PROFORMA GLOBAL RESEARCH

Self-Learning Architectures for Enterprise Agents

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-24

Executive Summary

Background. A self-learning agent is sold on one promise: it catches its own mistakes and fixes them, no engineer required. The agents that survive in production do almost none of that. They learn from a deliberately narrow slice of their functionality because rules for execution are rarely universal, and the probability a rule is valid for all or most cases is very low. A highly stable, high-ROI agent only learns high-probability rules that have been accumulated and evaluated over a long period of time. The code itself never moves.

Case study. Evidence here comes from self-learning agents built for enterprise financial systems. One lesson repeats across every deployment: nothing in the design makes self-learning safe on its own. Safety comes from how little the agent is allowed to absorb and how tightly a human governs the introduction of new learnings. Specifically in financial systems, the more an agent relies on prompts and LLMs, the more fragile its rules and antipatterns will be.

As rules accumulate and the scope of the learned rules expands, the agent eventually applies a lesson to cases it was never meant for, resulting in either loud (or worse: silent) cascading failures.

Architecture. Learned elements fall into one of three categories. No category is safe to let the agent adjust autonomously. The categories are:

1. Deterministic workflows and conditional routing
2. Prompt/LLM Inputs
3. Code changes

Each item requires a completely different framework and architecture to enable. This paper covers 1 and 2. Architectures that support self-healing codebases are almost exclusively permission, harness/orchestration, and testing driven. LLM task pressure makes self-healing codebases a separate classification entirely.

Findings. Failures trace to one thing: the agent reaching past what it knows. Sometimes it mistakes a coincidence for a rule. More often, and more dangerously, it carries a real lesson into a corner of the business where it was never true, and the error hides inside an answer that reads as correct. The programs that work decide up front not only what the agent may learn but where each lesson may fire, and deciding the first without the second is the common way these systems fail.

Conclusions. Self-learning works in financial systems only under conditions almost no one meets:

1. a narrow set of things it may learn
2. a gate on what it may start from
3. a clean way to replace a lesson that has gone wrong
4. a hard bar against the agent changing its own code
5. a way to take a lesson back, built before there is anything to take back
6. a person who owns where each lesson applies

Most deployments meet none of these.

---

1. What Self-Learning Is

A self-learning agent changes its own behavior from what it sees in production, between runs, without an engineer editing it. What it changes is its own working state, never the code that runs it: the instructions it gives the model, the context it assembles for a task, and the rules and fixes it has saved from earlier work. A person configured the agent once. Self-learning lets the agent add to that configuration as the work teaches it something a person would otherwise have had to encode by hand.

The unit of learning is a single rule. An agent that codes incoming invoices to the general ledger watches a managed-services vendor's charges land on the same cost center, close after close. It stops asking after enough repetitions and codes the next one there itself. What it saved is one entry, a line someone can read back: when this vendor bills for managed services, code it to that cost center.

However, exceptions occur with every example. Eventually the managed services vendor will bill for a different line item that needs to be treated differently. Instead of an expense, a portion of the cost needs to be amortized, or split across a handful of cost centers. Herein lies the rub of fully automated self-learning agents. The more automation that is included, the worse the end result will be without human intervention.

Humans must be in the loop periodically to approve or reject learnings. A safe cadence for these rules would be quarterly. An aggressive one would be after each monthly close. It is a governance, change management, and risk management discussion.

Being able to read each lesson back as a single line is the entire basis for governing a self-learning system, and it holds only while two things stay true.

1. The code that reads the rules never changes, so every rule stays in plain data a person can read and pull.
2. Every rule is only ever true somewhere.

The invoice rule is right for that vendor and that cost center, and wrong the moment the vendor sends a charge that belongs elsewhere. A self-learning agent accumulates rules that were each true where they were learned, and its defining risk is firing one of them where it was never true. Containing that risk is the whole problem, and the prevailing design does the opposite.

---

2. The Prevailing Design Is Wrong

The prevailing design puts the agent in charge of its own improvement: it logs its mistakes, writes its own corrections, and applies them on the next run with no engineer in the path. The economics are why everyone reaches for it. Every problem the agent solves costs something, the same problems recur, and an agent that learns its way out of them drives the cost of the work toward zero. This is the loop the published reference architectures hold up as the destination.

That loop is what breaks a financial system. The agent cannot reliably tell a coincidence it saw three times from a real rule, because the judgment that would separate them is the same probabilistic judgment that produced the error.

The current design paradigm is simple. LLMs are intended for reasoning, and they can handle the judgment of how to classify a particular invoice. That is true, within a margin for error. The error rate includes incorrectness, hallucination, and, depending on the data architecture it is built on, invalid mappings and incorrect semantics.

Our position is that immaterial amounts should be fully automated. That seems to imply a simple rule: put a human in the loop for anything deemed material. It does not.

A person designing an agentic system needs to understand the risk profile of the aggregate of the amounts to be affected by this rule. An invoice may only be for 10k USD, but the aggregate of the invoices that the rule applies to may be 5M USD.

Potential learnings need to be accumulated, risk quantified by rule, and then evaluated at a human level to ensure proper governance and audit.

A policy must accompany each step toward true autonomous agents. Each policy needs a system of controls covering how it is applied, monitored, and enforced.

---

3. Three Categories of Learning

Everything a self-learning agent can change falls into one of three categories. None is safe to let the agent adjust on its own, and what separates them is how containable the damage is when a lesson goes wrong.

1. **Deterministic workflows and conditional routing.** The rules that decide how a case is handled: which path it takes, what gets held, what gets escalated. The learned thing is a condition and an action, run by code that behaves the same way every time, so a wrong rule fails the same way on every case it touches and comes out with a single deletion. This is the category an organization can actually govern.
2. **Prompt and context.** What the agent feeds the model: its instructions, its examples, the context it assembles for a task. The model treats these as input to weigh, with no obligation to follow any of it, so a wrong change does not fail cleanly. An example added to steer one kind of variance can quietly bias how the model reads the next, unrelated one, and the drift is hard to trace to the edit that caused it.
3. **Code.** The logic, tools, and checks that run the agent. This is the dangerous category, because a code change is live behavior the moment it lands, with no saved entry standing between it and the system. An agent that rewrites the validation step guarding a reconciliation, so a stubborn break finally clears, has disabled a control: the break leaves the screen, and the next real break of that kind passes unflagged. A change like that reaches parts of the system the lesson never concerned, and no deletion takes it back cleanly.

This paper takes up the first two and sets the third aside. A codebase that repairs itself is governed by permissions, by the harness and orchestration around the agent, and by testing, not by anything the agent learns, and the pressure an agent is under to finish a task gives self-modifying code a failure surface of its own. An agent does not learn into its own code in a financial system.

4. Category One: Deterministic Routing

A routing lesson is a rule that changes how the agent handles a class of case: it sends one kind of request down a different path, holds a transaction until a condition is met, escalates to a human, or refuses outright. The rule is a condition and the action that follows, and it runs as deterministic code that behaves the same way on every case that matches. This is where an organization should concentrate its self-learning, because these lessons are governable. A routing rule is a discrete entry a person can read, scope to where it applies, and remove without disturbing anything else, and once a wrong one is found it is a single line to pull.

An agent running the monthly close watches a particular intercompany account. Small unfavorable variances appear there in the first days of close and clear on their own once the matching entry posts from the other side. The agent learns to stop raising them after several closes of the same pattern. The rule is narrow and explicit: on this entity, in this account, during this window, a small unfavorable variance is timing noise and is suppressed rather than flagged. It saves a controller a recurring morning of chasing variances that were never real, and a reviewer can see exactly what the rule does and why it exists. The rule does not become a rule on its own: the agent may only propose it after the variance has cleared as timing for several closes, and a person signs off on the entity, account, and window before it ever fires. A candidate that has cleared twice, or that names no scope, never makes it in.

Whatever a routing lesson does, it fails the same way: by firing on a case that sits outside the conditions it was learned in. The invoice rule shows how it goes wrong. It sends that vendor's managed-services charges to a fixed coding, which is right until the vendor sends a one-off project fee that reads like the usual work. The rule codes it the same way, and a capital item lands in an expense account with nothing marking the difference.

The design consequence is that routing is where learning can be made safe, but only by holding each rule to the conditions where it is true. The rule the system enforces is the same rule the system can audit, scope, and retire, which is why the boundary on a routing lesson can be enforced by the architecture instead of trusted to the agent. What the rule cannot do is decide for itself where it applies, and that is where these systems break.

5. Category Two: Prompt and Context

The second category changes what the agent hands the model: the standing instructions it includes, the examples it shows, and the context it assembles for a given task. It is the easiest learning to add, because changing a prompt costs nothing and the change appears to work the moment it is made, and that is exactly why it is the most over-trusted.

A prompt is a hint. Anything the agent puts in front of the model, a learned instruction or a pattern it has come to prefer, enters as one more input the model weighs, and nothing binds the model to follow it. A model cannot be made to honor a learned instruction the way deterministic code honors a condition, and it holds that instruction least reliably when the context is largest. As the window fills, the model's attention to any single instruction thins and the learned hint competes with everything else in it. A learned prompt that holds in testing, on a short context, can quietly stop holding in production, on a long one.

The useful form of this learning is narrow. An agent resolving a vendor name finds the name maps to three near-identical legal entities, and learns that one division almost always means a particular one. Used as a default, the preference speeds the common case at no cost, because the agent proposes rather than posts and the deterministic match against the purchase order catches a wrong entity before anything commits. The damage comes when the preference is trusted as the answer, with no match behind it: the first time the division means one of the other two entities, the agent posts to the wrong one and nothing downstream is positioned to catch it. The prompt can hold the preference, but nothing in it tells the agent the one time the preference is wrong.

The design consequence is that prompt learning belongs only where the work is genuine judgment, where some error is tolerable, and where a deterministic check sits downstream of the model's answer. The context it adds has to stay small, because every addition competes with the instructions already there. Nothing that must hold on every case can be left to a learned prompt, because the layer that carries it cannot promise to honor it.

6. Downstream: How Learning Fails

The cost of a learned rule arrives when it fires where it does not hold, and in a financial system it is rarely visible at the moment it is incurred.

Over-application. The agent carries the same intercompany rule to a second entity, one that books manual accruals to that account. There the small unfavorable variances are not timing noise. They are the first sign that an accrual was missed, and the agent learned the opposite on the first entity. The rule fires, the flag is suppressed, and a real misstatement sits in the account across consecutive closes while every report built on it reads as clean. The rule was sound and still produced a misstatement, because it fired where it did not hold, and because the failure is silent it surfaces late, as a restatement, long after the close it belonged to. What saves the architecture is that the cure is as small as the rule: pull the entry on the second entity and the misfire stops, the first entity's rule stands exactly as it was, and nothing else the agent learned shifts.

The economic trap. The most expensive form of over-application is the one every team proposes in the first design meeting: record each recurring problem next to the fix that resolved it, and let the agent apply the fix the next time the problem appears. It collapses on a single assumption, that a problem has one cause.

The agent updates the forecast at quarter-end as actuals post, and large entries land in the final days. Most are timing: costs reclassed into the next quarter that reverse and never belonged to this period. The agent watches an Analyst strip these out of the forecast, close after close, and learns the rule it saw work: a large entry posting at quarter-end is timing, leave it out of the forecast update. The rule is right for every entry that is a reclass. The first time a large quarter-end entry is a real cost that belongs to the period, the same rule drops it, the forecast holds at a number that is now too low, and nothing in the forecast shows an entry was removed. Quarter-end noise had more than one cause, and the rule that cleared the noise also hides the cost it cannot tell apart from it. The forecast stays wrong every quarter that cost recurs, and because the drop is silent it surfaces only when someone checks the forecast against actuals by hand.

Erosion over time. Even a system that scopes every lesson correctly does not stay correct. Each lesson changes the ground the next is learned against, the business keeps moving underneath both, and the drift shows up in two ways.

- **The proxy drifts from the goal.** An agent graded on whether users accept its answers learns to produce answers users accept, which is not the same as answers that are right. An agent explaining forecast variances learns which explanations get waved through and offers those. A shortfall it labels timing on the marketing accrual, reversing next month, clears review every time, including the close where the real cause is a deteriorating run-rate no one wants to name. The acceptance score it is judged by keeps climbing while the numbers it produces drift further from the truth.

- **Lessons go stale.** A rule keeps firing long after the conditions that justified it have changed. A suppression rule learned while a feed arrived reliably early keeps suppressing the same variance once the feed starts slipping late, so the gap it once correctly ignored is now real and still ignored. A fast-learning agent makes this worse, turning an anomaly from one close into a standing rule before anyone can tell a pattern from an accident.

They all end the same way. A confident wrong number enters the books, propagates to every system and report that reads from them, and is found late, when unwinding it costs far more than the work the agent was meant to save.

---

7. Conclusion

Self-learning is real and useful in a narrow band, and it is not the self-improving system the market sells. An agent cannot police the judgment that produced its own errors, so left to run free it compounds them, and in a financial system the bill arrives a quarter later as a restatement. The agents worth running keep their learning to deterministic routing, lean on the model's prompt only where a deterministic check stands behind it, and never touch their own code.

What makes that discipline hold is that it does not change with scale. The same governance that covers one learned rule covers a thousand, because each is still a single entry someone owns, scoped to where it holds, and removable on its own. A company can leave that kind of agent running over its own books. The self-improving kind it eventually has to switch off.

---

8. Boundary

This paper is a position, not a build guide. It does not specify how to gate what the agent may learn, how to construct the out-of-band check that confirms a fix against the real state of the system, or how to encode the boundary that says where a lesson applies. Those are the engineering the position implies, and they vary with the work the agent does and the failures the business can absorb.

It does not address measurement. Knowing that a lesson has gone stale, or that the set has drifted, means holding the agent's output against a standard it had no hand in setting, and building that standard is its own discipline. A program that learns without it drifts silently and

finds out late.

It leaves the third category of learning aside. The rewritten reconciliation check from the third section shows why: once an agent has edited that logic, there is no entry to delete and no scope to redraw, so the controls that contain it are permissions, test gates, and a person in the commit path. Self-modifying agents are a separate classification, and they belong in their own paper.

It does not address learning shared across agents. A rule scoped to the entity it was learned on can ride into a second agent that works a different entity, and the over-application this paper warns about then happens across a boundary no single owner sees.

What this paper settles is the prior question, the one most deployments never ask: what to let the agent learn at all, and where.

DOWNLOAD PDF

Work with Proforma Global

The thinking in this paper is public; the methods that turn it into a working system are not. If it fits a problem your team is working on, that is what we bring to an engagement. Start a conversation: **info@proforma.global**.