When the Model Was Not the Problem: Lessons from a Failed AI Workflow Orchestration Pilot

Ashraf Elnashar[1], Jules White[2], Douglas C. Schmidt[3]
[1]Nexoviq AI, Irvine, CA, USA, [2]Vanderbilt University, Nashville, TN, USA, [3]William & Mary, Williamsburg, VA, USA
ashraf.elnashar@nexoviq.ai, jules.white@vanderbilt.edu, dcschmidt@wm.edu

Abstract

Production AI workflow systems increasingly translate open-ended user intent into executable plans spanning models, tools, APIs, and enterprise services. Their reliability, however, depends not only on planning quality, but also on the orchestration boundary where generated plans become executable artifacts. This paper presents an industry study of a gated real-API pilot of a dynamic intent-to-workflow orchestration system for software-engineering automation. Across 50 intents, the full system achieved 80.63% structural similarity to expert workflows, 88% component-selection accuracy, and 92% of manual-workflow output quality in controlled benchmarking.

The central contribution of our study is not that the real-API pilot failed, but that we nearly acted on the wrong root-cause diagnosis. Across 1,050 automated real-API orchestration attempts, every automated configuration—including a deterministic template baseline—recorded 0% execution success because no workflow reached component invocation. We initially attributed the collapse to free-form LLM outputs rejected by a strict JSON parser but controlled replication overturned that diagnosis: planner outputs were parse-valid in 80–100% of full-corpus trials, and a lightweight repair pass recovered all tested parsing failures. Pilot logs showed zero tokens consumed and uniform 30-second timeouts, which is more consistent with a harness-level failure before model-call completion than with parser rejection of completed model outputs.

The broader lesson is methodological as well as architectural: production AI teams must validate failure diagnoses as rigorously as they validate models. Fallback-populated metrics can mask complete execution collapse and support plausible but incorrect root-cause narratives. We thus recommend release gates for executable workflow artifacts, orchestration-level recovery, execution-success monitoring, template fallback, and controlled replication of any proposed root cause before it drives an engineering fix or publication claim.

1 Introduction

From plans to operational workflows. AI-enabled software systems are moving beyond single-step code generation toward multi-step workflows that combine models, retrieval systems, build and test services, issue trackers, documentation tools, and external APIs. Users increasingly express high-level intents (*e.g.*, diagnose a failing continuous integration run, localize the defect, propose a patch, rerun tests, and draft a release note) rather than invoking one tool at a time. The central engineering challenge is thus not whether a model can produce a plausible plan, but whether the surrounding system can transform an open-ended request into a machine-valid workflow that can be validated, observed, retried, cost-controlled, and executed against real components.

Dynamic intent-to-workflow orchestration addresses this challenge by translating natural-language intent into an executable workflow graph whose nodes select tools and whose edges encode data dependencies. Recent systems such as Opal and LangGraph-based orchestration frameworks illustrate the promise of this direction [Google Research, 2024, Wang and Duan, 2024, LangChain Inc., 2023]. What remains under-documented, however, is what happens when orchestration logic leaves controlled evaluation and is connected to real APIs, runtime infrastructure, and execution paths.

The execution boundary. Figure 1 summarizes the workflow path examined in this study. A user's natural-language

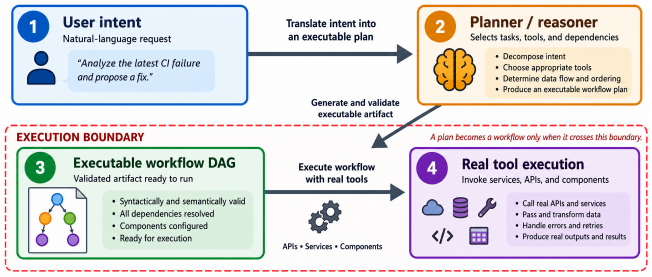

Figure 1: Intent-to-Workflow Orchestration Pipeline.

intent is first translated into tasks, tools, and dependencies; that plan must then become a validated executable workflow DAG before it can cross the execution boundary and invoke real services, APIs, and software components. The figure highlights the paper's core deployment insight: benchmark-quality plans create operational value only when the validated artifact reaches real execution.

This boundary between planning and execution is the focus of our study. In controlled benchmarking, our orchestration system appeared promising: it produced workflow structures and component selections that compared favorably with expert-designed workflows. In our real-API pilot, however,

every automated configuration failed before any workflow reached component invocation. We initially attributed this collapse to a plausible and common explanation: free-form LLM outputs appeared likely to violate the strict JSON contract expected by the execution runtime.

Our initial diagnosis was plausible but wrong. Fallback-populated metrics preserved similarity, quality, and cost values despite 0% execution success, making a complete execution failure look like degraded performance. Controlled replication falsified the parsing hypothesis: planner outputs were largely parse-valid, and repair recovered the tested failures. The available evidence instead points to a harness-level failure before model completion, though the missing runtime trace prevents us from confirming the exact mechanism.

Real-API pilot lesson and contributions. The strongest lesson from this pilot is not simply that an AI workflow orchestration system failed, but that we nearly accepted the wrong causal story. We therefore treat orchestration as a first-class failure domain: the key question is not only whether an LLM can plan a workflow, but whether the surrounding infrastructure reliably turns intent into execution—and whether the evidence supports the explanation when a pilot fails.

This paper makes four practitioner-focused contributions. It reports a gated pilot in which a benchmark-promising orchestration stack failed at the planning–execution boundary before any automated workflow reached component invocation. It shows how fallback-populated similarity, quality, and cost metrics concealed infrastructure failure and made a model-centered diagnosis appear credible. It then presents a controlled multi-model replication that overturned the original JSON-parsing diagnosis, demonstrating why deployment failure diagnoses should be treated as testable claims rather than dashboard conclusions. Finally, it distills release gates for production orchestration: executable-artifact validation, execution-success monitoring, orchestration-level recovery, deterministic fallback, and empirical validation of root-cause hypotheses.

The resulting message is directly aligned with deployment-oriented AI practice: AI systems fail not only when models reason poorly, but also when the operational path from model output to real execution is brittle, poorly instrumented, or misdiagnosed. For teams deploying agentic and workflow-based AI systems, the lesson is clear: validate the artifact, monitor whether it crosses the execution boundary, and verify the failure story before acting on it.

Paper organization. The remainder of the paper is organized as follows: Section 2 describes the real-API pilot setting; Section 3 analyzes benchmark and pilot results; Section 4 summarizes operational lessons; Section 5 explains validity limits; Section 6 compares our study with related work; and Section 7 presents concluding remarks.

2 Industrial Context and Pilot Setting

Nexoviq AI is an AI research and consulting company that builds workflow automation systems for software-engineering clients. Its target workflows often begin as broad requests (*e.g.*, "analyze this failing build, localize the fault, propose a patch, rerun tests, and draft a release note") that must be decomposed into coordinated service calls. We study an intent-to-workflow orchestration pattern that uses LLM-driven task decomposition and component selection to generate executable directed acyclic graphs (DAGs) over a controlled registry of 25 components spanning code generation, repository analysis, test execution, documentation and issue management, CI/CD integration, and data services.

The runtime includes retries, fallback behavior, and observability over traces, costs, and execution outcomes. This setting captures a practical software-engineering automation challenge: workflows must remain flexible as model providers and service interfaces change, but they must also produce artifacts that are valid, schedulable, and safe to execute. Manual workflow authoring remains a useful baseline, but it becomes expensive and brittle as the number of intents, tools, and execution paths grows.

The evaluation covered the eight conditions shown in Table 1: three baselines, the full orchestration system, and four ablations. Across 50 intents, three complexity tiers, and three replications per condition, the study comprised 1,200 total runs. We used a hybrid setup with simulated components for expensive services and real APIs for selected high-value components. A run counted as successful only if the workflow artifact reached component invocation; fallback to a placeholder workflow scored 0%, regardless of downstream metric values.

Table 1: Evaluated System Conditions

Condition	**Purpose**
B0 Manual	Expert-designed workflow
B1 Template	Deterministic template automation
B2 Single-shot	Single-shot LLM generation
FULL	Full orchestration (planning, selection, recovery, & optimization)
A1 No reasoning	No component-selection reasoning
A2 No recovery	No retry/fallback
A3 No optimization	No parallelization/redundancy elimination
A4 No coordination	No cross-service consistency

The A1–A4 ablations are grouped because the deployment argument turns on their shared 0% real-execution outcome; their simulated ablation values are reported separately in the replication package.

The real-API pilot exercised all eight conditions at full replication, yielding 150 runs per condition. The seven automated conditions—B1, B2, FULL, and A1–A4—therefore constituted 1,050 real orchestration attempts. The 0% execution-success results reported in Section 3 are measured over that full set, not over a small collection of illustrative failures. This scale matters because the observed collapse was uniform across both model-driven and deterministic automated configurations.

Operationally, the workflow path contains four checkpoints: interpreting the request, selecting and sequencing components, validating the resulting DAG, and invoking the selected services. Each checkpoint can fail independently, so success at one stage does not prove that the next stage was

reached. We therefore separate planning-oriented measures from execution-oriented evidence: the former describe what the system appears capable of generating, whereas the latter establish whether the artifact crossed the orchestration boundary and invoked real components. This distinction is central to interpreting the pilot results and to avoiding inflated conclusions from fallback-populated metrics.

We treat this as an operational pilot failure rather than a laboratory benchmark failure because the system was connected to real APIs and exercised inside the orchestration harness intended for deployment. Component invocation was deliberately gated to prevent downstream user harm and external repository modification. The consequence was operational rather than hypothetical: the pilot failed its rollout gate, shifted engineering effort from workflow expansion to infrastructure diagnosis, and halted broader automation work until the orchestration boundary could be hardened.

The affected stakeholders were internal engineers evaluating software-engineering automation. This yielded a failed real-API pilot rather than a public production outage. That scope is important since the lesson is not about downstream user harm, but about a realistic orchestration path with real execution dependencies failing before operational value could be delivered.

The intent corpus, summarized in Table 2, was stratified into three complexity tiers. Tier 1 intents covered simple single-service tasks, Tier 2 intents required moderate coordination across two or three service domains, and Tier 3 intents involved seven or more steps across four or more domains. This stratification matters because orchestration is not equally valuable for all tasks: simple intents often favor deterministic templates, whereas higher-complexity intents are where dynamic planning should provide the greatest benefit.

Table 2: Intent Corpus Used in the Real-API Pilot

Tier	Count	Steps	Service domains
Tier 1 (Simple)	15	1–3	1
Tier 2 (Medium)	20	4–6	2–3
Tier 3 (Complex)	15	7+	4+

Each component in the 25-service registry carried capability, cost, and quality metadata. The full system used that metadata to reason about trade-offs rather than selecting the first matching component. We tracked workflow correctness, component-selection quality, output quality, and execution success, but treated execution success as the dominant outcome because attractive benchmark scores are operationally meaningless if the orchestration layer fails before tool execution begins.

3 Results: Benchmark Promise vs. Real-API Pilot Reality

This section contrasts the strong performance of our benchmark (which showed 80.63% structural similarity, 88% component-selection accuracy, and 92% output quality) with complete execution failure in the real-API pilot. Every automated condition, including deterministic and model-driven configurations, recorded 0% execution success. Section 3.4 then revisits our original parsing-based diagnosis and explains why the available evidence is more consistent with failure before model-call completion.

3.1 Benchmark Performance Before the Real-API Pilot

The controlled benchmark suggested that dynamic orchestration is viable in principle. The full system reached 80.63% structural similarity to expert workflows, 88% component-selection accuracy, and 92% of manual output quality at 2.1× faster execution than manual design. Component-selection reasoning was the strongest contributor to quality, and these results justified a real-API pilot.

Under controlled conditions, the system generated plausible workflow structures and component choices. The pilot thus exposed a deployment gap between benchmark-quality planning and operational execution, rather than disproving the orchestration concept itself. These simulated numbers do not represent real API execution: FULL narrowed the gap to manual design and exceeded both template-based and single-shot quality, yet none of these gains carried over to the pilot, as shown in the **Real Execution Success** column in Table 3.

3.2 Real-API Pilot Failure

Table 3 shows how only the manual baseline (B0) succeeded. Across 150 runs per condition, all automated configurations—AI-driven and deterministic—recorded 0% execution success, yielding 1,050 failed real-API orchestration attempts. We initially attributed this uniform collapse to a parsing failure: LLM planners were believed to wrap valid workflow JSON inside explanatory prose, causing a strict-JSON parser to fail and fall back to a placeholder workflow before any component was invoked.

Table 3: Benchmark (Simulated) and Real-API Pilot Results by Condition.

Condition	Simulated Structural Similarity	Simulated Selection Accuracy	Simulated Output Quality	Real Execution Success
B0 Manual	94.36	100%	100%	100.0%
B1 Template	73.31	85%	75%	0.0%
B2 Single-shot	74.59	75%	82%	0.0%
FULL	80.63	88%	92%	0.0%
A1–A4	—	—	—	0.0%

Our initial diagnosis was plausible since it matched the uniform failure across LLM-driven conditions and pointed to an actionable fix (schema-constrained generation). However, it was insufficiently tested since the deterministic template baseline (B1), which emits no free-form text at all, failed identically, and we had not yet replicated the free-form failure mode itself under controlled conditions. Section 3.4 reports that replication and the evidence that changed our conclusion.

Figure 2 shows execution success dropping from near-complete success in the benchmark setting to 0% for every automated configuration in the real-API pilot. The key signal is not the fallback similarity score but the complete failure

of all automated conditions to achieve real execution, while benchmark-like evaluation fields remained populated by fallback multipliers—the proxy-metric trap examined next. Recovery mechanisms had been designed for component-level failures, such as retrying a tool or switching providers, not for orchestration-level failures in which the workflow itself never becomes executable. That distinction changes how such systems should be staged, budgeted, and validated.

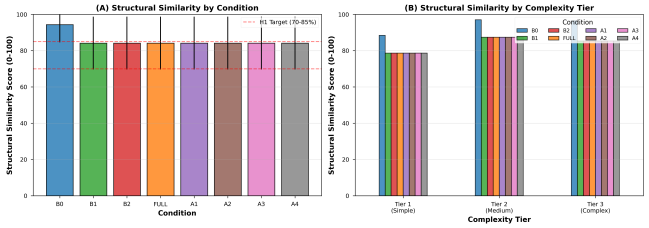

Figure 2: Structural Similarity Scores and Execution Success Rates in the Real-API Pilot.

3.3 Why the Dashboard Was Misleading

Figure 3 shows the dashboard pattern that pushed us toward the wrong diagnosis. Although no automated workflow

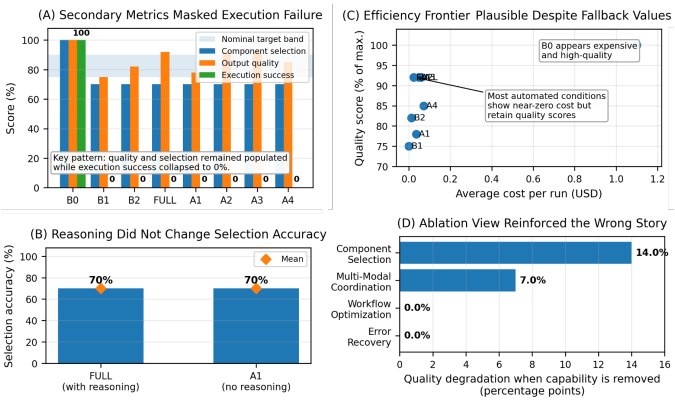

Figure 3: Component-selection, Output-quality, Cost, and Ablation Outcomes in the Real-API Pilot.

reached component invocation, fallback logic still populated similarity, component-selection, quality, and cost fields. The system thus looked degraded rather than collapsed. Since these values came from fallback defaults and multipliers rather than generated outputs or successful orchestration, secondary metrics obscured total execution failure.

Low API spend reinforced the illusion: it reflected early failure, not operational efficiency. Although component-selection reasoning still shaped the fallback analysis, downstream quality gains are irrelevant once no workflow crosses the orchestration boundary. Execution success must therefore gate all similarity, quality, selection, and cost metrics; otherwise, dashboards can hide catastrophic failure and support the wrong root-cause story.

3.4 Controlled Root-Cause Replication

Our initial diagnosis made a testable prediction: if free-form generation reliably fails to produce parse-valid JSON, replaying the planning prompt under controlled conditions should reproduce that failure. We tested three planner-output conditions: *free-form output* parsed with the pilot's json.loads parser; *the same response* passed through a lightweight extraction-and-repair pass; and *schema-constrained generation* via JSON mode or function calling. Schema-constrained generation guaranteed a conforming object where supported; the free-form and repair conditions tested whether the original parsing diagnosis reproduced under controlled replay of the 50-intent corpus.

We treated our original root-cause diagnosis as a testable real-API pilot hypothesis, as follows:

- **Original diagnosis:** prose-wrapped planner output caused the strict JSON parser to reject workflows before execution.
- **Prediction:** replaying the same planner prompts should yield near-0% parse-valid outputs.
- **Result:** parse-valid output rates were 80–100% and a lightweight repair pass recovered all tested failures, as shown in Table 4.[1]
- **Contradictory evidence:** the deterministic template baseline, which emits no free-form text, failed identically.
- **Better-supported explanation:** each automated pilot run consumed zero tokens and terminated at a flat 30-second timeout, indicating that no model call ever completed.
- **Lesson:** execution-grounded replication is required before claiming a root cause.

Table 4: Controlled Replication of the Planner's Free-Form Output (Section 3.4)

Model / prompt	Scope	Free-form	Repair	n
GPT-4, strong prompt	Full corpus	100%	100%	50
GPT-4, original-style prompt	Full corpus	80%	100%	50
Tier 1 subset	Subset	33%	100%	15
Tier 2–3 subset	Subset	100%	100%	35
Frontier model (2026), strong prompt	Full corpus	100%	100%	135

Replication overturned our original parsing diagnosis. Under the original prompt style, GPT-4's free-form parse rate dropped to 33% for simple Tier-1 intents while remaining 100% for the more complex Tier-2 and Tier-3 workflows—the opposite of the intuition that complex workflows were the primary source of parsing failures. Moreover, JSON mode was unavailable on the GPT-4-0613 endpoint, so our previously proposed remedy could not even have been applied to the pilot configuration. Although prose wrapping affected 20% of the original GPT-4 corpus (67% of Tier-1 intents), a lightweight repair pass recovered every tested failure, demonstrating that parsing errors were real but readily correctable.

More importantly, parsing failures cannot explain the observed pilot collapse. They were too infrequent, prompt-

[1] Tier rows are subsets of the GPT-4 original-style condition.

Table 5: Release Gates for Preventing Orchestration-Boundary Failure

Release gate	What it prevents	Signal that the gate failed
1. Schema-valid workflow artifact	Prevents parser rejection before execution	Response contains prose, partial JSON, or malformed fields
2. Executable dependency graph	Prevents the runtime from receiving unschedulable or incomplete workflows	Missing inputs, cycles, or unresolved component bindings
3. Orchestration-level recovery path	Recovers from malformed plans before any component execution begins	Repeated fallback to placeholder workflows or null plans
4. Execution-success monitoring	Distinguishes real workflow execution from dashboard artifacts	Similarity or quality fields populated while execution success is 0%
5. Template fallback policy	Constrains risk for routine requests when dynamic planning is unstable	Frequent fallback on common intents without bounded behavior
6. Root-cause replication before remediation	Prevents plausible diagnoses from becoming untested fixes	Diagnosis rests on plausibility, a single trace, or fallback-populated dashboards

sensitive, recoverable by repair, and absent from the deterministic template baseline. Yet every automated condition failed uniformly. The pilot logs instead recorded `tokens_consumed = 0` and a flat 30-second runtime for each automated run, indicating that execution failed in the orchestration harness before any model call completed. he practical lesson is that parser hardening remains worthwhile, but production release gates must first verify end-to-end orchestration—dispatch, response receipt, validation, scheduling, and workflow handoff—before interpreting downstream quality, cost, or routing metrics.

3.5 Revised Diagnosis and Implications

Replication and pilot logs together overturned our diagnosis. Parsing failures were real but recoverable, and they could not explain a uniform 0% collapse across all automated conditions, including the deterministic template baseline. The stronger evidence points to an upstream harness failure: zero tokens consumed and uniform timeouts suggest that no model invocation completed, though the missing runtime trace prevents us from confirming the exact mechanism.

The lesson is not that structured outputs are unnecessary; they remain sound engineering practice. The lesson is that our dashboard could not distinguish invalid model output from a harness that never obtained model output. That blind spot made the wrong diagnosis look credible.

The corrected diagnosis changed the remediation strategy. Rather than treating schema-constrained generation as the primary remedy, we focused on hardening the orchestration boundary with stage-specific telemetry and deterministic smoke tests through the same execution path. The resulting strategy is simple: instrument the path from prompt dispatch to component invocation before trusting downstream workflow metrics.

4 Operational Lessons for Production AI Orchestration

This section summarizes our real-API pilot experience, including the controlled replication that overturned our own initial root-cause diagnosis, and distills the following four operational lessons for teams introducing dynamic orchestration into products and internal platforms.

1. Replicate failure diagnoses before adopting fixes. Structured-output guarantees remain good engineering practice for workflow artifacts that must be executable [OpenAI, 2023]. In this pilot, however, schema-constrained generation was not the demonstrated fix: free-form generation already succeeded 80–100% of the time on the model generation associated with the pilot failure. The broader lesson is diagnostic: treat a proposed root cause as a hypothesis, not a finding, and reproduce it under controlled conditions with the model, prompt, parser, and execution path used in the pilot before adopting a fix or publishing a causal claim.

2. Instrument orchestration as a first-class failure domain. Recovery logic should not be limited to individual tool invocations. Production systems need orchestration-level validation, repair, retry, and telemetry that distinguish prompt dispatch, model-call completion, parser invocation, artifact validation, scheduler handoff, and component invocation. Without that stage-level visibility, teams cannot tell whether a failure occurred in the model, parser, harness, scheduler, or downstream service.

3. Validate deterministic execution before introducing dynamic planning.

Template-based automation remains attractive for routine, high-frequency tasks because it provides predictable execution and bounded cost. Reserve dynamic orchestration for novel, higher-complexity tasks where template coverage is poor. Once workflow validity is guaranteed, component-selection reasoning produced the largest quality gains in our benchmark study, so it should be the next engineering priority ahead of secondary features such as optimization or richer coordination.

4. Budget for integration overhead, not just model usage. Operational costs are shaped by failed attempts, retry logic, validation overhead, and the engineering effort required to harden the orchestration layer; simple per-token projections underestimate true operational cost.

Together, these lessons argue against an autonomy-first rollout. Dynamic orchestration should be introduced as an engineering system, not merely as a model capability. Teams should begin with deterministic execution paths, instrument narrow pilots to identify the failed stage of the pipeline, and expand only after the harness, parser, validator, scheduler, and

component handoff have all been exercised successfully.

Table 5 summarizes the minimum release gates our pilot experience suggests. These checks sit at the orchestration boundary; a team that validates model quality but not workflow executability remains exposed to complete system failure at deployment time.

5 Deployment Implications & Validity

Beyond the root-cause revision in Section 3.5, two deployment implications emerge. First, recovery addressed component failures but not orchestration failures in which workflows never became executable. Second, cost estimates based on successful orchestration underestimated real deployment costs, where failed orchestration attempts can dominate before any downstream execution occurs.

Industrial deployment should therefore follow a staged validation strategy: benchmark under controlled conditions, validate orchestration in a focused real-world pilot, and only then scale execution. Likewise, proposed root causes should be replicated before informing engineering fixes, release decisions, or publication claims.

Our study has important validity limits. The original orchestration-runtime trace was unavailable, so the harness-level failure proposed in Section 3.4 is the best-supported explanation rather than a confirmed mechanism. It is consistent with the observed zero tokens consumed, uniform timeouts, and replication evidence that rules out free-form JSON parsing as the main cause of the uniform 0% collapse. Our replication also covered only two model generations, one prompt family, and 235 free-form planner responses, so other systems may exhibit different parsing behavior. These limitations narrow the causal claim, but not the central deployment lesson: fallback metadata can conceal complete execution failure and reinforce incorrect root-cause diagnoses.

6 Related Work

Prior agent frameworks, such as LangChain, Toolformer, and ReAct [LangChain Inc., 2023, Schick et al., 2023, Yao et al., 2023], make multi-step tool use more accessible, and surveys of LLM-based agents [Wang et al., 2023] emphasize the promise of autonomous planning and tool use. These systems and surveys provide less evidence, however, about the reliability of the orchestration boundary in deployment.

Traditional workflow platforms, such as Zapier and Apache Airflow [Barricelli and Valtolina, 2020, Apache Software Foundation, 2020, van der Aalst et al., 2003], remain attractive because their control flow is explicit and machine-valid from the start. Our contribution is therefore not a new agent framework, but a deployment lesson about how to instrument, evaluate, and diagnose the boundary between generated plans and operational execution.

An ASE 2025 study found that interface and coordination failures—not model output quality—were the dominant defect source in LLM agent orchestration frameworks [Chen et al., 2025]. Likewise, WorkflowLLM [Bao et al., 2024] showed planning quality and execution reliability can diverge significantly. These findings are consistent with our case study: planning quality and execution reliability can diverge, and the orchestration boundary can fail before any downstream tool has a chance to help.

JSONSchemaBench [Peng et al., 2025] shows even capable models produce schema-invalid output at non-trivial rates. This finding is consistent with our replication's modest, prompt-sensitive failure rate (see Section 3.4), but not the uniform 0% real-API pilot collapse. Execution-failure-aware evaluation work [Silva et al., 2024, Liang et al., 2022, Patel et al., 2024] treats execution success as a hard gate rather than a secondary metric, which is a treatment we extended to the diagnosis of failure itself in our study.

7 Concluding Remarks

This paper examined a real-API pilot of a dynamic intent-to-workflow orchestration system spanning 50 intents, 25 components, and eight experimental conditions. Controlled benchmarking suggested that the approach was viable: the full system produced strong workflow structure, component selection, and output quality. Yet none of these gains translated to the pilot, where all automated conditions failed before reaching component invocation.

Our initial explanation for that collapse—strict parsing of free-form LLM output—did not survive controlled replication. The strongest evidence instead points to a harness-level failure that prevented model calls from completing, although the unavailable runtime trace prevents us from claiming that mechanism as definitively confirmed. The resulting lessons extend beyond this particular system:

- **Treat root-cause diagnoses as hypotheses.** Validate suspected failures by reproducing them with the same model, prompt, and execution path before adopting a fix or making a causal claim.
- **Treat orchestration as a first-class failure domain.** Validate, repair, and make workflows executable before downstream components run, with recovery spanning construction, scheduling, validation, and handoff.
- **Prioritize executability over sophistication.** Establish deterministic execution, workflow validation, and reliable component invocation before introducing dynamic orchestration.
- **Make execution success the primary release gate.** Gate releases on verified execution and orchestration health before interpreting similarity, quality, cost, or component-selection metrics.

The central lesson is both methodological *and* architectural: production AI teams must validate what their models generate, verify that the surrounding infrastructure executes those outputs, and confirm that the evidence supports the explanation when execution fails. In workflow orchestration, the model may be the most visible component, but system reliability depends on the entire path from intent to execution.

Data Availability Statement

All data, analysis scripts, and representative workflow artifacts are publicly available at github.com/nexoviq-ai/ase26-orchestration.

References

Apache Software Foundation. Apache Airflow: A platform to programmatically author, schedule and monitor workflows. In *Proceedings of the VLDB Endowment*, 2020.

S. Bao et al. WorkflowLLM: Enhancing workflow orchestration capability of large language models. *arXiv preprint arXiv:2411.05451*, 2024.

B. R. Barricelli and S. Valtolina. End-user development of API-based workflow automations. In *Proceedings of the International Conference on Advanced Visual Interfaces*, pages 1–5, 2020.

Y. Chen et al. An empirical study of bugs in LLM-based agent workflow orchestration frameworks. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2025.

Google Research. Opal: Multimodal AI for creative tasks. Technical report, Google Research, 2024.

LangChain Inc. LangChain: Building applications with LLMs. https://python.langchain.com/, 2023.

P. Liang, R. Bommasani, T. Lee, D. Tsipras, D. Soylu, M. Yasunaga, Y. Zhang, D. Narayanan, Y. Wu, A. Kumar, B. Newman, B. Yuan, B. Yan, C. Zhang, C. Cosgrove, C. D. Manning, and C. Re. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

OpenAI. Function calling and other API updates. *OpenAI Blog*, 2023.

A. Patel, O. S. Svendsen, and Y. Liu. Cost-effective language model inference: Strategies and trade-offs. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2024.

N. Peng et al. JSONSchemaBench: Benchmarking LLMs on JSON schema-constrained generation. *arXiv preprint*, 2025.

T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

M. Silva et al. Execution-aware evaluation of LLM-generated tests: Lessons from flakiness in production. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024.

W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. In *Distributed and Parallel Databases*, volume 14, pages 5–51. Springer, 2003.

J. Wang and Z. Duan. Agent AI with LangGraph: A modular framework for building multi-agent systems using large language models. *arXiv preprint arXiv:2412.03801*, 2024.

L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. R. Wen. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432*, 2023.

S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.