FlowX.AI

TECHNICAL PAPER

RELIABLE AGENT EXECUTION VIA LAYERED STATE-MACHINES (RAILS)

The Graph Is Deterministic, Even When the Nodes Are Not

How Agent Builder makes enterprise AI agents reliable by compiling visual workflows into deterministic state machines, in which the stochastic model is one bounded, typed node among many.

Bogdan Răduță
Head of Research, FlowX.AI
bogdan.raduta@flowx.ai

FlowX.AI · Applied Research

August 2026 · Version 1.1

ABSTRACT

The reliability problem with enterprise AI agents is usually misdiagnosed. The instinct is to blame the model: it is stochastic, so the agent is unpredictable. But the unpredictability that blocks production is rarely the model picking a slightly different word; it is the agent taking a different *path*, calling a tool it should not have, looping without end, or producing an output a downstream system cannot consume. That is a property of how the agent is orchestrated, not of the model's sampling. We describe **RAILS (Reliable Agent Execution via Layered State-machines)**, the execution architecture of FlowX.AI's Agent Builder, and argue a thesis in the spirit of containment engineering: *a workflow can be deterministic even when its nodes are not*. Agent Builder defines an agent as an explicit graph of typed nodes and edges, compiles it by topological sort into a LangGraph state machine of parallel execution phases[2], and runs it phase by phase with checkpointed, resumable state. Control flow is deterministic by construction: a Condition node branches on a Python expression, an Orchestrator routes via structured output into a fixed set of branches, an Aggregator merges parallel results by declared rules. The stochastic model is confined to specific node types, wrapped by middleware that caps calls, retries, falls back, and times out, and bracketed by guardrail and privacy nodes. The result is that the parts of the system that must be predictable, the control flow, the I/O contracts, the side effects, are predictable, while generation is placed deliberately and observed. We present the compilation pipeline, a determinism gradient over the full catalog of thirty-eight node types across eight categories, the containment middleware, the multi-agent patterns, and an illustrative regulated claims workflow, and we show why determinism is an architectural property to be engineered, not a model property to be hoped for.

KEYWORDS agent orchestration · determinism · state machines · LangGraph · workflow compilation · node catalog · containment middleware · resumable execution

1 Introduction

Run the same prompt through the same model twice and you may get two different sentences. Enterprises have made their peace with that; a paraphrase is not what keeps an agent out of production. What keeps it out is a different kind of variance: the agent that resolves a case one way on Monday and the opposite way on Tuesday, that calls a payment API it was never meant to touch, that loops until it exhausts a budget, or that returns a blob a downstream service cannot parse. None of these is a sampling artifact. They are consequences of *how the agent is orchestrated*, of letting a stochastic model decide, at runtime, what happens next.

The dominant pattern in agent frameworks invites exactly this. An autonomous loop hands the model a set of tools and lets it choose, step by step, which to call and when to stop[3]. That is powerful for open-ended exploration and genuinely unsuited to a regulated process, where the path a transaction takes is itself a controlled artifact: it must be the same every time, explainable after the fact, and bounded in what it can touch. Anthropic draws the distinction cleanly in its own guidance, separating *workflows*, where LLMs are orchestrated through predefined code paths, from *agents*, where the model dynamically directs its own process[5]. The reliability an enterprise needs lives on the workflow side of that line.

> *A workflow can be deterministic even when its nodes are not. You do not get reliable agents by making the model predictable; you get them by building a harness whose control flow, contracts, and side effects are predictable, and placing the model inside it as one bounded node.*
>
> RAILS · CORE THESIS

This paper describes how FlowX.AI's **Agent Builder** realizes that harness, an architecture we call **RAILS (Reliable Agent Execution via Layered State-machines)**. An agent in Agent Builder is not a free-running loop but an explicit graph: typed nodes connected by edges, authored visually or generated, and compiled into a deterministic state machine. The control flow is fixed and inspectable; the input and output of the whole agent are typed contracts; every node is a known type drawn from a bounded registry; state is checkpointed so a run can be replayed or resumed; and the stochastic model appears only inside specific node types, wrapped by middleware that limits, retries, and times out its behavior, and bracketed by guardrail and privacy nodes. Determinism here is not a claim that the model is deterministic. It is a property of the graph: the things that must be predictable are, and generation is confined, typed, and observed.

1.1 Contributions

- A reframing of agent reliability from a model property to an architectural one: the orchestration graph, not the model, is where determinism is won or lost.
- The RAILS compilation pipeline: how a visual node-and-edge workflow becomes a topologically-sorted, phase-parallel LangGraph state machine with checkpointed, resumable state.

- A determinism gradient over the full catalog of thirty-eight node types in eight categories, classifying each as deterministic, bounded, or generative, and showing how to compose them so nondeterminism is isolated.
- The containment middleware (thirteen addons plus per-node timeout and error policies) that bounds the failure modes of the stochastic node types.
- An illustrative regulated claims workflow demonstrating deterministic control flow, a confined model, and replayable, auditable execution.

2 Why Orchestration, Not the Model, Decides Reliability

2.1 Two ways to drive an agent

There are broadly two ways to make an LLM do multi-step work. In the *autonomous-loop* style, the model is given tools and a goal and decides at each turn what to do next, stopping when it judges the task done. In the *orchestrated-graph* style, a fixed structure decides what runs and in what order, and the model is invoked at specific points to do specific things. The first maximizes flexibility; the second maximizes predictability. Both are legitimate, and Agent Builder supports agentic nodes inside graphs, but for a regulated process the orchestrated graph is the only one whose path can be guaranteed, explained, and bounded, which is why it is the substrate rather than the exception.

2.2 What "deterministic" has to mean here

Determinism for an enterprise agent does not require that a generative node emit identical tokens every run; that is neither achievable nor necessary. It requires that the parts on which correctness, compliance, and integration depend are fixed: the *control flow* (which nodes run, in what order, under which conditions), the *contracts* (the typed input the agent accepts and the typed output it returns), and the *side effects* (which external systems are touched, and within what limits). A statechart formalism makes exactly this separation, fixed structure with well-defined transitions[1], and it is the formalism a compiled agent workflow approximates. RAILS is the application of that idea to LLM orchestration: pin the structure, type the boundaries, bound the effects, and let generation vary only inside a node whose output is then validated or consumed deterministically.

2.3 The cost of getting it wrong

An autonomous loop in a regulated setting fails in ways that are expensive and hard to detect: a path that diverges between two runs of the same case is an auditability failure; an unbounded tool choice is a security and cost failure; an output that does not match a schema is an integration failure that surfaces downstream. Table 1 contrasts the two postures against the properties an enterprise process actually requires; the rest of the paper shows how Agent Builder delivers the right-hand column.

Table 1. Autonomous agent loop versus a compiled orchestration graph.

Property	Autonomous loop	RAILS (compiled graph)
Control flow	Model decides at runtime	Fixed by the graph, compiled ahead
Same case, same path?	Not guaranteed	Guaranteed
I/O boundary	Free-form	Typed Start / End contracts
Side effects	Unbounded tool choice	Explicit nodes + call limits
Replay / resume	Rare	Checkpointed state, resumable
Where the model varies	Everywhere	Confined to specific node types

3 The RAILS Architecture

Agent Builder turns a visual workflow into a running state machine through a fixed compilation pipeline. The author draws nodes and edges; the platform compiles them; the compiled graph executes phase by phase with checkpointed state. Figure 1 shows the pipeline.

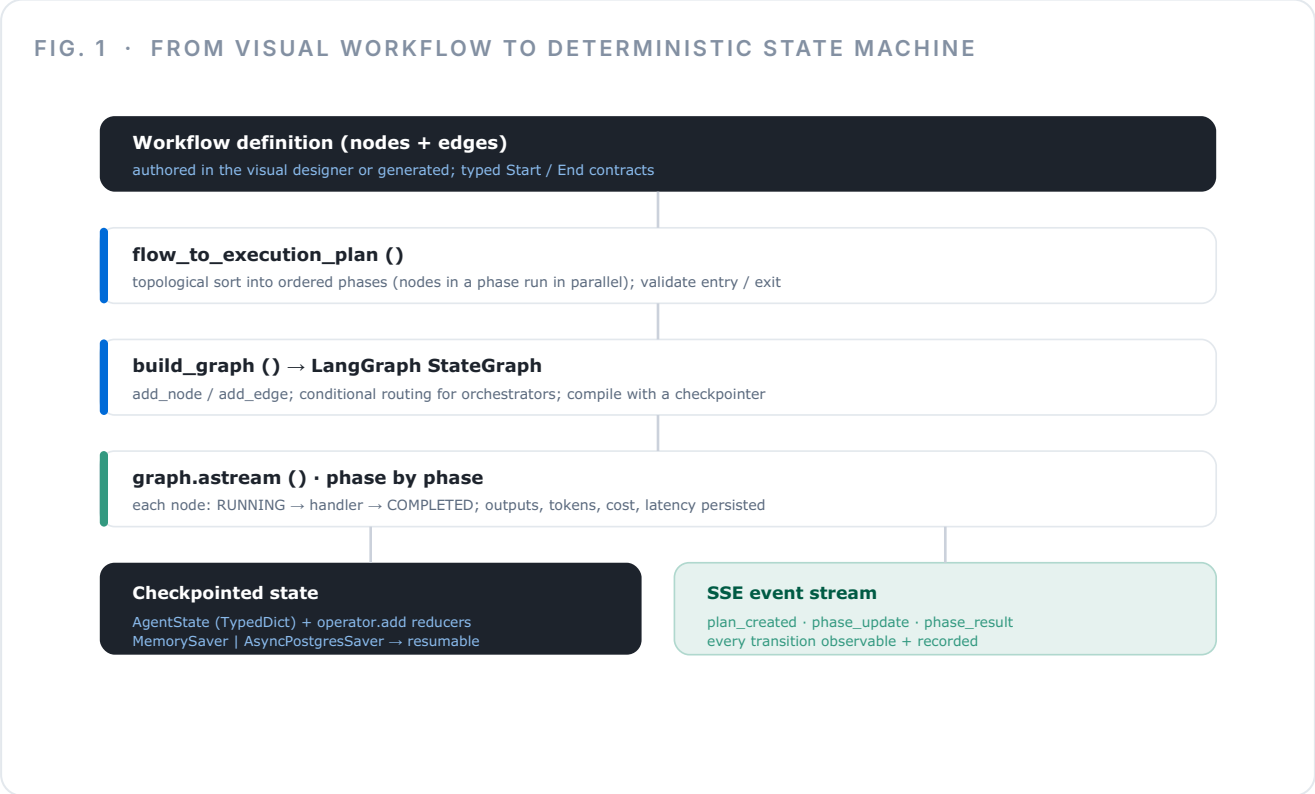


Figure 1. Compilation is deterministic and happens before execution. A topological sort fixes the order of phases (and which nodes run in parallel within a phase); the graph compiles to a LangGraph StateGraph with explicit conditional routing only where an Orchestrator chooses a branch; and execution streams every phase transition while persisting state to a checkpointer, so any run is observable, recorded, and resumable.

3.1 Workflows as explicit graphs

An agent is a JSON structure of nodes and edges. The graph is the program: it says which operations exist and how control passes between them, and nothing runs that is not a node on it. Two of those nodes are special. The **Start** node defines the workflow's input parameters, which is to say its API contract; the **End** node defines the output schema returned to the caller. Because the boundary is typed, the agent presents a stable interface to the systems around it regardless of what happens inside, and an integration cannot be broken by the model choosing to return something differently shaped.

3.2 Compilation to a state machine

Before anything executes, the workflow is compiled. A topological sort over the nodes and edges produces an ordered sequence of *phases*; nodes with no dependency between them land in the same phase and run in parallel, and the sort also validates that the graph has proper entry and exit points. The phases are assembled into a LangGraph *StateGraph*, with conditional routing inserted only where

an Orchestrator node selects among branches. The compiled graph is then run with a checkpointer attached. The important consequence is that the control flow is decided at compile time from the graph, not at run time by the model: the same workflow yields the same execution structure on every invocation.

3.3 Deterministic control flow

Branching, routing, and merging are handled by node types whose decision logic is explicit, not left to free generation. A **Condition** node evaluates a Python expression and takes its TRUE or FALSE edge, the ordinary if/else of the graph. An **Orchestrator** node does use the model to classify an input, but it is constrained to emit *structured output* that selects from a fixed, declared set of branches[4], and it supports a parallel fan-out mode; the model's influence is bounded to choosing among options the author defined, never to inventing a new path. An **Aggregator** node merges the results of parallel branches by a declared mode, concatenate all, take the first non-empty, let the model pick the best, or synthesize a combined answer, so even result-merging is a chosen rule rather than an emergent behavior. Control flow is therefore deterministic where it is purely logical and bounded where it is model-assisted.

3.4 Replayable state

Execution threads a single typed state object, an *AgentState*, through every node; its accumulator fields use append reducers so each node's output, logs, and token and cost metadata are added rather than overwritten, leaving a complete record of the run. That state is written to a checkpointer, an in-memory saver for single-shot calls or a PostgreSQL saver for multi-turn and resumable executions, and each phase is also persisted as a database record with its inputs, outputs, model, tokens, cost, and latency. A run can therefore be paused and resumed, and, just as importantly for a regulated setting, reconstructed exactly after the fact: what ran, in what order, on what input, producing what output, at what cost.

4 Placing the Stochastic Core: a Determinism Gradient

A compiled graph is only as reliable as the discipline with which generation is placed inside it. The node catalog spans a gradient from fully deterministic operations to fully generative ones, and good design uses the gradient deliberately: keep control flow and contracts on the deterministic end, push generation into clearly marked nodes, and convert generative output back to deterministic form (a schema, a branch, a validated field) as soon as possible. Figure 2 shows the gradient with representative nodes.

FIG. 2 · THE DETERMINISM GRADIENT OVER NODE TYPES

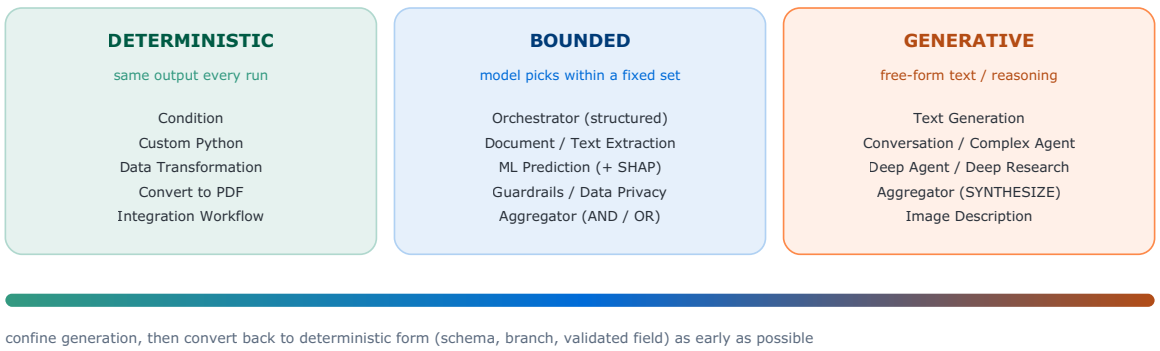


Figure 2. Node types fall on a gradient. Deterministic nodes return the same output for the same input; bounded nodes let the model choose within a fixed set or schema; generative nodes produce free-form content. The architecture does not remove the generative end, it lets a designer place it precisely and surround it with deterministic structure, so the unpredictable part of the system is small, marked, and wrapped.

This is the operational meaning of the thesis. A claims workflow can route deterministically on a Condition, classify with a bounded Orchestrator, extract fields against a schema with Document Extraction, and only then call a generative agent to draft a customer explanation, whose output is in turn passed through a Guardrails node before it leaves. The graph is mostly deterministic; the one genuinely generative node is boxed on both sides. Determinism is not all-or-nothing, it is a budget the designer allocates, and the catalog is what makes the allocation possible.

5 The Node Catalog

Agent Builder ships thirty-eight node types in eight categories, matching the Designer canvas toolbar. Each is a typed unit with a defined contract; the tables below give every node, what it does, and where to use it. Appendix A classifies them by determinism tier.

5.1 Basic

Node	Capability	Where to use
Start	Workflow entry point; defines input parameters and optional model selection	Every workflow begins here; defines the API contract
End	Workflow exit point; defines the output schema	Every workflow ends here; defines what is returned to the caller
Sticky Note	Documentation and annotation; no execution logic	Notes for teammates, documenting decisions, context on complex flows

5.2 Agents

Node	Capability	Where to use
Conversation Agent	Streaming chat agent (SSE) for interactive, context-aware dialogue with addon middleware	Chat flows, support bots, conversational interfaces with real-time streaming
Complex Agent	Multi-tool reasoning agent with iterative problem solving, web search, knowledge bases, MCP servers, SubAgents	Multi-step reasoning needing tools, RAG, or external integrations
SubAgent	Specialized sub-agent that executes as a tool inside a parent agent	Delegating subtasks (a "search" or "calculator" sub-agent) within a parent
A2A Agent	Agent-to-Agent protocol communication with external or internal agents	Distributed agent systems, calling specialized external agents
Orchestrator Agent	LLM routing via structured output; classifies input and routes to the correct branch; single or parallel fan-out	Routing requests to specialists (billing vs support vs sales), intent classification
Aggregator Agent	Merges parallel branch outputs; modes AND (concat), OR (first non-empty), BEST (LLM picks), SYNTHESIZE (LLM combines)	After parallel branches, to merge, choose, or synthesize a unified result
Deep Agent	Research-focused agent with iterative reasoning, web investigation, and file generation	In-depth research, report generation, multi-round search and analysis
Deep Research	Comprehensive topic research with optional advanced web crawling	Broad topic exploration, market research, competitive analysis
Browser Automation	AI-driven browser interactions; navigate, fill forms, extract data, proxy and screenshots	Scraping JS-rendered pages, automating form submissions, dynamic-site collection

5.3 Tools

Node	Capability	Where to use
Extract from Document	File-based text extraction from uploaded documents (PDF, DOCX, ...)	Pulling text from uploaded files for downstream processing
Extract from Web Page	Web scraping and content extraction with configurable crawl depth	Crawling product pages, scraping articles, building datasets
ML Prediction	ML inference with SHAP explainability via ModPod; LightGBM, scikit-learn, custom models	Propensity, risk, churn scoring, any ML inference inside a workflow
Convert to PDF	Convert text or content to PDF with optional headers and footers	Generating PDF reports and downloadable documents
Extract from Knowledge Base	Vector similarity search over indexed knowledge bases	RAG retrieval, FAQ lookup, context injection for agents
Add to Knowledge Base	Store and index content into vector databases	Ingesting documents and processed data into vector stores
Guardrails Filter	Content safety and compliance filter validating agent inputs/outputs against policies	Compliance on outputs, content moderation, off-topic prevention
Data Privacy	PII detection, redaction, and protection with configurable entities and language	GDPR compliance, anonymizing data before LLM processing, masking outputs
Integration Workflow	Call and compose existing integration workflows with parameters	Reusing workflows as building blocks, triggering external integrations
Condition	Conditional branching via Python expression; TRUE / FALSE edges	If/else gates, routing on a score, validity checks, business rules
Custom Python Code	Execute sandboxed Python with access to input data	Custom logic, data manipulation, API calls, calculations

5.4 Audio

Node	Capability	Where to use
Speech to Text	Audio transcription with multiple providers (document or CMS source)	Transcribing calls, meetings, voice messages for processing
Text to Speech	Audio generation with configurable voice, speed, and output location	Voice responses, audio content, accessibility audio

5.5 Text

Node	Capability	Where to use
Text Understanding	Extract and comprehend information from text with configurable operations	Sentiment, intent detection, summarization, key-point extraction
Text Generation	Create new text content from instructions and input	Content creation, email drafting, copywriting, responses
Text Transformation	Convert and transform text between formats and styles	Translation, tone adjustment, format conversion, rewriting
Text Extraction	Extract specific structured information from unstructured text	Entity extraction (names, dates, amounts), key-value parsing

5.6 Image

Node	Capability	Where to use
Image Description	Generate detailed text descriptions of images	Accessibility text, captioning, visual content cataloguing
Image Analysis	Analyze image content and extract insights with vision models	Visual inspection, defect detection, classification, moderation

5.7 Document

Node	Capability	Where to use
Document Generation	Create documents from data with configurable headers and footers	Formatted reports, proposals, letters from workflow data
Document Understanding	Analyze document content and extract information, with optional KB indexing	Contracts, policies, manuals: understanding meaning and auto-indexing
Document Extraction	Extract structured data from documents (PDF, images)	Invoices, forms, receipts into structured JSON (IDP pipelines)

5.8 Structured Data

Node	Capability	Where to use
Data Enrichment	Enhance existing data with LLM-derived information	Augmenting CRM records, derived fields, enriching datasets
Data Generation	Create synthetic data from input schemas and instructions	Test datasets, mock data, synthetic training data
Data Transformation	Format, structure, and reshape data between formats	Reshaping JSON, schema conversion, normalizing API responses
Textual Representation	Convert structured objects into human-readable text	Turning records into natural-language summaries and narratives

6 The Specialist Services Behind the Nodes

Several node types are deliberately thin: rather than ask a general language model to do heavy or sensitive work, they delegate to a dedicated microservice built for that one job. This is itself a determinism strategy. A purpose-built OCR-and-layout service reads a scanned table more predictably, more cheaply, and more auditably than prompting a vision model to "read the table," and a PII engine with declared entity types and strategies redacts more reliably than instructing a model to "remove personal data." Offloading to a specialist replaces an open-ended generative step with a bounded, versioned, independently testable one, which is exactly the move the determinism gradient of Section 4 rewards. The principal services are summarized in Table 2.

Table 2. Specialist services the graph delegates to, and the nodes they sit behind.

Service	Behind	What it does, and why it is more deterministic than prompting
Doc Parser (Docling)	Document Extraction / Understanding	Layout analysis, table extraction, signature and stamp detection, page-rotation correction, and per-page OCR confidence grading with a vision-model fallback below threshold; six selectable engines. A measured confidence, not a guess.
Doc Converter	Document nodes (pre-step)	LibreOffice-based conversion of arbitrary formats to PDF before parsing, so downstream extraction sees one normalized input.
Data Privacy (Presidio)	Data Privacy node, pii_detection addon	Detection and redaction of 25+ PII entity types (including locale-specific identifiers such as the Romanian CNP) with declared strategies (block / redact / mask / hash) and reversible de-anonymization. Policy-driven, not model discretion.
Prompt Shield	prompt_shield addon	ML-based prompt-injection detection via a semantic classifier, in block or warn mode. A dedicated detector, not the same model judging itself.
ModPod	ML Prediction node	ML model lifecycle and inference for LightGBM, scikit-learn, and custom models, with SHAP explainability. A trained model gives a reproducible score with attributions.
Speech (STT / TTS)	Speech to Text, Text to Speech	Transcription and synthesis across providers; a transcription is a bounded operation with a confidence, fed to deterministic downstream nodes.
Web Crawler	Extract from Web, Browser Automation	Content extraction and Playwright-driven interaction with configurable depth; isolates the messy, side-effecting web behind a bounded fetch step.
Dynamic Classifier	Classifiers	A self-improving document classifier (a cheap model with an LLM-judge feedback loop) that returns a label and a calibrated confidence the graph can branch on deterministically.

The pattern is consistent: where a task has a correct, checkable answer, the platform routes it to a specialist that returns a typed result with a confidence, and the graph then branches deterministically on that confidence rather than on free text. The general model is reserved for the work that genuinely needs open generation, and the services absorb everything that does not.

7 Containment Middleware

Confining the model to specific node types is necessary but not sufficient; those nodes still need their behavior bounded. Agent Builder wraps agent execution in pluggable middleware (addons) that hook the lifecycle around each model call, before and after the model, and before and after the agent, so that the nondeterministic step runs inside a cage of deterministic policy. Thirteen addons ship built in, and they fall into three jobs.

Bounding cost and failure. *model_call_limit* and *tool_call_limit* cap how many times a node may call the model or a tool, foreclosing runaway loops and cost spirals. *model_retry* and *tool_retry* re-attempt transient failures with backoff; *model_fallback* switches to an alternative model when the primary is unavailable. Combined with per-node timeout policies and node-level error handlers with configurable retry, these convert the open-ended failure modes of a generative call into bounded, policy-governed outcomes.

Safety and privacy. *pii_detection* anonymizes personal data before it reaches the model and can restore it after; *prompt_shield* detects and blocks prompt-injection attempts in block or warn mode; the Guardrails node filters outputs against policy. The stochastic core never sees raw PII it does not need, and its outputs are checked before they leave.

Context and continuity. *summarization* compresses history when the token budget is nearly full; *context_editing* trims stale tool output; *long_term_memory* persists facts across sessions; *llm_tool_selector* narrows the tool set before a call; *planner* tracks multi-step tasks; *human_handoff* escalates to a person. These keep a long-running agent coherent without surrendering control of its boundaries. Appendix B lists all thirteen with their hook points.

8 Composing Determinism with Agency

RAILS does not forbid agency; it frames it. Four patterns let a designer introduce as much model-directed behavior as a task warrants while keeping the surrounding structure fixed. **SubAgents** turn specialized agents into tools a parent invokes, so domain expertise is modular but the parent's control remains explicit. **Skills** inject domain knowledge and tools into a single agent from markdown documents, adding capability without adding agents. **Router (Orchestrator)** classifies input and dispatches to specialized branches, bounded as in Section 3.3 to a fixed branch set. **Custom Workflow** mixes deterministic logic and agentic nodes freely and can embed the other three as components. The selection rule is simple: use the least agency that solves the problem, and place whatever agency you do use inside the graph rather than in place of it.

9 Observability and Testing

Determinism pays its largest dividend in what it makes possible after the fact. Because every phase persists its inputs, outputs, model, tokens, cost, latency, and logs, and because the FLOWX Observatory callback captures every model call, tool invocation, and retrieval, an execution is fully reconstructable; per-node token and USD cost are tracked against model profiles, and third-party tracers (Langfuse, Opik) can run alongside. The same property makes agents *testable* in a way autonomous loops are not: Agent Builder maintains test datasets and test cases, runs them as test runs, and records per-case results, so a fixed-control-flow workflow can be regression-tested like ordinary software. An agent whose path is deterministic is an agent whose behavior can be pinned by a test, which is the precondition for changing it safely.

10 Illustrative Application

Consider a claims-intake workflow at an illustrative regulated insurer, Meridian Insurance. The configuration here is illustrative and chosen to show how the gradient is used in practice.

The graph begins at a **Start** node whose typed contract accepts a claim document and a policy reference, the agent's stable API. A **Document Extraction** node parses the document into structured fields against a schema (a bounded operation). A **Condition** node branches deterministically on the extraction's confidence: below threshold, the claim is routed to a human queue; above it, processing continues. An **Orchestrator** classifies the claim type via structured output into one of a fixed set of branches, motor, property, health, each leading to a specialist **Complex Agent** with the relevant knowledge base attached. The agent's drafted assessment passes through a **Data Privacy** node that masks personal data and a **Guardrails** node that checks the output against policy before an **End** node returns a typed result. Middleware caps the agent's model and tool calls and times each node out.

Every part of this that must be predictable is. The path a given claim takes is fixed by the graph and reproducible; the routing decision is a Python condition and a bounded classification, not free generation; the only genuinely generative step, the assessment draft, is boxed by privacy and guardrail nodes and bounded by call limits; and the entire run is checkpointed and recorded phase by phase, so an auditor can replay exactly what happened to any claim. The agent uses real model intelligence where it adds value, drafting an assessment from retrieved policy, while the insurer retains a process whose control flow, contracts, and side effects never depend on a sample from a distribution. That is determinism by construction, and it is what makes the workflow deployable in a regulated line of business at all.

11 Empirical Evaluation

Section 4's determinism gradient (Figure 2) is an architectural claim: a BOUNDED node (structured output selecting among a fixed set of branches, for example an Orchestrator) should route more reliably than a GENERATIVE node whose free text is parsed after the fact. We tested this directly with a standalone routing harness, run against two independent model providers, rather than taking the gradient on faith. The implementation and raw results are released as supplementary material.

11.1 Setup

Ten support-ticket routing decisions (four teams: billing, technical, account, other) were run under two node conditions at each provider's default sampling temperature (not fixed to zero, so any stability we observe is a property of the node's I/O contract, not of suppressed randomness): a **constrained** condition, a forced tool call whose schema restricts the response to one of the four category labels (the BOUNDED Orchestrator pattern), and a **freeform** condition, an open-ended "which team, and why" completion with the routing decision recovered afterward by naive keyword search (the GENERATIVE pattern this architecture argues against). Six tickets are unambiguous by construction; four were written to be genuinely boundary cases (for example, a ticket disputing both a duplicate charge and a locked account). Each of the 40 (ticket, provider, condition) cells was repeated 20 times, for 1,600 calls across Claude Haiku 4.5 and GPT-5.4-mini.

11.2 Structural validity: the guarantee that held perfectly

Across all 800 constrained-condition calls, **zero** returned a category outside the declared four-value enum: the schema-forced tool call cannot produce an invented path, by construction, regardless of model or ticket. This is exactly the guarantee Section 5's BOUNDED tier and Limitations bullet 4 ("bounded routing depends on structured output holding") claim, and it held with no exceptions in this sample. The freeform condition also produced zero unparseable responses in this sample, which is a fact worth reporting rather than omitting: with two capable, well-instructed models on a routing task, the "invented path" failure mode the architecture is designed against did not materialize empirically here. The structural point stands regardless, a freeform node has no contract preventing that failure, while the constrained node has one that cannot be violated, but the empirical gap between them was narrower than the architecture's simplest pitch would suggest, on this task, with these models.

11.3 Routing stability

Table 1 reports mean stability (the fraction of a ticket's 20 repeats agreeing with the modal decision) by provider and condition.

Table 1. Mean routing-decision stability across 10 tickets x 20 repeats, by provider and node condition.

Provider	Condition	Mean stability
Claude Haiku 4.5	Constrained (BOUNDED)	0.960
Claude Haiku 4.5	Freeform (GENERATIVE)	**0.995**
GPT-5.4-mini	Constrained (BOUNDED)	0.940
GPT-5.4-mini	Freeform (GENERATIVE)	0.945

Both conditions were highly stable for both models (0.94–1.00), and the constrained condition was not uniformly more stable than freeform in this sample, one provider was slightly more stable free-form than constrained. Instability was not uniform across tickets: it concentrated on the four tickets deliberately written as boundary cases, plus one nominally unambiguous ticket (a data-retention policy question) that one model split between “other” and “account” roughly evenly, evidently pulled toward “account” by the phrase “deleted accounts” despite the question being about policy, not a specific account. That case is instructive: it occurred under the *constrained* condition, and the schema did nothing to prevent it, because the schema guarantees a valid label, not a consistent judgment. This is precisely Limitations bullet 2, determinism is a design discipline, not an automatic guarantee, made concrete: structural validity and judgment stability are different properties, the architecture delivers the first unconditionally and the second only as well as the underlying decision is genuinely unambiguous.

11.4 What this evaluation does and does not show

This experiment isolates one BOUNDED node type against one GENERATIVE node type for a single routing decision; it does not exercise phase-parallel execution, checkpointed replay, containment middleware, or a full multi-node graph, none of which a routing-stability test can speak to. The result supports the architecture’s structural claim precisely (a schema-constrained decision cannot leave the declared set; an unconstrained one has no such guarantee, even though it did not fail here) and complicates its simplest rhetorical version (that constraining the node also makes the underlying judgment more consistent, which this data does not show). Both are worth knowing before deploying the pattern in a regulated line of business. The routing harness, the raw sweep, and a pytest suite that reproduces Table 1 are released as supplementary material.

12 Discussion

The reframing has a practical consequence for how teams build. The question stops being “how do we make the model reliable enough to trust with the whole task?” and becomes “how little of the task must actually be generative, and how tightly can we box that part?” The catalog and the gradient turn that into a design activity: most of a regulated workflow is deterministic plumbing, and the model earns its place at a few well-chosen nodes. This is the same containment philosophy that runs through the rest of the platform. Where the hallucination-control work argues that trustworthiness is a property of the harness around the model, RAILS argues that reliability is a property of the graph the model runs inside; the two are the same move applied to different failure modes.

RAILS is also the substrate the other platform capabilities assume. Node-level self-adaptation tunes individual nodes; evidence-graded ROI prices the workflows; governance evidences their behavior against policy; the build copilot constructs these graphs from natural language. All of them depend on the agent being a compiled, inspectable, replayable graph rather than an opaque loop. Determinism is not a constraint the platform tolerates; it is the foundation the rest of it is built on.

13 Limitations

- **Generative nodes remain nondeterministic.** RAILS fixes control flow, contracts, and effects; it does not make a Text Generation node emit identical prose across runs. Where exact output reproducibility matters, that node must be followed by deterministic validation or extraction.
- **Determinism is a design discipline, not an automatic guarantee.** A designer can still route on a generative node's free text or leave an agent's tools uncapped. The architecture makes the deterministic design expressible and easy; it does not forbid the undisciplined one.
- **Parallel phases require care.** Nodes that run in the same phase must be genuinely independent; a hidden dependency between them is a correctness risk the topological sort cannot detect on the author's behalf.
- **Bounded routing depends on structured output holding.** An Orchestrator's safety rests on the model returning valid structured output selecting a declared branch; malformed output must fall back to a default branch rather than an invented path. Section 11 found this held with zero exceptions across 800 calls on two providers, but did *not* find that the schema improves the underlying judgment's consistency on genuinely ambiguous inputs.
- **Replay reconstructs, it does not re-execute identically.** Checkpointed state lets a run be reconstructed and resumed, but re-running a generative node yields fresh generation; audit relies on the recorded outputs, not on deterministic re-execution of the model.

14 Conclusion

Enterprises do not fail to deploy agents because models paraphrase; they fail because an autonomously-driven agent cannot promise the same path, the same contract, and the same bounded effects on every run, which is precisely what a regulated process demands. Agent Builder answers this by making the agent a compiled state machine rather than a loop: an explicit graph of typed nodes, sorted into phases and executed with checkpointed, resumable state; control flow that branches on code and routes through bounded structured choices; a catalog whose nodes span a determinism gradient so generation can be placed precisely and wrapped; and middleware that cages the stochastic node types in cost, safety, and timeout policy. The model still does the work only a model can do, but it does it as one node among many inside a harness that is deterministic by construction. A workflow can be deterministic even when its nodes are not, and that, not a more obedient model, is what makes enterprise AI agents reliable enough to ship.

References

[1] Harel, D. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8(3):231–274, 1987.

[2] LangChain, Inc. *LangGraph: Low-Level Orchestration for Stateful, Multi-Actor LLM Applications.* github.com/langchain-ai/langgraph (accessed 2026).

[3] Yao, S., Zhao, J., Yu, D., et al. *ReAct: Synergizing Reasoning and Acting in Language Models.* Proc. ICLR 2023.

[4] Willard, B. T., & Louf, R. *Efficient Guided Generation for Large Language Models.* arXiv:2307.09702, 2023. (Structured / schema-guided decoding.)

[5] Anthropic. *Building Effective Agents.* Anthropic Engineering, 2025. (Distinguishes workflows, predefined code paths, from agents, dynamic self-direction.)

[6] Object Management Group. *Business Process Model and Notation (BPMN) Version 2.0.* OMG, 2011.

[7] van der Aalst, W. M. P. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2004.

[8] Lewis, P., Perez, E., Piktus, A., et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* Advances in Neural Information Processing Systems (NeurIPS) 33, 2020.

[9] Lundberg, S. M., & Lee, S.-I. *A Unified Approach to Interpreting Model Predictions (SHAP).* Advances in Neural Information Processing Systems (NeurIPS) 30, 2017.

[10] Microsoft. *Presidio: Context-Aware, Customizable PII Detection and De-identification.* github.com/microsoft/presidio (accessed 2026).

[11] National Institute of Standards and Technology. *Artificial Intelligence Risk Management Framework (AI RMF 1.0).* NIST AI 100-1, 2023.

[12] European Parliament and Council. *Regulation (EU) 2024/1689 (Artificial Intelligence Act).* Official Journal of the European Union, 2024.

[13] International Organization for Standardization. *ISO/IEC 42001:2023, Information technology, Artificial intelligence, Management system.* ISO, 2023.

Appendix A Node Determinism Tiers

The same catalog, classified by how much the model influences a node's output. Design keeps control flow and contracts in the deterministic tier and converts bounded and generative output back to deterministic form as early as possible.

Tier	Rule	Nodes
Deterministic	Same input gives the same output; no model in the path	Start, End, Sticky Note, Condition, Custom Python, Data Transformation, Convert to PDF, Add / Extract Knowledge Base, Integration Workflow, Extract from Document / Web
Bounded	Model chooses within a fixed set or schema, or output is validated	Orchestrator (structured route), Aggregator (AND / OR / BEST), Document Extraction, Text Extraction, Document Understanding, ML Prediction, Guardrails, Data Privacy, Image Analysis, Speech to Text, Data Enrichment
Generative	Free-form content or open reasoning	Text Generation / Transformation / Understanding, Conversation Agent, Complex Agent, SubAgent, A2A Agent, Deep Agent, Deep Research, Browser Automation, Aggregator (SYNTHESIZE), Image Description, Document Generation, Data Generation, Textual Representation, Text to Speech

Tiering is by typical use; some nodes shift tier with configuration (for example, a generative node constrained to structured output behaves as bounded).

Appendix B Containment Middleware Reference

Thirteen built-in addons hook the agent lifecycle. Each wraps the nondeterministic model call in deterministic policy.

Addon	Hook	Purpose
model_call_limit	Before Model	Cap total model calls to prevent runaway cost
tool_call_limit	Before Model	Cap tool executions
model_retry	Before Model	Retry failed model calls with backoff
tool_retry	After Model	Retry failed tool calls with backoff
model_fallback	Before Model	Fall back to an alternative model on failure
pii_detection	Before & After Model	Anonymize PII before the model; restore after
prompt_shield	Before & After Model	Detect and block prompt injection (block / warn)
summarization	Before Model	Summarize history when the token budget is nearly full
context_editing	Before Model	Trim or clear stale tool outputs
long_term_memory	Before Model & After Agent	Persist and retrieve facts across sessions
llm_tool_selector	Before Model	Select the most relevant tools before the call
planner	Before Agent	Plan and track multi-step tasks
human_handoff	After Model	Hand the conversation to a human operator

Appendix C Worked Execution Plan

How the case-study graph of Section 9 compiles. The topological sort fixes the phases before any node runs; nodes in the same phase are independent and run in parallel.

Phase 0. *Start* validates the typed input contract (claim document + policy reference).

Phase 1. *Document Extraction* parses the document to a schema (bounded).

Phase 2. *Condition* evaluates the extraction confidence (deterministic): below threshold routes to a human queue and the plan ends; otherwise it continues.

Phase 3. *Orchestrator* classifies the claim type via structured output into exactly one declared branch, motor / property / health (bounded routing).

Phase 4. The selected *Complex Agent* drafts an assessment using its attached knowledge base (generative), wrapped by model and tool call limits and a node timeout.

Phase 5. *Data Privacy* masks personal data, then *Guardrails* validates the draft against policy (bounded).

Phase 6. *End* returns the typed result. Every phase is recorded with inputs, outputs, model, tokens, cost, and latency, so the run replays exactly.

FlowX.AI

FlowX.AI builds and orchestrates enterprise-grade AI agents for regulated industries, with a focus on banking, financial services, and insurance. Agent Builder compiles visual agent workflows into deterministic LangGraph state machines, so that institutions can deploy autonomous capabilities whose control flow, contracts, and side effects are predictable, observable, and auditable by construction.

FlowX.AI Technical Paper · RAILS v1.1 · Bogdan Răduță, Head of Research · bogdan.raduta@flowx.ai