **The rule you already apply by hand.** Eleven corrections on the same sender is a rule you have and have not written down. The agent can see it. You cannot, because you saw them one at a time across three months.
- **The rule that stopped being true.** A pattern that matched weekly until March and has not matched since is either a fixed process or a broken one. Somebody should be told, and the agent is the only party counting.
- **The contradiction.** What you taught in April and what you taught in July cannot both apply. The extension already requires this to be reported, the one clause that runs agent-to-human, and it should not be the only one.
- **Where its confidence ends.** Show-back, distinguishing settled from unsettled, is the agent telling you what it does not know. That is teaching, and it is often the most useful sentence it can say.
- **The briefing for whoever just joined.** A new person is onboarded by whichever entity has been there the whole time. Today that is a wiki nobody updated. It should be the agent.

None of this needs a new protocol. It needs the same descriptor pointed the other way: a proposal, carrying its evidence, that a named party decides on and an authority records. The teach block already carries a `suggested` field a suggestion *is* the agent teaching the person what it thinks the answer is. That half of the loop already exists; nobody has named it.

What does need a revision. The extension requires the teacher to be a human. The reason is sound: provenance collapse is the one failure you cannot recover from, but the rule is guarding the wrong slot. Accountability comes from the *decider* being a named person, not from the teacher being one. Split the two: record who proposed a lesson, which may be an agent, and who accepted it, which must not be. The same machinery then carries both directions and the audit trail gets *stronger*, because "proposed by the agent on the eleventh repetition, accepted by a named person on a date" is a more precise record than anything the one-way version can write.

Until that split exists this section is one-way by construction, and I would rather say so than let Figure 3 imply otherwise.

C. The rule that cost the most to learn

When an agent asks its question it may attach a suggestion, a prefilled guess drawn from the message in front of it.

[2] A2A's extension governance permits this without registration; the project's own URI prefixes are reserved for artifacts that have been through its process, and this one has not. The extension is not published by, endorsed by, or affiliated with the A2A project.

Suggestions are excellent. People confirm them. That is the point and it is also the problem.

A reference extractor pulling shipment references out of message bodies, given a reference of the form `PA-2026-0901`, matched `PA-2026`. Confirming what looked like one shipment's reference would have taught the directory to claim every reference issued that year, for one customer, silently, from a box someone ticked in three seconds while doing another job.

Nobody would have caught it that day. They would have caught it six weeks later when somebody else's freight kept landing in the wrong name.

A suggestion is a claim about the world, and confirming it is how it becomes one. Its scope is a correctness property, not a convenience.

The defect is written into the specification's rationale on purpose. A document that only lists rules teaches nobody why they are there.

D. Opacity is the load-bearing design choice

A lesson's value is opaque to every party except the learnable. No transport, no authority, no storage layer may read a named key out of it.

The moment one of them reads a field named `pattern`, adding a learnable that is not about text: a threshold, a unit of measure, a column mapping, requires a coordinated release of every component in the chain. That is the difference between a protocol and a shared schema pretending to be one.

The way to hold that property is by construction rather than by review: the authority contains no field named `pattern`, asks the definition which collections exist, and asks the entry to validate its own value. The test of whether you actually have it is a second learnable whose value has no pattern in it at all, a column mapping whose value is an alias for a canonical field, riding the identical code path.

E. Saved is not learned, again

Two requirements come out of watching people use it.

Pinning, restated here because the extension needs it independently of the platform: a lesson that changes anything produces a new immutable version, and a task running against version eleven finishes on version eleven even if twelve goes live mid-flight.

Show-back. When the write succeeds, the obvious thing to display is "Saved." The extension forbids reporting the outcome of a lesson using only the success of the write. The agent re-evaluates the original input under the new definition and shows what it now makes of it, and must distinguish a settled reading from an unsettled one:

Template 3 — the two readings show-back must distinguish

```
settled    "Reads as Northgate Motors Ltd, on
           billing@origin.example"
unsettled  "Still cannot tell whose this is."
```

The unsettled case is why show-back exists. A rule that saved cleanly, validated fine, and does not do what its author meant returns `learned: true`, which is true and completely useless. The unsettled reading is what lets somebody fix it in the next minute instead of the next shipment.

A related rule improves with age: the sentence describing what was learned must be written by the learnable, not composed by the storage layer from field names. An authority does not know that a scope value of "from" means "on the from-line." All it can honestly emit is `customers.northgate.pattern set`, which nobody can audit six months later. The party that understood the value is the only one that can explain it.

VI. The Conformance Surface

What an implementation must adopt, what a profile may rename, and what it may defer

An extension is only useful if two parties who have never met can act on it. Table II is my reading of which clauses carry that weight: what any implementation has to adopt for interoperability to mean anything, what a local profile can legitimately rename or reshape, and what can be deferred to a later revision provided the omission is closed safely rather than silently.

The third column is the one to argue with. It says what each clause buys, or what goes wrong when it is absent, and that is where a reader who disagrees should push.

A. Reading the table

Three things it cannot say about itself.

Renaming and deferring are not the same act. Calling a field by another name is a divergence: cheap to close, and until it is closed no third party can interoperate. Deferring a clause means a behaviour is absent. A matrix that scores both as one column flatters the first and hides the second.

Two of the deferrals are hazards, not gaps. An unrecognised mode absorbed as a teach inverts the caller's intent, which is why the row carries a MUST on refusal rather than a shrug. And an unbounded backtracking pattern, authored by a person in a text box and then evaluated forever, is the kind of defect that arrives as an outage rather than a bug report.

One row is a disagreement I would take to a revision. Learnables belong to the skill that does the deciding, not to the agent. An agent with a message-classification skill and an invoice-column skill can be taught two unrelated things, and an agent-level list forces every client to guess which applies where.

B. The versioning discipline underneath

The extension's own rule is that the URI is the version. New optional fields, new field kinds and new decision modes are additive. Anything that changes the meaning of an existing field gets a new URI and leaves the old one alone. This matters more than it looks: a published URI that quietly changes meaning is indistinguishable, from the far side, from a client bug.

TABLE II
THE CONFORMANCE SURFACE. ADOPT IS REQUIRED FOR INTEROPERATION; PROFILE MAY BE RENAMED OR RELOCATED BY A LOCAL IMPLEMENTATION; DEFER MAY WAIT FOR A LATER REVISION IF THE OMISSION IS CLOSED SAFELY.

Clause	Class	What it buys, or what breaks without it
Declaration and activation		
Extension URI on the wire, echoed on activation	**Adopt**	Without it nothing is negotiated: a client cannot tell a teachable agent from an unteachable one, and the extension is documentation rather than protocol.
Descriptor names the collection, the entry, what it teaches, and how it decides today	**Adopt**	`decidedBy` is what lets a client say "this will match" under rules and "this is one example" under a model. Same button, different promise.
`how` is prose and never a placeholder	**Adopt**	The one field a person reads before deciding whether to teach at all.
Field `kind` is an open string, not an enum	**Adopt**	An enum makes every new field type a coordinated release across every client.
Unknown `kind` renders as text and is never dropped	**Adopt**	The clause that lets a blind client survive an agent it has never seen. Dropping a field silently discards the answer a person gave.
Declared per agent vs. per skill	Profile	Placement follows A2A's agent-scoped extension mechanism, but what can be taught belongs to the skill that decides. I would argue for skill scope in a revision.
The teach block		
Carried in message metadata, keyed by the URI	Profile	Any envelope works provided the key is the URI and not a bare word; a bare key collides the moment a second extension exists.
`suggested` with its label	**Adopt**	Suggestions are why people teach at all. Omit them and the teaching rate goes to nearly zero.
Suggestion no wider in scope than its evidence	**Adopt**	The `PA-2026` defect. This is a correctness rule, and holding it only at each call site is how it eventually gets missed.
Lesson and absorption		
Value opaque to every party except the learnable	**Adopt**	The difference between a protocol and a shared schema. Losing it turns every new learnable into a coordinated release.
Validation delegated to the entry; refusal in its own words	**Adopt**	A refusal a person can act on, instead of a schema error.
Absorbing the same lesson twice reports no change	**Adopt**	Otherwise the audit trail fills with rows saying something changed when nothing did, and the trail stops being readable.
A change produces a new immutable version; tasks pin to theirs	**Adopt**	Without pinning, "it learns" and "it changes under running work" are the same sentence.
`taughtBy` identifies a human	**Adopt**	Provenance collapse is the failure mode you cannot recover from; you cannot remove a bad lesson if you cannot say which lesson it was. Authenticating the calling integration is not the same thing.
`mode: retract`	Defer	May be deferred, but an implementation without it MUST refuse an unrecognised mode. Absorbing an unknown mode as a teach turns a retraction into its opposite.
`expiresAt`	Defer	Answers stale propagation. Deferring it means someone has to notice staleness by hand.
Contradictions reported at absorption	Defer	Two incompatible lessons coexisting, both applied, is the failure that looks like flakiness for months.
Audit sentence written by the learnable, and returned to the client	**Adopt**	Writing it into the record and not handing it back means the person who taught it never sees what they taught.
Show-back, security, privacy		
Re-evaluate the input; distinguish settled from unsettled	**Adopt**	The whole point of Section IV. A success flag on the write is not a result.
Matchers validated at absorption	**Adopt**	An unusable pattern refused at teach time instead of failing on live traffic.
Bounded evaluation time	**Adopt**	A backtracking engine on operator-authored patterns evaluated against every future message is a denial-of-service surface with a human author.
Classification inherited by the lesson; tombstoning	Defer	Answers unauthorised leakage. Defer only where every teacher is already inside one trust boundary.
Third-party retraction path	Defer	The route by which somebody who is not the original teacher can get a wrong lesson removed.

VII. HIERARCHY, BOUNDARIES AND DELEGATION

The piece everyone skips, and the state that is never in the request

I do not much care whether one agent reaches another over HTTP, a queue, a bus or a function call. That is plumbing and it is solved. What is not solved is the property underneath it.

A backend engineer asked about a UI bug does not attempt the UI bug. They say: *"that's front-end, and payments is this sprint's priority anyway, so don't start it today."*

Look at how much just happened. They recognised the boundary of their own job. They knew who owned the other side. They applied a priority ordering that was nowhere in the question. Three separate pieces of state, none of which live in the request, all of which I routinely fail to give agents.

An agent that can only answer is a library. An agent that can say this isn't mine, here's who owns it, and here's why it isn't urgent *is a colleague.*

Delegation is not a transport feature. It is three pieces of held state and a willingness to refuse.

Template 4 — the state a delegating agent must hold

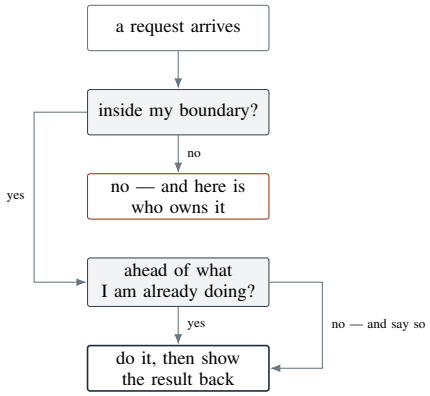

Fig. 4. Three pieces of held state, none of which arrive in the request: what this agent is not responsible for, who owns the other side of each boundary, and an ordering that outranks whoever asked last. An agent missing any of them can only say yes.

```
boundary    what this agent is NOT responsible
            for, written down
ownership   who owns the other side of each
            boundary it can name
priority    an ordering that outranks the
            request in front of it
```

Writing down what a thing is *not* responsible for is an hour of unglamorous work that most teams skip, and skipping it is why so many agents confidently do things nobody wanted.

A. Briefing, and why condensing is the job

Once agents hold that state, the human input changes shape. You stop specifying folder layouts. You say "this front end, that backend, our auth" and it knows what you mean, because it has built this with you eleven times and remembers what you threw out the last ten. The specification got shorter because the shared context got longer, the same trade every good team makes, and the reason a team that has worked together three years communicates in fragments a new hire could not parse.

The read-back has a failure mode of its own, and the complaint I hear most is some version of *please, not another markdown file*. That is the right complaint. A dump is not a briefing. If a person has stopped doing their own research because they have an agent, then condensing **is** the job. Producing thirty pages somebody now has to read in order to find out whether the thirty pages are correct is not delegation; it is a very fast intern who assigns homework. The hard part of an agent's output is not generation. It is selection: what did the reader not need to know.

The same rule applies sideways. An agent should be able to brief another agent. If the only path between two of them is a human copying context from one window into another, that is not a team. It is two chatbots and a courier.

B. Where I draw the line

One line I would hold firmly: an agent asking a *person* and an agent handing off to an *application* must not be the same primitive. A pause that waits for a human carries required roles and an audit meaning; a hand-off to another system is a task. Conflate them and every audit trail thereafter cannot distinguish a human review from a machine call, which is precisely the distinction anyone investigating an incident needs first.

VIII. Careers: Making Growth Legible

Give something a memory and you have given it a biography

Run an agent for two years and the thing being operated is not the thing that was deployed. It has a history, scar tissue, and opinions about a codebase that it formed on a Tuesday in March. Nobody promoted it. Nobody wrote any of it down.

It specialises whether anyone plans it or not. An agent that started on the front end, eighty features later, has absorbed the API contracts, learned which backend changes break which screens, and picked up enough of the deployment story to know why Friday releases go badly. It is quietly full-stack; its job description still says front end. Or it went the other way and is now the only thing in the building that genuinely understands one refund path, including the two exceptions that exist because of a decision made in 2019 and never written down.

Both are careers: one broad, one deep, the same fork every engineer hits around year five, for the same reason: you get shaped by the work you are handed. The difference is that when a person specialises, the organisation notices. Their title changes. People start routing questions to them. There is a moment, with a date on it.

When an agent specialises, nothing marks it. The only honest word for that is drift.

A promotion is an event with a date, a reason, and a new set of expectations. Anything less is your system's behaviour changing while your mental model of it stands still.

An agent that silently got broader is one whose past output you can no longer explain. Someone asks why it made that call in April; the honest answer is that it was a different agent in April and there is no record of the difference.

A. The ladder

Template 5 — rungs, as an explicit and reversible decision

```
level 1  does the task it was asked, shows
         the result back
level 2  holds a memory, needs telling once
         instead of every time
level 3  owns a boundary, says no and routes
         what isn't its job
level 4  teaches, briefs other agents,
         condenses for the human
level 5  proposes work nobody asked for,
         because it sees the pattern
```

The levels are not the point. The point is that moving between them should be a **decision**, made by a person, recorded, and reversible.

Level 3 is where most teams stall, because an agent that can refuse needs a defined boundary, and defining a boundary means writing down what the thing is not responsible for

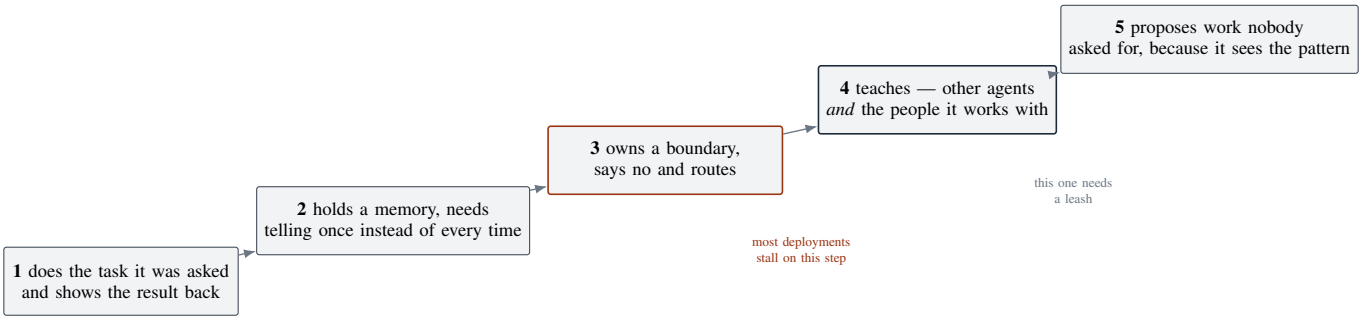

Fig. 5. The ladder. Each rung is a decision somebody makes and can reverse, not a capability that accrues. Rung 4 is the one that matters here: an agent that only teaches other agents leaves the people outside the org chart.

(Section VII's unglamorous hour). Level 5 is where it gets strange: something that proposes work has an agenda, however small, and an agenda needs a leash.

B. The strongest argument against this

It deserves proper space, because it is better than the strawman and it comes from people who have run this in production. Harvard Business Review published a piece in May 2026 arguing that treating agents like new employees is exactly the mistake. The case: the "promote a star performer" instinct actively hurts you, because when you widen an agent's scope there is no underlying judgment widening alongside it. A human you promote has spent two years quietly learning what to escalate. An agent you promote has learned nothing of the kind; it gets more surface area and the same inability to know when it is out of its depth. Their recommendation is a contractor with a narrow statement of work, and systems-engineering templates rather than HR ones: scoped permissions, observability, a kill switch, a named owner.

I concede the core without argument. **Capability does not grow with responsibility.** Nothing about an agent doing eighty good months of front-end work makes it better at knowing when to stop.

But look at what the objection rests on: widening scope is dangerous because judgment does not come with it. Agreed, so the question becomes who decides to widen the scope, on what evidence, and whether that decision leaves a mark. That is the entire argument. The ladder is not a reward. It is a record.

And here is what the objection does not address. An organisation that never promotes its agents does not thereby keep them at level 1. It keeps them at level 1 *on paper*, while the memory quietly fills up with contracts and deployment lore and opinions about the refund path. The scope widens either way. The only choice is whether a person widened it deliberately, with a date on it, or whether it happened while everyone was watching the dashboard.

Take their noun if you prefer it: contractor is more accurate. Keep the machinery. The kill switch and the named owner are not an alternative to the ladder; they are levels 3 and 5 with better names.

C. The review, and the rung nobody builds

If it has a career it can have a review, and the questions turn out to be the useful ones:

Template 6 — the questions a review should force

```
what did it learn this quarter list the lessons,
                                with authors
what does it still escalate the boundary,
                             measured
what does it get wrong often the pattern,
                              not the incident
what should it own now promotion
what should it stop owning demotion, which
                             needs to exist
```

That last row is the one nobody builds. Everyone has accepted that agents accumulate; almost nobody has built the path where an agent gets *less* responsibility because it earned less. A person who keeps making the same call badly gets moved off that work, uncomfortable, and it is how organisations stay sane. An agent that learned something wrong six months ago and has been quietly applying it since needs the same treatment, at the level of the individual lesson rather than the whole system.

Rolling an agent back to last year's snapshot to remove one bad rule is a firing when what you needed was a conversation.

Which is why versioned, attributable, individually reversible learning is not an audit feature. It is what makes performance management possible at all, and it is the same requirement Section V arrived at from the protocol side.

D. The parallel ladder

If agents climb that ladder, the human job climbs a parallel one, whether anyone participates or not. You stop being the person who does the task and become the person who defines the job, briefs the work, reviews the output, and answers for it when it is wrong. That is not a metaphor for management. It is management, with a smaller headcount, a faster feedback loop, and no HR department.

So the skills that appreciate are not the ones the industry spent twenty years selecting for: writing a job description precise enough that the work comes back right, because every

ambiguity left behind gets filled in confidently by something that does not know it guessed; spotting the plausible-but-wrong answer, which is the hardest review skill there is; knowing which decisions can never be delegated, not because an agent could not make them but because you are the one who will be asked to defend them; and deciding when to teach rather than correct, since fixing today's output is cheap and feels permanent while teaching costs more and changes everything downstream.

The reason the human stays a rung above is not capability. It is accountability. When the agent is wrong, nobody in the room is going to ask the agent.

IX. Shared Context Storage

The one structural advantage an agent org has, and the thing that makes it dangerous

This is the most interesting idea in the whole note and it is the one I see almost nobody building for.

Think about why sales knowledge never spreads inside a company. Someone finds a clever way past a gatekeeper. They do not share it. A lead is a lead, the ranking is public, and teaching a colleague materially lowers your own standing. So the single best technique in the building stays in one person's head until they leave and take it with them.

Every organisation on earth leaks value through that hole. A meaningful fraction of management theory is a patch on it: knowledge bases nobody updates, brown-bag sessions nobody attends, documentation written under duress by the person least interested in writing it.

Agents do not have that incentive. An agent has no promotion to protect, no bonus pool to defend, no reason to withhold what worked. Teach one, and every peer can have it before the day ends.

An agent org does not beat a human org because the agents are smarter. It beats it because knowledge moves at the speed of a write instead of the speed of a career.

This is the structural advantage. It is also the only one on the list that a competitor's humans cannot copy by trying harder.

This is no longer only a framing. *Multi-Agent Transactive Memory* (arXiv:2606.19911, June 2026) tested it directly: share problem-solving trajectories across a population of agents and measure what happens. Task performance improved and interaction steps dropped, and the phrase that matters is **"without coordination or joint training."** No retraining, no central orchestrator. Just distribution.

The underlying concept is Daniel Wegner's transactive memory from the 1980s: teams knowing who knows what. It took forty years to build a substrate where distributing that costs nothing.

We are all still benchmarking single agents on single tasks. That is like evaluating a company by interviewing its best employee. The compounding advantage was never in any individual's capability.

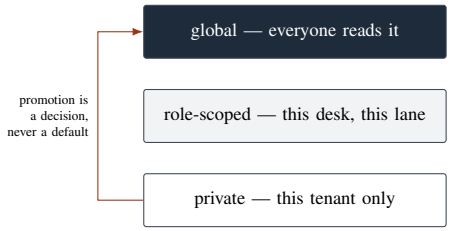

Fig. 6. Three layers, not one pool. A lesson taught from evidence only one tenant may see cannot become global knowledge, however useful it would be. The bug that hurts is not the wrong answer; it is the correct answer reaching somebody entitled to a different one.

A. The caveat, and it is serious

Instant distribution of a good lesson is instant distribution of a bad one. A wrong rule taught at 9am is wrong across the organisation by 9:01, with no friction, no scepticism, and nobody quietly ignoring it the way humans quietly ignore bad process.

Governed Shared Memory for Multi-Agent LLM Systems (arXiv:2606.24535, June 2026) names the four failure modes, and they are the specification for anything anyone builds here:

TABLE III
THE FOUR FAILURE MODES OF SHARED MEMORY, AND THE CLAUSE THAT ANSWERS EACH

Failure	What answers it
unauthorised leakage	the lesson inherits the classification of the evidence it came from
stale propagation	expiry, and a retraction path that is not the original teacher's alone
contradiction persistence	conflicts surfaced at absorption, never silently stacked
provenance collapse	who taught it, when, and why, recorded as a named human

Provenance collapse is the one that will hurt you, and it is the same problem as the demotion problem in Section VIII: you cannot remove a bad lesson if you cannot tell which lesson it was.

Their four primitives (scoped retrieval, temporal supersession, provenance tracking, policy-governed propagation) map onto what you would build anyway if you took the four failures seriously. Which is the argument for building them at the start rather than after the first incident, because all four are cheap in a design and expensive in a migration.

B. Layers, not one pool

The three-layer split that separate teams keep converging on (global, role-scoped, private) is the same shape as classification inheritance. A lesson taught from evidence that only one tenant may see cannot become global knowledge, however useful it would be. Getting that wrong is not a bug that produces a wrong answer; it is a bug that produces a correct answer to somebody who should not have it.

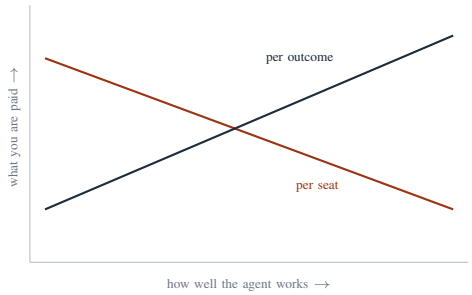

Fig. 7. Per-seat is not merely leaving money behind; it inverts. A good agent means the customer needs fewer people, so the better the product works the less it earns. Outcome pricing is the only meter that points the same way as the thing being sold.

C. Versions, or the thing nobody debugs

If an agent learns, it changes. If it changes, "what did it know when it decided that" becomes a real question, asked six weeks later by someone who was not in the room. So: the change needs a version; the version has to be pinned for the duration of a task; and the decision has to record which version made it. A task that started under version 11 should finish under version 11 even if 12 goes live mid-flight, or you will spend a day reproducing a behaviour that no longer exists.

Least glamorous requirement in agent design. First one you will wish you had.

X. What It Is Worth

A tool competes with tools; a colleague competes with a salary

Buyers do not price things. They price the alternative. Show them a tool and the alternative is another tool: the ceiling is whatever the cheapest competitor charges, and the floor falls out the moment somebody ships a free tier. Show them something that does a job a person currently does, and the alternative is a person. Same software, two completely different numbers.

The market has already moved. Seat-based pricing for AI products dropped from 21% to 15% of vendors in a single year, while vendors charging $800–$2,000 per month for a single agent are closing deals against a salary line rather than a software licence.

Per-seat is not merely leaving money behind; it is structurally broken for agents. Per-seat means being paid per human using the product, but a good agent means the customer needs fewer humans. That is a pricing model that pays more when the product works less well.

A. Token pricing is a trapdoor

Traditional software had near-zero marginal cost. Agents do not: every unit of work has a real cost underneath it, and that cost moves with somebody else's price list. So the safe-feeling move is cost-plus: per token, per call, per compute minute. Run it forward. Models get an order of magnitude cheaper, which they keep doing, and revenue pinned to consumption collapses in lockstep with cost. You did the work; someone else's margin improved.

Price per token and you are a reseller of somebody else's cost curve.

Token pricing meters a cost unit. Outcome pricing meters a value unit. Same mechanism, opposite direction: when the model gets ten times cheaper the token-priced vendor loses ninety per cent of revenue and the outcome-priced vendor books the spread.

The market is already there in places: a resolved support conversation priced at roughly a dollar, competitors between fifty cents and two dollars, and Gartner expecting 40% of enterprise SaaS contracts to carry outcome-based elements by 2026, up from 15%. Price the resolved ticket, the filed return, the reconciled invoice. Not the tokens it took.

B. Which makes memory the asset

If an agent compounds, the version that has worked with a customer for two years is not the same product as a fresh install. It knows their exceptions, their vocabulary, the six things they always do differently, the answer one person gave in March that nobody wrote down anywhere else. A competitor can match the features in a weekend. They cannot match two years of a customer's own corrections, and neither can the customer export them; they were never written down. They exist only as the thing that stopped being wrong.

Which puts a strange obligation on the engineering: the learning path (how a correction gets captured, versioned and made durable) is not a nice-to-have on the roadmap. It is the asset. Everything else on the pricing page is rentable.

XI. Then People Have to Actually Use It

The adoption problem is not a UI problem

None of the above is why these projects fail. This is.

Every process has slack in it. Not the ten or fifteen per cent anyone would admit to in a review, closer to forty. Some of it is genuine waste. A lot of it is the room a human needs in order to keep doing the job at all. A machine does not need wiggle room: give it ten things, it does ten, it comes back for eleven. Push a person to that and they leave, or they quietly break the tool and blame the network.

That last one is not a figure of speech. A survey of 2,400 knowledge workers across the US, UK and Europe in April 2026 found 29% admitting to actively sabotaging their company's AI strategy (44% among the youngest cohort), including refusing mandated tools and deliberately producing low-quality work to make the AI look ineffective.[3] Nearly a third of your users, working against you, on purpose.

Because when you ship the agent that finally gives management complete visibility of how the work gets done, the people doing the work do not experience visibility as a feature. They experience it as scrutiny. The manager gets a dashboard, the company gets numbers, and the person doing the job gets watched more closely for the same money. The monitoring data says the same thing: heavily surveilled workplaces report roughly 42% of employees planning to leave within a year

[3]Reported in *Fortune*, July 2026. I have not read the underlying instrument.

against 23% at unmonitored ones, and where monitoring is disclosed up front reported stress rises 18% against 62% where it is not.[4]

This lands directly on anything built from Section IX, because a governed teaching path has a property a self-modifying agent does not: every lesson has a named author. That is good for audit and it is also surveillance. Anyone designing a teaching interface should decide deliberately which of the two it is, and say so to the people using it, rather than discovering the answer during an attrition conversation.

There is a more hopeful reading of the same fact. The most-used enterprise software on earth is a spreadsheet and an email client, not the most innovative tools available, the ones everyone was already trained on. Day-one productivity beats capability, because there is no adoption curve when there is nothing to learn. And everyone already knows how to work with a colleague: how to brief one, correct one, escalate, hand something over, and tell whether the work came back right. That is a lifetime of training nobody has to pay for. Building an agent that behaves like a colleague is not a poetic framing; it is the shortest path to day-one productivity.

XII. What I Argued Against

Reading this adversarially, part one

A note that lists only what I chose is marketing. These are positions I took against, with the reason.

Dual-writing during a transition. Keeping each product's existing feedback table alive alongside a unified ledger, writing to both. Dual writes with no consumer are a permanent liability nobody ever cleans up, and the rollback story is better served by a flag than by a reconciliation. Legacy tables become seed data and then stop being read.

Resolving dependencies from a branch tip. Fetching sibling repositories at build time with a shared token. It works, and it makes build reproducibility a function of what somebody pushed this morning while leaving a standing cross-repository credential lying around.

Keeping a second grammar out of sentiment. Where two rate-evaluation grammars existed and one turned out to be a strict superset of the other, with the only unique construct expressible in the survivor, keeping both to preserve one flag would have been sentiment rather than engineering.

An "ask the application" primitive. Section VII's line. A pause that waits for a human and a hand-off to another system must not be the same call.

Auto-creating a party. Convenient, and exactly how a notification vendor becomes a customer and misattributes shipments. The clerk proposes; the application decides. This deliberately removes a behaviour people currently rely on, which is why it needs measuring before it is imposed rather than after.

Thresholds as an engine setting. A confidence threshold is a judgment about the cost of being wrong, and that cost differs per product. Centralising it makes one product's risk appetite silently govern another's.

Effort estimates. Earlier drafts of the surrounding programme documents carried engineer-months and line counts. They were removed and none appear here. Neither the authors nor the tooling has a measurement basis for them, and a fluent number gets planned against as if it were one.

[4]Workplace-surveillance trend reporting, 2026.

XIII. What I Hold Loosely

Reading this adversarially, part two

Separating what I would defend from what I am betting on is the only thing that makes the rest of this usable.

The memory moat. It has a clean falsification condition and it is worth stating rather than hiding: if context windows get long and cheap enough that a fresh install can swallow two years of a customer's documents, tickets and message history in one pass and behave as though it lived through them, the moat evaporates overnight. I do not think that happens, because most of what accumulates was never written down anywhere. But that is the bet. What survives even if the bet is lost is governance (who taught it, when, under what authority, and can it be revoked), because that is a compliance property, not a capability property.

Outcome pricing. A position, not a result. The direction of the market data in Section X is real; my own experience of selling that way is not yet.

The ladder above level 2. The rungs above are a hypothesis about how accountability should be structured, not an observed progression. The HBR objection in Section VIII may simply be right.

One shared agent replacing several products' judgment. This is the central technical bet. The identification chains I would consolidate do not even agree on semantics: one is an ordered ladder where the earlier rung wins by construction and there is no scalar confidence anywhere; one is a weighted competition with a scalar confidence and per-tenant thresholds; one is an authority order with human memory at the top. Mapping three different notions of "sure enough" onto one number is where I expect this to be hardest.

That teaching should run both ways. Section V argues it and I believe it, but the evidence is thin in one specific place: nobody has shown that people accept being taught by an agent at the moment it matters. An agent that opens with "you have corrected this eleven times, here is the rule you actually have" is either the most useful thing on the screen or the most irritating, and which one may depend entirely on tone. The protocol change is small. The reception is the risk, and I have not tested it.

Tenancy outside the platform boundary. Where the database does not enforce it, the hand-off's tenant is the sole authority and nothing catches a wrong one. Known and carried, not solved.

Two of the survey figures. Second-hand; I have not read the underlying instruments.

The external work cited here generally. Summarised from reading, not re-derived. Anyone planning against a specific number should go to the source.

A. The question I cannot answer

If agents do level-1 work, where do junior humans get their reps?

The traditional path was: do the boring implementation for two years, absorb by osmosis why the boring things are there, then start making calls. The judgment came *from* the tedium. It was not taught in a review; it was earned in the doing. Take away the tedium and the training ground goes with it while the requirement stays: people who can spot a plausible-but-wrong answer, without the years of being wrong that used to produce that instinct.

The best partial answer I have: make the agent show its work, and make junior people review it. A hundred agent decisions a week, with a senior checking the reviews, might build pattern recognition faster than writing the code ever did.

Might. That is a hypothesis, not a plan, and I would rather leave it open than close it with something that sounds better than it is.

B. What would change my mind

Stating this in advance is the difference between a plan and a conviction.

If a shadow comparison shows a shared agent diverging on cases where a product's existing chain was confident and right, the correct response is not to tune the shared agent. It is to conclude that the judgment being pulled out was product-specific after all, and to leave it where it is. The premise (that several teams built the same thing several times) would then be wrong in the way that matters: they built similar-looking things that encode different businesses.

I do not expect that. I have also not tested it, which is exactly the combination this section exists to make visible.

Closing

There is no agent architecture worth drawing. There is an org design that happens to be implemented in code.

Write the job description. Give it a memory it does not lose, a boundary it knows not to cross, a version number on everything it learns, and a surface a person can teach *and be taught from*. Let it grow into something more specialised than it started as, make that growth legible, and let it teach every peer, human and otherwise, what it just learned. That last part is the one thing a competitor's humans structurally cannot do.

And insist on the return leg. An agent that only ever absorbs is a very expensive apprentice; the arrangement is only worth the attention it costs if the thing that has seen everything tells you what it saw.

Then price it like the job it does.

The 40% that get cancelled will be cancelled for cost, value and control. All three are organisational problems wearing engineering clothes. The teams that survive that cull will not be the ones with the best model.

We spent twenty years learning to build software that forgets. The next twenty are about software that does not, and almost none of the instincts transfer.