RO: Reliable Orchestration of LLM Programs via Deterministic-Generative Separation

Saurabh Jha
IBM

***Abstract*—Failure analyses of multi-agent LLM systems show that the dominant failure mode is orchestration, not reasoning: across seven frameworks and 1,600+ execution traces, plan abandonment, state corruption, and tool-management errors dominate. We trace this to two coupled choices in current stacks: (i) the LLM is asked to perform deterministic bookkeeping (loops, state, control flow) and non-deterministic reasoning at the same time, and (ii) no program exists before execution—the LLM emits the next step turn by turn, so non-determinism leaks into *control*, not just data. We present RO (Reliable Orchestration), a stored-program runtime for LLM workflows. A typed program P exists before execution and is built from three node families: control & state, *deterministic* `goal` nodes (sandboxed code or MCP tools, auto-fulfilled by the runtime), and *generative* `goal` nodes (LLM oracle, bounded by an instruct–validate–repair (IVR) pattern with schema gating). The central invariant is precise: *orchestration correctness is independent of LLM behavior*. We show that the diagnostic controller for ITBench [1]—originally a 12,518-line Rust implementation—can be expressed as a 444-line RO program (a $28\times$ reduction), inheriting its prior $7\times$ Majority@k F1 gain over ReAct [2] while achieving 2–$3\times$ wall-clock speedup from automatic concurrency.**

***Index Terms*—large language models, program execution, agent orchestration, type safety, reliability**

I. Introduction

LLM-powered agents excel on bounded semantic tasks—classification, summarization, question answering—where context fits in a single model window and tool interfaces are well-defined [3], [4]. Production deployments, however, reveal a reliability gap rooted not in reasoning failures but in an *architectural conflation*: current frameworks require the LLM to perform two fundamentally different jobs at once.

Reasoning: classification, analysis, generation—tasks requiring semantic understanding where LLMs excel. **Orchestration**: loop management, state tracking, control flow, data structure operations—bookkeeping tasks where LLMs consistently degrade.

Figure 1 sketches the architecture: a four-component runtime—coordinator, control & dispatch, typed memory, repair agent—and a uniform per-node state machine that pulls the LLM in only at `goal` boundaries through a typed schema gate. Programs are built from three node families—control & state, *deterministic* `goal` (sandbox/MCP tool, auto-fulfilled), and *generative* `goal` (LLM oracle, untrusted)—and on a routine pipeline the LLM is invoked only at the generative goals; everything else executes reproducibly in the trusted zone.

A. Orchestration dominates, and it is deterministic.

Across production agent systems, orchestration concerns dominate the operation mix:

Context rot in extended sessions. Claude Code [5] and Codex CLI [6] sessions degrade as orchestration state (tool histories, intermediate variables, loop counters) accumulates alongside reasoning in the LLM context [7]. The reasoning is not getting harder—bookkeeping is crowding it out.

Controller failures in agent loops. The MAST taxonomy [8], derived from 1,600+ annotated execution traces across seven multi-agent frameworks, finds that system design failures—plan abandonment, tool repetition, state management errors—dominate over reasoning errors. The LLM fails not at semantic tasks but at managing its own execution state.

Persistent error rates prevent scale-up. MAKER [9] shows that even a 1% per-step error rate makes million-step tasks infeasible without error correction. Their solution—extreme decomposition into microagents with multi-agent voting—reaches zero errors but spends multiple LLM calls per step to correct what are fundamentally orchestration failures.

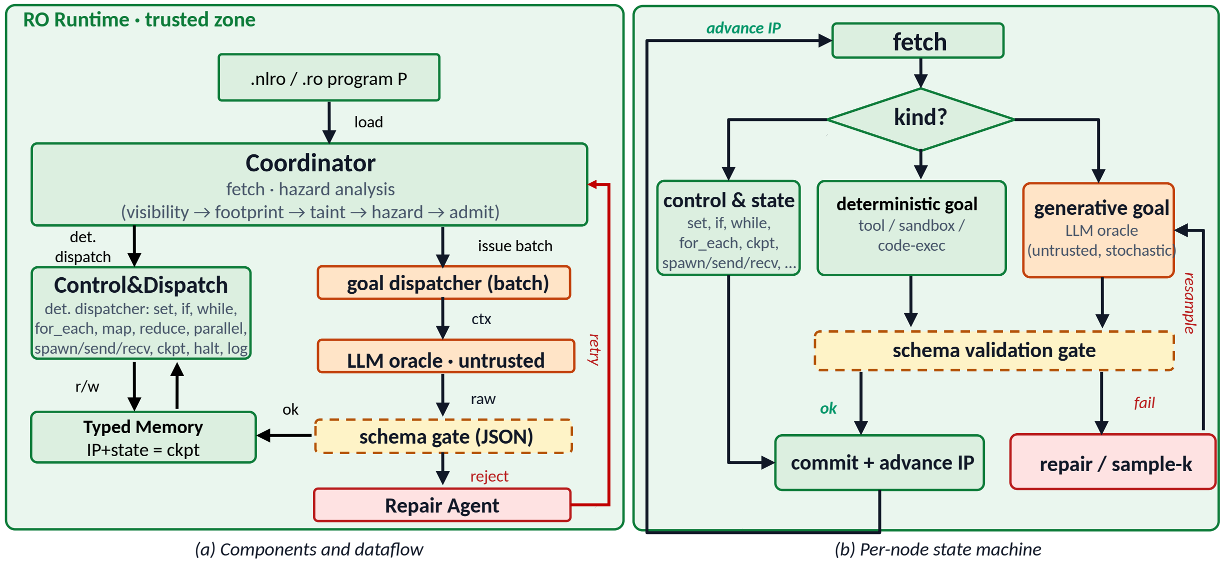
The common thread: orchestration dominates the operation mix and is deterministic in nature. RO eliminates orchestration errors structurally rather than statistically: deterministic execution for state and control, LLM invocation only for reasoning.

B. Two execution models for LLM workflows

The deeper question is *where the program lives*. Today's agent stacks and RO answer it differently.

Incremental synthesis (today's agents). No program exists before execution. The LLM receives a task, emits the next code or tool call, the runtime executes it, results return, and the cycle repeats. The "program" is the running concatenation of LLM-emitted fragments; control flow and termination are decided turn by turn. Non-determinism is in *control*: different LLM responses yield different programs with different iteration and termination. This produces a characteristic failure mode—the agent either terminates too early ("I am done") or too late ("gets derailed")—because there is no program-level termination condition to enforce.

Stored-program with oracle I/O (RO). A typed program P exists before execution—authored in natural language, compiled to a structured IR. The runtime sequences P deterministically: fetch, dispatch, advance. The LLM is consulted only at `GOAL` nodes, through a typed schema contract. Non-determinism is confined to *data*: given any fixed sequence of `GOAL` responses, the trace is fully determined; the LLM cannot



(a) Components and dataflow *(b) Per-node state machine*

Fig. 1. RO architecture. **(a)** Components and dataflow. **(b)** Per-node state machine. See §IV for details.

change what runs next, in what order, or whether the program terminates.

This is the model of Turing's oracle machine [10], modernized: a deterministic substrate that consults an external, untrusted oracle through a typed query, where the oracle contributes content but the deterministic machine retains control. The stored-program architecture [11] is the engineering analogue—the program is data the runtime can analyze before any oracle query. Because P exists as data, termination, resource bounds, and parallelizable subgraphs are answerable before any LLM call; an IP plus typed memory is a serializable snapshot (conversation history is not); typed schemas are composition interfaces; and dependency analysis enables parallelism, prefetching, batching, and caching.

Table I places RO against recent systems that share this concern. Three properties motivate the architecture and guide the rest of the paper:

Expressivity. As LLM automation moves from developer tooling into enterprise operations, workflow authors are increasingly domain experts—SREs, compliance analysts, clinicians—rather than programmers. They express intent in English and expect the system to handle execution; engineers need an inspectable representation to audit and tune. RO accepts `.nlro` (natural language) and compiles it to `.ro` (typed IR).

Separation of concerns. Entangling orchestration with reasoning degrades both. Jha et al. [2] demonstrated this on diagnostic tasks: separating deterministic control flow from an LLM reasoning policy yielded a 7× gain in Majority@k F1 over ReAct baselines. But that controller—the ITBench diagnostic agent [2]—is a bespoke Rust implementation, 12,518 LOC, hard-coding control flow, state, checkpointing, and MCP bindings for a single workflow. RO turns the separation principle into infrastructure: state, control flow, checkpointing, scheduling, and schema validation become runtime services,

TABLE I
PUTTING RO IN PERSPECTIVE.

	Nightjar [12]	**Mellea** [13]	**MAKER** [9]	**RO**
Runtime	CPython	CPython	LLM voting harness	**Workflow-aware**: hazard, batch, replay
State	Shared Python	Python + struct. out.	Implicit (per-step)	Typed memory + IP
LLM boundary	Inline NL blocks	`@generative` slots	Per-subtask agent	`goal` (det.+gen.)
Validation & repair	Effect/handler typing	IVR (schema, sample-k)	Multi-agent voting	IVR (schema, sample-k)
Authoring	Python	Python	Prompt per agent	NL (→ IR)
Prog. as data	✗	✗	✗	✓

so the equivalent controller reduces to 444 lines of `.nlro`—a 28× reduction (§V).

Reliability. "Reliable" here is a precise property: *orchestration correctness is independent of LLM behavior*. The runtime's control flow, memory integrity, and termination guarantees hold for any `GOAL` response—hallucinations, refusals, malformed outputs. Reasoning errors are bounded at the schema boundary and retried under a budget (§IV). Audit trails and checkpoint/resume make execution *replayable from logs* given the same `GOAL` responses.

RO and these systems are complementary rather than competing. Nightjar [12] and Mellea [13] embed LLM calls in Python: a general-purpose runtime that is deterministic but not workflow-aware. Mellea and RO share the same instruct–validate–repair (IVR) pattern at the LLM boundary; what differs is the runtime around it. Because RO's IR distinguishes deterministic from generative goals, the runtime can hazard-analyze, batch, auto-fulfill, and replay oracle calls—services CPython cannot provide because it sees only opaque function calls. MAKER [9] corrects orchestration errors statistically via

voting; RO eliminates the orchestration class of errors structurally and bounds reasoning errors at the schema boundary.

II. The Trust Boundary

The mapping to the stored-program model is direct. Memory, control flow, scheduling, and dispatch live inside the runtime—the *trusted zone* (green region of Figure 1, left). The LLM lives outside as an external oracle—the *untrusted zone*—reachable only at `GOAL` boundaries. `GOAL` is the only opcode without a Von Neumann analogue: the ALU is deterministic, the oracle is stochastic. To safely integrate a stochastic component into a deterministic substrate, RO applies an instruct–validate–repair (IVR) pattern at every `GOAL` [13]: the oracle's response is validated against a typed schema (JSON Schema [14]) and a rejection triggers sample-k repair before any commit. *The damage a stochastic component can inflict on surrounding execution is bounded by the IVR contract.*

This yields one operational invariant: LLM outputs never directly mutate control flow, the instruction pointer, or memory layout. They (1) arrive as structured data, (2) pass schema validation, and (3) enter typed memory as values consumed by subsequent deterministic instructions. Two flavors of `GOAL` sit on opposite sides of this boundary: a *deterministic* `GOAL` (sandboxed code or MCP tool) runs inside the trusted zone—reproducible, so the runtime auto-fulfills it without external round-trip; a *generative* `GOAL` (LLM oracle) runs outside—stochastic and adversarial-by-default, so the schema gate is the load-bearing contract and the sample-k/repair loop exists only for this path. When a `GOAL` returns generated code that a downstream node delegates to a sandbox or MCP tool, the trust boundary shifts to that sandbox; RO guarantees only that *which code runs, in what order, with what inputs* stays deterministic—analogous to an OS scheduling processes deterministically while delegating isolation to the process layer.

III. The RO Language

RO accepts two program formats. **Natural language** (`.nlro`): English-like specifications (e.g., "for each review, ask the AI to analyze sentiment; return sentiment and confidence") compiled once, ahead of execution, by a frontier LLM. The LLM has no runtime role except at generative-`GOAL` boundaries. **Typed IR** (`.ro`): a structured node language whose nodes fall into three families. **(i) Control & state**—typed declarations, `set`, collection ops, `if`/`while`/`for_each`/`match`/`try`, aggregates `map`/`reduce`/`parallel`, actors `spawn`/`send`/`recv`, `checkpoint`, `halt`—runs against typed memory and never leaves the runtime. **(ii) Deterministic `goal`**—a sandboxed code execution (`ro_code_exec`) or MCP tool call—is fulfilled in-runtime and gated through the same IVR check; reproducible-by-construction, so its failure mode is a typed tool error, not a hallucination. **(iii) Generative `goal`**—the single interface to non-deterministic reasoning: `goal "description" -> result:{schema} given: {context} using: {tools}`—emits a `GoalRequest` to the LLM oracle and gates the response.

The validate step uses JSON Schema [14] (e.g., `{sentiment:enum[pos,neg,neu], confidence:[0,1]}`); responses that fail the contract are rejected and surfaced as a typed event. For generative goals, the harness resamples up to k times following Mellea's IVR pattern [13]; persistent failure escalates to the repair path (§IV). Deterministic goals skip the sample-k loop: a tool error is returned as a typed `result` field. The validate step itself is mechanism-agnostic: constrained decoding [15], rewrite models, or repeated sampling [16] all satisfy the contract.

Concretely, the ITBench diagnostic agent [2] is 444 lines of `.ro`: 17 *generative*-`goal` nodes (the LLM-bound reasoning steps), a handful of *deterministic*-`goal` calls into Kubernetes, Prometheus, and log-fetch MCP tools, and $\sim$420 control & state statements managing the iterative graph exploration and convergence test. *The LLM is invoked only for reasoning steps; everything else is runtime.*

IV. Runtime Architecture

The runtime is organized as four cooperating components (Figure 1, left); execution at every node follows a uniform state machine (Figure 1, right).

Coordinator. Walks the typed IR, resolves expressions against typed memory, and dispatches each node. Before dispatching a candidate `GOAL`, it runs a five-pass dependency analysis—visibility, footprint, taint, hazard, admission—to identify frontier `GOAL`s whose footprints are disjoint from in-flight and pending writes. Disjoint `GOAL`s are emitted as a batch and resolve concurrently; results are joined in declaration order. `parallel`, `map`, and `spawn` expose the same machinery as language-level constructs.

Control & Dispatch. Two dispatchers share this layer. The *deterministic dispatcher* executes control & state node kinds (`set`, `enqueue`, `if`, `while`, `for_each`, `map`, `reduce`, `checkpoint`, `halt`, `log`, actor primitives) directly against typed memory. The *GOAL dispatcher* assembles the bounded `given` context and dispatches by kind: deterministic goals are auto-fulfilled in-runtime via the configured sandbox or MCP tool with no client round-trip; generative goals emit a typed `GoalRequest` and await a `GoalResult`. Both are gated through the same schema validation before commit.

Typed memory. State is declared with structural types: scalars, collections (`queue`, `stack`, `set`, `dict`, `list`), and records—no untyped "context blobs." All read/write traffic is mediated by the coordinator. Memory plus the instruction pointer is the entire serializable state; `checkpoint` writes both atomically.

Repair agent. On schema rejection, GOAL refusal, or persistent timeout, the runtime keeps the `GoalRequest` in-flight and assembles a `RepairContext` (goal description, resolved `given`, schema, rejected output and reason, retries remaining, surrounding program slice). The harness then (i) resamples under a constraint hint, (ii) mutates `given` via `update_memory`, or (iii) rewrites the program slice and

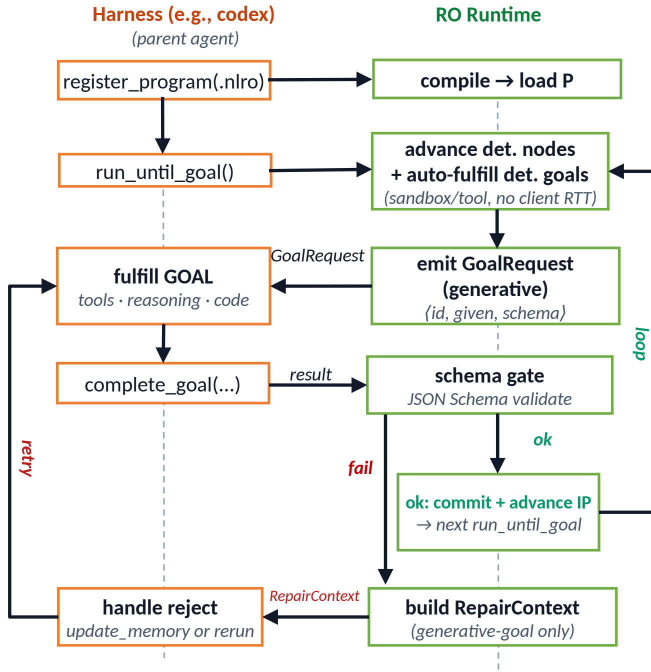


Fig. 2. RO in MCP mode. A harness (e.g., Claude Code, Codex CLI) drives the loop via four tools: `register_program` loads P once; each `run_until_goal` advances the runtime deterministically (control & state plus auto-fulfilled deterministic goals—no client round-trip) until a `GoalRequest` or halt. Only *generative* goals reach the harness, which returns a typed result via `complete_goal`; the schema gate either commits (ok) or returns a `RepairContext`. RO owns program, memory, and schedule; the harness owns reasoning at each generative `GOAL`.

resumes from the prior `checkpoint`. Programs are data: the same LLM that fulfills `GOAL`s can edit the program that issued them.

Error modes. Generative-goal schema violation: rejected at the gate, sample-k retry; the program never sees the bad value. Schema-valid but semantically wrong: caught downstream by program invariants. GOAL refusal/timeout: one retry, then escalates to repair. Tool error (deterministic goal or inside `using`): returned as a typed `result` field, no sample-k loop.

Deployment and replay. The same engine runs standalone or as an MCP server [17] (Figure 2): a harness calls `register_program` once, then alternates `run_until_goal` and `complete_goal`. During `run_until_goal`, the runtime advances all control & state nodes *and* auto-fulfills deterministic goals via the configured sandbox/tool—no client round-trip. Only generative goals emit a `GoalRequest` the harness must fulfill via `complete_goal`. Every `GOAL` writes an audit record; a recorded trace replays deterministically when generative-`GOAL` responses are sourced from the log.

Extension to Python: ROC. Drawing on Recovery-Oriented Computing [18], `rolib` (a Python library) turns exceptions into `GOAL` boundaries via a `@generative` decorator: frame state is captured, exposed via MCP, corrective code runs, execution resumes. Exception–remediation pairs collected in staging compile into deterministic production policies.

V. Demonstration

A. Re-hosting the ITBench diagnostic agent

The ITBench diagnostic agent [2] is an iterative LLM-driven algorithm that explores the IT resource graph to localize the root cause of an incident. The Rust implementation is 12,518 LOC across 19 files. The RO version is 444 lines in a single `.ro` file—a $28\times$ reduction: 17 *generative*-`goal` nodes for the LLM-bound reasoning, a handful of *deterministic*-`goal` calls into Kubernetes, Prometheus, and log-fetch MCP tools, and $\sim$420 control & state statements for the iterative graph exploration and convergence test. State, scheduling, schema validation, and audit logging are runtime services, not application code.

Reliability inherited from separation. On ITBench [1] the RO program reproduces the Rust implementation's diagnostic outcomes. Jha et al. [2] report a $7\times$ Majority@k F1 improvement over a ReAct baseline on the same algorithm; RO does not re-prove that result but generalizes the mechanism into a runtime service, so future controllers in the family no longer need a bespoke implementation.

Concurrency. The Rust controller dispatches `GOAL`s serially; RO's hazard analysis (§IV) emits disjoint `GOAL`s as a batch, yielding 2–$3\times$ wall-clock speedup at no cost to determinism.

B. Towers of Hanoi without voting

MAKER [9] solves million-step Towers of Hanoi via k-ahead voting because each microagent must track peg state, enforce constraints, choose the next move, *and* format output. In RO, state (three `stacks`), constraints (an `if`), and schema are runtime concerns; one `goal` per step asks only "which move?". For 20 disks ($\approx 10^6$ steps), RO issues $\sim 10^6$ calls; MAKER needs $\sim k \cdot 10^6$ to repair orchestration errors RO never makes. Hanoi is closed-form—this isolates the cost of conflating orchestration with reasoning, not a benchmark claim.

Future work. Two extensions close the loop between authoring and operation. `ro plan` introduces an interactive planner that co-develops the `.nlro` program with a human during *development*, pinning decisions into the IR rather than re-deriving them in every run. `ro learn` closes the *operational* loop: failure traces from the schema gate, retry exhaustion, or human feedback are folded back into the program's prompts, schemas, and tool sets to reduce semantic derailment on the next run. Together they turn a stored RO program into a living artifact—authored once, continuously hardened against the failure modes it encounters.

References

[1] S. Jha, R. R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen, J. Clark, B. Bhavya, M. Verma, H. Kumar, H. Kitahara *et al.*, "ITBench: Evaluating AI agents across diverse real-world IT automation tasks," in *Forty-second International Conference on Machine Learning*, 2025.

[2] S. Jha, R. Arora, Bhavya, N. Zheutlin, P. Toro Isaza, L. Shwartz, Y. Deng, D. Sow, R. Mahindru, and R. Puri, "Think locally, explain globally: Graph-guided LLM investigations via local reasoning and belief propagation," *arXiv preprint arXiv:2601.17915*, 2026.

[3] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2023.

[4] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[5] Anthropic, "Claude code," https://docs.anthropic.com/en/docs/claude-code, 2024.

[6] OpenAI, "Codex CLI: A lightweight coding agent for your terminal," https://github.com/openai/codex, 2025.

[7] K. Hong, A. Troynikov, and J. Huber, "Context rot: How context degradation affects LLM performance," https://research.trychroma.com/context-rot, 2025.

[8] M. Cemri, M. Gupta, R. Paleja, S. Chaudhuri, and U. Topcu, "Why do multi-agent LLM systems fail?" *arXiv preprint arXiv:2503.13657*, 2025.

[9] E. Meyerson, G. Paolo, R. Dailey, H. Shahrzad, O. Francon, C. F. Hayes, X. Qiu, B. Hodjat, and R. Miikkulainen, "Solving a million-step LLM task with zero errors," *arXiv preprint arXiv:2511.09030*, 2025.

[10] A. M. Turing, "Systems of logic based on ordinals," *Proceedings of the London Mathematical Society*, vol. 45, pp. 161–228, 1939, introduces oracle machines (§4).

[11] J. Von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993, originally circulated 1945.

[12] E. Y. Cheng, L. Weber, T. Jin, and M. Carbin, "Sharing state between prompts and programs," *arXiv preprint arXiv:2512.14805*, 2025.

[13] IBM Research, "Mellea: A library for writing generative programs," https://github.com/generative-computing/mellea, 2025.

[14] A. Wright, H. Andrews, B. Hutton, and G. Dennis, "JSON Schema: A media type for describing JSON documents," IETF Internet-Draft, https://json-schema.org/, 2022.

[15] B. T. Willard and R. Louf, "Efficient guided generation for large language models," *arXiv preprint arXiv:2307.09702*, 2023.

[16] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini, "Large language monkeys: Scaling inference compute with repeated sampling," *arXiv preprint arXiv:2407.21787*, 2024.

[17] Anthropic, "Model context protocol," https://modelcontextprotocol.io, 2024, accessed: 2024.

[18] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kıcıman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies," UC Berkeley, Tech. Rep. UCB/CSD-02-1175, 2002.