PROFORMA GLOBAL RESEARCH

Enterprise Agents in Financial Systems

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-18

Executive Summary

Background. A language model cannot be trusted with a business process that touches money. It can be trusted with pieces of one. The line between those two statements is where most enterprise agent programs fail, and no framework draws it for the architect. The discipline that draws it correctly is deterministic-first design, and it is not the default of any agent stack on the market.

Case study. Evidence in this paper comes from agent systems built for enterprise financial environments. One observation holds across every deployment: language models do not solve business problems. They design approaches, judge ambiguity, and fill gaps that rules cannot close. Asking a model to run a business process end to end is the failure mode wearing a friendly interface.

Architecture. Production business agents are deterministic-first. Every fact that a rule can verify is verified that way before the model is invoked. The agent compiles what is known, identifies what is missing, and only then calls the model on the narrow questions the rules could not answer. The model fills gaps. Code does the work. The risk profile of the process decides how much human review the workflow demands, and that decision is encoded in the workflow itself.

Findings. The most common architectural error in current agent design treats tool calls as deterministic primitives. They are not. A tool call is a prompt and a schema. The model decides whether to call the tool, what arguments to pass, and how to read the return. Each decision is subject to the same hallucination behavior as any other model call. Tool use fits where creativity is the work. In business-critical processes it imports the failure surface the architecture was supposed to remove.

Conclusions. Business processes need rigid execution and tolerate close to zero hallucination. Chatbot patterns work in chatbots because a human is in the loop. In financial workflows the human is not in the loop most of the time, and the cost of a wrong decision is paid in remediation work the agent was supposed to prevent. Agents that can be trusted with money run on deterministic primitives invoked by orchestration code, with the model used for judgment, tools confined to creative work, and risk rating treated as an architectural property rather than a deployment switch.

1. Where We Are

Agent architecture above the work layer is now well-described. The architecture below it is not, and that is where business agents fail. The orchestration shape an architect picks is the easier decision. Deciding who owns each task inside that shape, a language model or deterministic code, is the harder one, and it is the decision most teams get wrong.

Every task inside an agent has an owner. The owner is a language model, a piece of deterministic code, or a hybrid that uses both. The wrong defaults look like the right answer until the work hits production.

For business processes that touch money, the cost of getting it wrong is concrete. A wrong invoice match produces a payment that should not have happened. A wrong reconciliation hides an error that compounds across a quarter. A wrong classification flows into the general ledger and a controller spends two weeks chasing it down. Agents that can be trusted with that class of decision are built on a discipline most teams skip.

2. The Misconception and the Discipline

The widespread misconception puts a language model in charge of a business process end to end, whether the process is an invoice, a reconciliation, or a customer dispute. The framing makes the model the solver. The model is poorly suited to that position.

A language model does not solve a business problem. It designs approaches, judges ambiguity, and fills gaps that rules cannot close. Those activities are real and valuable. None of them is the work itself.

Asked to handle an invoice end to end, the model has no mechanism for enforcing the rules of the invoice. Nothing in the prompt verifies that the purchase order exists, that the customer number is in the master list, or that the line items reconcile against receiving. The model produces an output that looks like the right answer, and the rules it was supposed to enforce live nowhere.

The right pattern inverts this. Every fact a rule can verify is verified that way before the model is invoked. The agent walks a sequence of deterministic checks: whether the invoice carries a purchase order, whether that PO is valid, whether the customer number matches the master list, whether line totals reconcile, whether the tax calculation matches the configured rate for the jurisdiction, whether the vendor is approved. Each check returns pass, fail, or indeterminate. The agent compiles the results.

Only then does the model enter, scoped to the narrow questions the rules could not close: a line description that matches no SKU, a vendor name that resolves to three close candidates, a tax rate the deterministic logic could not place. The model judges these with full context of what is already known and returns a structured answer the workflow consumes.

Claude Code uses exactly this pattern in software engineering. The model designs the change, decides which files matter, and proposes the modification. The reading and writing runs as deterministic code. The model contributes intent and judgment; deterministic code provides reliable execution. The same pattern applies in financial systems, with the deterministic primitives replaced by the checks the business process actually requires.

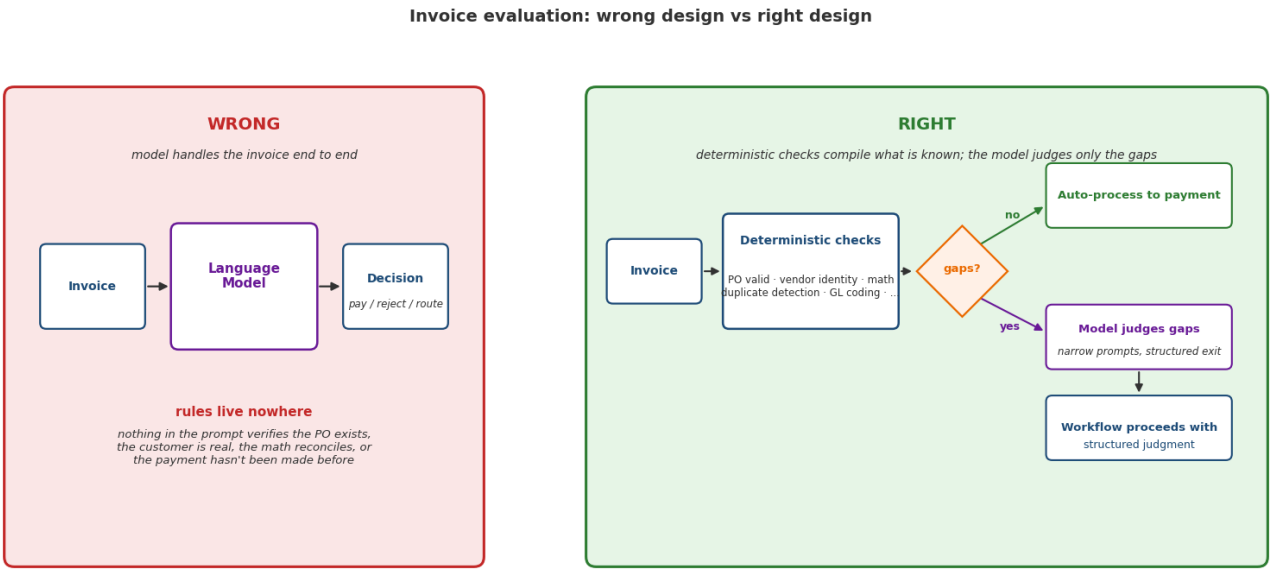

Figure 1: Invoice evaluation, wrong and right. The wrong design hands the invoice to the model end to end; rules live nowhere and confident output hides the absence of verification. The right design runs deterministic checks against everything the business already knows, leaves only the gaps for the model to judge, and proceeds with structured judgment to the next step.

3. The Tool Misconception

The architectural error most likely to defeat a business-process agent treats tool calls as deterministic primitives. They are not. A tool in a language-model framework is a function description and a calling convention. The model reads the description, decides whether the situation warrants calling the tool, generates the arguments, receives the result, and interprets it. The function body is deterministic. Nothing wrapping it is.

Three sources of non-determinism live inside every tool call: the decision to call, the arguments generated, the reading of the return. Each can hallucinate, misclassify, omit, or misread in the same way the model can do those things in any other call. Wrapping deterministic code in a tool and exposing it to the model does not move the system toward determinism. It buries the deterministic part of the system inside a non-deterministic shell.

Work where the surrounding judgment is the point of the system tolerates this fine. Research, content generation, conversational assistance, novel-problem investigation. The model deciding when and how to use a tool is the work the architect wanted the model to do.

Critical business processes are different. Every place the model's judgment enters is a place hallucination can enter. Letting the model decide when to call a tool, with what arguments, and what to do with the result imports the failure surface the architecture was supposed to remove. The determinism of the function body does not propagate to the decisions around it.

The default for business processes should lean away from tool calls and toward deterministic primitives invoked by the orchestration code itself. The orchestration code walks the workflow, calls the primitives, and decides what to do with the results. The model is invoked only where judgment is the actual work, on narrow tasks with focused context. Tools have a place in this architecture, but a small one, and they are not the default.

The misconception persists because tool use feels architecturally clean. The function being called is deterministic, the framework supports the pattern out of the box, demos run on simple cases. None of those properties predict the system will hold up in production on work that does not tolerate the failure modes tool use imports.

4. A Worked Example: Invoice Evaluation

An invoice arrives. The business question is whether to pay it, against what purchase order, in what amount, on what schedule, with what coding to the general ledger. The wrong design hands the invoice to a language model and asks for the answer. The right design runs deterministic logic over everything the business already knows and gives the model only the questions the rules cannot close.

The deterministic checks that belong in any production invoice agent include the following.

- **Purchase order presence and validity.** The invoice references a PO number. The PO exists in the system. The PO is open and unliquidated. The PO has remaining capacity for the invoice amount.
- **Vendor identity.** The vendor on the invoice matches an approved vendor record. The banking details on the invoice match the banking details on file. The vendor is not on a sanctions list.
- **Customer or entity identity.** The bill-to entity matches a legal entity authorized to receive the goods or services.
- **Line item reconciliation.** Each line on the invoice has a matching line on the purchase order. Quantities billed do not exceed quantities ordered. Unit prices match the negotiated rate or fall within a configured tolerance.
- **Receiving confirmation.** Goods invoices reconcile against the receiving report. Service invoices reconcile against a service confirmation, timesheet, or milestone signoff.
- **Math.** Line subtotals add to the invoice subtotal. Tax is calculated correctly for the customer jurisdiction. Discounts are applied per the terms. The invoice total reconciles end to end.
- **Duplicate detection.** The invoice number has not been processed before for this vendor. The combination of dollar amount, vendor, and date does not match a prior payment.
- **Approval threshold.** The invoice amount falls within the approval authority of the next reviewer in the chain.
- **Payment terms.** The terms on the invoice match the terms on the PO. The due date is calculated correctly from the invoice date and the terms.
- **General ledger coding.** The expense category derives from the PO's category mapping. The cost center derives from the entity and the requisitioner. No coding requires interpretation if the PO was correctly issued.

Each check returns pass, fail, or indeterminate. The agent runs them in dependency order and compiles the results. By the time compilation completes, the agent holds a full picture of what is known about the invoice and what is not.

The model is invoked only on the gaps. A line description that does not map to a known SKU through fuzzy matching. A vendor name that resolves to three candidates with similar legal names. A tax calculation that produced a non-standard rate the rules could not place. A duplicate-detection result where the invoice number matches a prior payment but the dollar amount differs by an amount that suggests a credit memo rather than a duplicate. The model judges each with full context of what the deterministic layer established and returns a structured judgment the workflow uses to proceed or to route for review.

Failure points of the wrong design are visible in the inverse pattern. A model asked to evaluate the whole invoice may approve a payment against an expired PO because expiration was never checked, classify a duplicate as a fresh invoice because the payment history was never queried, or produce a tax calculation that looks right and is computationally wrong because the rate was estimated rather than computed. None of those failures show up in the model's output. The model produces a confident answer in every case. The cost is paid downstream when the wrong payment leaves the system.

---

5. Risk-Rated Orchestration

A real business does not have one workflow. It has dozens, and they cannot all run at the same level of automation. A $500 supply invoice from an established vendor with a clean history can move through end to end. A $5 million invoice for professional services with a complex coding structure cannot. The architecture handling both shares its primitives and its shape. The orchestration changes based on the risk profile of the work.

Risk has to be encoded into the workflow as an architectural concern, not added later as a policy. Risk classification happens early, often as the first deterministic check the agent performs. The risk score then routes everything that follows.

Figure 2: Three risk tiers, one architecture

Risk tier	Deterministic checks	Model on gaps	Human review
Low	Full sequence	Within confidence threshold	None
Medium	Full sequence	Yes	On any indeterminate check, low-confidence judgment, or cumulative ambiguity over threshold
High	Full sequence	Yes	Mandatory on every model decision, full audit trail of all checks and judgments

The tiers share the deterministic spine. They differ in where the human enters and how much of the model's judgment is treated as final.

A team that skips risk-as-architecture ends up with one of two failures. The workflow is conservative everywhere, which makes low-risk work too slow and too expensive because it is being treated like high-risk work. Or the workflow is permissive everywhere, which means high-risk work runs without the controls the risk demanded. Both failures are common and both are expensive.

The risk model is designed first. The orchestration is built around it, routing each piece of work through the level of review its risk profile demands. The primitives do the underlying work regardless of tier.

6. The Default Problem

Framework defaults produce the opposite of this discipline. The default agent loop hands the model a problem and a set of tools and asks it to figure out the answer. Tool use is structured so the model decides when and how to call deterministic code. Risk is treated as deployment configuration rather than an architectural decision. No default looks unreasonable in isolation. Stacked together they produce systems that are not safe to point at money.

Agents that are safe invert these defaults:

- Deterministic primitives mirror the business process and do the actual work.

- Tools are treated as a misleading abstraction in business-critical contexts and used sparingly.
- Risk rating is encoded into the workflow so human review is applied where it is required and skipped where it is not.

The discipline costs more at design time and less at every other point in the system's life.

A finance team automating its invoice process is not trying to demonstrate that an agent can read an invoice. It is trying to remove cost from a process that runs every day. The cost the agent removes has to be greater than the cost the agent creates. Deterministic-first design is what makes that math work. The opposite produces an agent that demos well and is unsafe to leave running.
