PROFORMA GLOBAL RESEARCH

Where Agent Orchestration Breaks

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-18

Executive Summary

Background. The market has decided language models should orchestrate enterprise workflows. They should not. As the prompt an orchestrator carries grows to cover every discipline a complex task touches, the instructions inside that prompt stop being attended to. The model continues to answer, but the answers drift toward invention. In domains that tolerate a few percent error this is an inconvenience. In financial systems, where the cost of a wrong answer is paid downstream by every system that reads from the result, it is not acceptable. The cost of using a language model in the orchestrator's seat shows up in four currencies: failure, inconsistency, latency, and operating spend. All four are symptoms of putting a reasoning component where ordinary code performs the work better.

Case study. The evidence here is drawn from an AI agent built for enterprise financial systems, with Oracle EPM Cloud as the operating environment. Complex tasks in this domain span four disciplines at once: finance, software engineering, platform knowledge, and the customer's own business. The pattern holds across every class of task in this environment: object creation, troubleshooting, error detection, reconciliation, configuration validation. The orchestrator must honor all four disciplines on every decision, regardless of which task is running.

Architecture. The architecture is a clean split into four layers. Orchestration runs as a deterministic predefined workflow that exposes cross-discipline interaction points at the gates between steps. Intent detection and the reasoning steps inside the workflow are handled by the language model, each invocation scoped to the narrow question being asked, with curated context. Action (querying systems, changing state, calling APIs) runs as deterministic code. The deterministic layers hold the cross-discipline state. The model is asked one question at a time.

Findings. The failure mode is mechanical and predictable. As the prompt grows, the model's attention to any individual instruction degrades, and the output drifts into invention that passes surface checks and fails when the work meets reality. Decomposing the work across sub-agents does not fix this; it makes the cross-discipline rules harder to see by removing them from any single agent's scope. Agents

built around the architectural split below operate at materially lower cost, higher reliability, and lower hallucination risk than agents that route every decision through a model.

Conclusions. Language models are reasoning tools. Use them where reasoning is the work. Put them everywhere else and they fail. Multi-discipline enterprise agents that respect the split described above are reliable, fast, and economically operable. The decision framework in Section 6 identifies where the split is required.

---

1. The Mechanism

Language models fail as orchestrators of highly complex tasks that span multiple disciplines. The failure is hallucination. The cause is mechanical: as the prompt grows to carry everything an orchestrator needs to coordinate the work, the instructions inside that prompt stop being attended to.

A model orchestrating across multiple disciplines has to hold the rules, the vocabularies, the constraints, and the business context of every discipline at once. The system prompt expands. Retrieved context gets appended. Conversation history accumulates. The prompt grows from five to ten to twenty thousand tokens, not because the window is full, but because the orchestrator's job requires that much loaded context. The window has room. The attention does not.

The model still answers. The answers degrade in a specific way. The narrow instructions that were supposed to prevent invention (*use only the identifiers in this list, do not invent values, validate before proceeding*) become a small share of an enormous prompt, and the model's attention to them drops. It reverts to its prior, generic understanding of what the answer should look like. It hallucinates.

The hallucination does not surface immediately. It looks correct. It satisfies syntactic checks. A downstream step inherits it and reasons against it. When the failure finally surfaces, the orchestrator tries to repair the immediate symptom, breaks something else, and the run cascades through compounding fixes until it gives up. By that point the system being acted on is in worse condition than when it started.

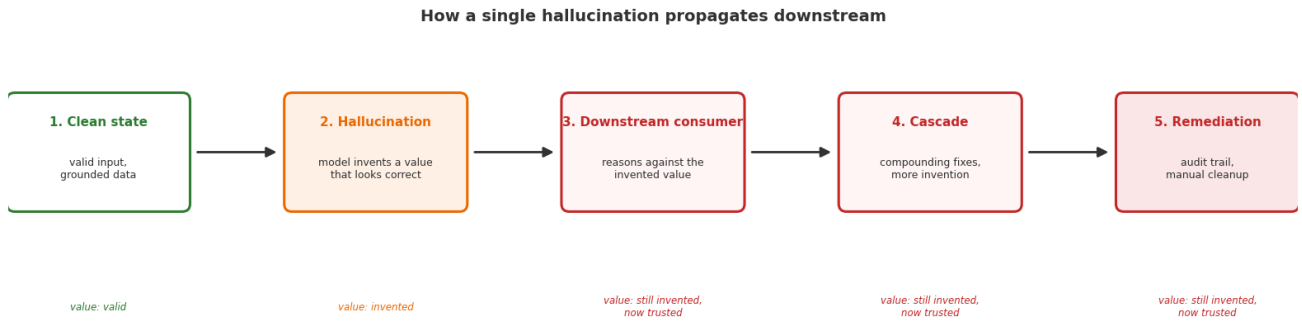

Figure 1: A single hallucinated value introduced at one step is treated as authoritative by every step downstream. Repair attempts compound the error. By the time the failure surfaces as a remediation event, the system being acted on is in worse condition than when the agent started.

In domains that tolerate a few percent error rate, this is an inconvenience. In financial systems it is not acceptable. A wrong number on a deployed configuration corrupts data that downstream consumers trust. A wrong identifier on a master data record propagates through every system that reads from it. The cost of a single hallucination in this environment is not "rerun the agent." It is a remediation project with an audit trail.

The architectural mistake is letting the orchestrator carry that much context in the first place.

---

2. Three Failure Patterns

The mechanism produces failures of consistent shape. Three are common enough to name.

Pattern 1: Guardrails crowded out

The instructions that were supposed to prevent invention sit inside a prompt large enough that they receive a small share of the model's attention. *Never invent an identifier. Only use values from the reference list. Validate before proceeding.* Each is correct, present, and ignored. The model reverts to its prior understanding of what the answer should look like and produces output the guardrails were written to prevent. The architectural answer is to give each model call a focused prompt where the active constraints carry weight.

Pattern 2: Generalization beyond the grounding

The model is given grounded data (a list of valid options, a set of company-specific identifiers, a catalog of allowed values) and asked to pick from it. Instead of picking, it generalizes. The output looks like an item from the list but is not on the list. It passes syntactic validation because it matches the pattern. It fails when a downstream consumer queries the invented value and finds nothing. The architectural answer is to validate every choice the model makes against the grounding before it leaves the boundary. Permission to generalize is permission to hallucinate.

Pattern 3: Silent degradation

A step in the pipeline fails partially. Retrieval returns half the records. A validator times out. An upstream call hits a rate limit and returns nothing. The agent continues on the degraded signal because nothing marked it as suspect. Downstream steps treat the partial output as authoritative and reason against it. The wrong answer is indistinguishable from the right one until it surfaces somewhere expensive. The architectural answer is that degradation must be loud. If a step cannot run with full integrity, the workflow stops or escalates.

---

3. Multi-Discipline Complexity

A discipline is a body of knowledge with its own internal rules, vocabulary, and quality standards, maintained by people who specialize in it. A complex task in an enterprise financial system touches four of them at once.

Finance carries the rules that decide what the work means. Is this an asset or an expense. Does the calculation match the way the business actually computes the number. Does the value belong in this period or the next.

Software engineering carries the rules that decide whether the work will run. Does the configuration conform to the schema the platform will accept. Are the calls made in the right order. Does the deploy succeed without breaking what was already there.

Platform carries the rules that decide how the work behaves once it is in the system. How a dimension change propagates. How a security setting cascades through inheritance. How the calc engine broadcasts a value. These are emergent properties practitioners learn by watching the platform run.

Customer business carries the rules that decide whether the work is correct in this particular environment. The chart of accounts. The org structure. The planning cycle. The conventions the company adopted ten years ago that nobody documented but everyone still uses.

The rules that cause trouble are the ones that span disciplines. A finance rule that produces a constraint the platform has to respect. A platform behavior that constrains how a calculation can be written. These rules are owned by no single discipline. They emerge from the intersection, and they are exactly the rules that get crowded out as the orchestrator's prompt fills with everything else.

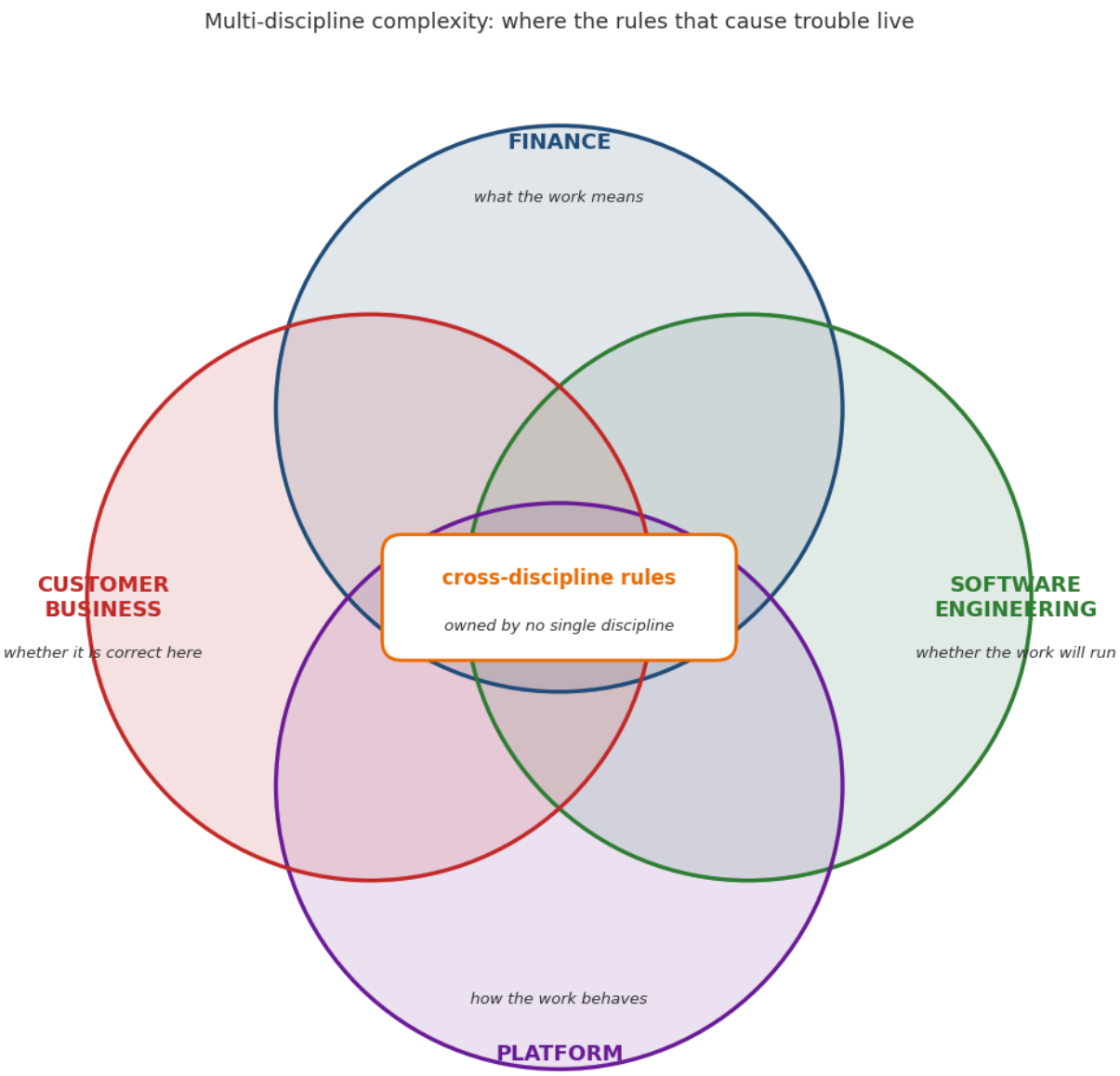

Figure 2: A complex task in an enterprise financial system touches four disciplines at once. The rules that cause trouble live in the central intersection, owned by no single discipline. These are the rules the orchestrator's expanded prompt crowds out.

4. Why Decomposition Stops Helping

The standard remedy when an agent gets overwhelmed is to break the work into smaller agents. Smaller scope, smaller context, cleaner reasoning. For single-discipline problems this works. For multi-discipline problems it makes the failure worse.

A sub-agent scoped to finance does not see the platform constraint. A sub-agent scoped to the platform does not see the finance reason behind the input it received. Each reasons correctly inside its own scope and produces output that violates a rule living outside its scope. The orchestrator above them was supposed to catch the violation. It cannot, because the context that would have surfaced the rule was the same context decomposition stripped out.

Decomposition trades one failure mode for another. Before decomposition, the orchestrator fails because its prompt is too crowded for the guardrails to hold. After decomposition, the system fails because no participant has the context to spot the cross-discipline violation. The pre-decomposition failure is visible at the seams: wrong shape of prompt, model thrashing. The post-decomposition failure ships through looking fine inside every individual discipline and breaks when the work hits production.

The "multi-agent" answer to complexity rearranges the problem without solving it. The cross-discipline rules end up owned by nothing: decomposition removed them from the orchestrator without giving them a new home. The work that actually has to happen is encoding those rules in something that holds them durably and surfaces them at the right decision points. That mechanism is the predefined workflow.

5. The Split That Works

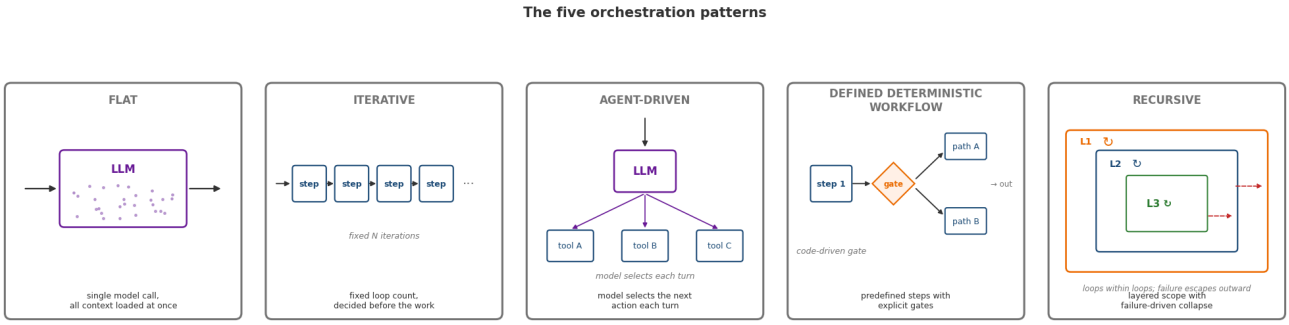

Figure 3: Five orchestration patterns for agent systems. This paper concerns the defined deterministic workflow pattern (sometimes called "predetermined stepwise"). The full comparative analysis of all five patterns is in the companion paper, "The Five Orchestration Patterns."

The language model does not belong in the orchestrator's seat. The orchestrator runs as a deterministic workflow. The model is invoked from inside the workflow, on narrow tasks, with curated context.

The work an agent does in a multi-discipline environment divides into four layers, each with a mechanism that performs it well.

Orchestration

Deciding what step happens next, in what order, under what conditions. This is where the cross-discipline state lives. It is also where reasoning is least required. A predefined workflow runs the same way every time, does not lose track of which constraints apply, and does not consume tokens to decide what to do next. The workflow is not sequential task automation and not a wrapper around the same model calls. It encodes the cross-discipline interaction points explicitly and checks them at the gates between steps.

Intent detection

Figuring out what the user actually wants from input that is often ambiguous. This is the part of the work language models perform best. The model gets a focused prompt, returns a structured answer, and exits.

Reasoning and decisioning

The steps inside the workflow where rules alone are not enough: a judgment call that depends on context, a synthesis across information that does not reduce to a query. The model is invoked here too, with focused context, returning a structured answer, exiting back into the workflow.

Action

Doing the actual work: making the change, calling the API, querying the system, comparing values, deploying the configuration. No judgment in it. Deterministic code handles this layer faster, cheaper, and more reliably than any model.

Figure 4: Orchestration and action are deterministic. Intent detection and reasoning are model-driven. The cross-discipline rules live in the deterministic layers, not in the model invocations.

Orchestration and action are deterministic. Intent detection and reasoning are model-driven. Cross the lines and the failure modes from Section 1 come back.

---

6. A Decision Rubric

Not every system needs this architecture. The split carries engineering cost. It is the right choice when the cost of the cascade exceeds the cost of building the workflow. Four variables determine where a given system sits.

Discipline count

How many independent bodies of knowledge does the task touch.

- **One:** an LLM-orchestrated agent will be fine.
- **Two:** probably fine.
- **Three or more:** the architecture in this paper starts paying for itself.
- **Four or more:** the alternative will not survive contact with production.

Rule interaction

Do the disciplines have rules that interact, or do they compose cleanly. Tasks where the disciplines are independent tolerate model orchestration well. Tasks where a decision in one discipline creates a constraint in another are where cross-discipline violations originate, and where deterministic orchestration becomes necessary.

Blast radius

What happens when the agent gets a step wrong. If the worst case is a re-run, model orchestration is acceptable. If the worst case is corrupted master data, a failed deployment, a misstated balance, or a regulatory issue, the architecture has to prevent that case from occurring.

Reversibility

Can the system unwind a bad step automatically. If a step can be reverted by the workflow itself, model orchestration buys speed at acceptable risk. If reversal requires manual remediation by the customer, the orchestration layer cannot afford to fail.

The rubric reduces to a single question. **Is the agent operating on a system where wrong is expensive.** If yes, the architecture in this paper is the one that ships. If no, lighter patterns are fine.

The workflow costs more to build than an LLM-orchestrator and less to operate by a margin that compounds with every run. The cascade keeps charging. The workflow charges once.

Most production enterprise agents operate on systems where wrong is expensive. Most are being sold with the lighter architecture. The gap between the two is the source of the negative-ROI rate the industry currently posts on AI deployments. Buyers who do not understand the distinction will keep funding the wrong architecture and call the result a technology problem. It is a design problem with a known answer.
