Dynamic AI Workflow Orchestration in Practice: Deployment Lessons

Ashraf Elnashar
Nexoviq AI
AI Research and Consulting
Irvine, CA, USA
ashraf.elnashar@nexoviq.ai

Jules White
Vanderbilt University
Department of Computer Science
Nashville, TN, USA
jules.white@vanderbilt.edu

Douglas C. Schmidt
William & Mary
Department of Computer Science
Williamsburg, VA, USA
dcschmidt@wm.edu

Abstract

Dynamic intent-to-workflow orchestration offers a compelling approach to complex automation by translating high-level user intent into executable workflows spanning heterogeneous AI models and services. This paper presents a deployment-oriented study of an orchestration system that constructs workflow DAGs over 25 registered components. In controlled benchmarking across 50 intents and 1,200 runs, the system demonstrated strong performance, achieving 80.63% structural similarity to expert-designed workflows, 88% component-selection accuracy, and 92% of manual-workflow output quality—suggesting substantial promise for automated orchestration.

However, these results did not translate into real-world execution. In deployment, every automated orchestration attempt failed prior to meaningful workflow execution because free-form model outputs could not consistently produce machine-valid workflow specifications. This breakdown reveals a critical gap between benchmark success and operational reliability, highlighting that interface robustness—not planning sophistication—was the primary barrier to deployment.

The paper makes three contributions to research on dynamic AI workflow orchestration. First, it provides an empirically-grounded analysis of a previously underreported failure mode in AI orchestration systems. Second, it demonstrates why proxy evaluation metrics can be misleading when systems fail at the execution interface layer. Third, it distills actionable design guidance for practitioners, including the use of schema-constrained outputs, the necessity of orchestration-level validation and repair mechanisms, and the recommendation to prioritize deterministic workflow templates before introducing dynamic planning.

Keywords

Software engineering, AI workflow orchestration, industry experience report, deployment lessons, tool orchestration

ACM Reference Format:
Ashraf Elnashar, Jules White, and Douglas C. Schmidt. 2026. Dynamic AI Workflow Orchestration in Practice: Deployment Lessons. In *ASE '26: Industry Showcase.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Automated software engineering workflows increasingly span multiple models, retrieval systems, build/test services, and external APIs. Examples include CI failure diagnosis, release-note generation, test-script synthesis, remediation planning, and cross-tool developer assistants. In these settings, the hardest systems problem is often not single-step generation quality; it is reliably transforming an open-ended request into an executable, machine-valid workflow that can be validated, observed, retried, and cost-controlled.

Dynamic intent-to-workflow orchestration addresses this problem by translating a user 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 this design direction [5, 6, 14]. Yet industrial teams face a practical question that remains under-documented: what actually happens when such orchestration logic is pushed from controlled evaluation into realistic deployment conditions?

This paper presents an industry-oriented account of that question. Rather than positioning orchestration only as a promising pattern, we examine the operational gap between benchmark-level performance and deployment behavior. In our controlled study, the full orchestration system achieved encouraging quality and component-selection results. In real deployment, however, the same system failed systematically at the orchestration boundary because free-form LLM outputs were not robustly parseable into machine-readable workflow specifications. That failure is important for practice because it reveals a bottleneck that sits above individual tool calls and below user-visible functionality.

The contributions of this paper are therefore practical rather than purely algorithmic. We report the deployment setting, analyze the dominant failure mode at the orchestration boundary, and derive concrete recommendations for industrial rollout. The central lesson is that production viability depends less on adding more orchestration features and more on reducing orchestration brittleness through executable interface contracts, structured output APIs, validation, and staged deployment strategies.

The remainder of this paper is organized as follows: Section 2 describes the industrial context and deployment setting of our work; Section 3 analyzes the results from our benchmarks; Section 4 summarizes software engineering lessons for deployment; Section 5 examines deployment implications and validity of our work; Section 6 compares our results with related work; and Section 7 presents concluding remarks.

2 Industrial Context and Deployment Setting

Nexoviq AI is an AI research and consulting practice that builds and operates workflow automation systems for software engineering clients. Dynamic orchestration is attractive because many client requests are under-specified at intake but require multiple coordinated services by execution time. A request such as "analyze this

ASE '26, Washington, DC, USA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

failing build, localize the fault, propose a patch, rerun tests, and draft a release note" must be decomposed into repository inspection, log analysis, code generation, test execution, validation, and reporting steps. Authoring and maintaining those workflows manually is expensive, especially as model providers change frequently.

The pattern we study translates a natural-language intent into an executable directed acyclic graph (DAG). In software engineering terms, that DAG is a contract artifact between planning and execution: downstream systems depend on it being valid, machine-readable, and schedulable. Nexoviq AI uses LLM-driven reasoning for task decomposition and component selection, together with a controlled registry of 25 components spanning code generation, repository analysis, test execution, documentation and issue-management tools, CI/CD integrations, and supporting data services. The runtime also includes retries, fallback behavior, and observability over workflow traces, costs, and execution outcomes.

The evaluation studied eight conditions: three baselines, one full orchestration system, and four ablations. Baselines included manual workflow design, template-based automation, and single-shot LLM workflow generation. The full system combined intent decomposition, component selection reasoning, workflow generation, cross-service coordination, workflow optimization, and error recovery. The overall study included 50 intents across three complexity tiers and three replications per condition, yielding 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 counts as real execution success only when the workflow artifact reached component invocation—runs that triggered fallback to placeholder workflows are scored as 0% regardless of downstream metric values.

Table 1: Evaluated System Conditions

Condition	Purpose
B0 Manual	Expert-designed workflow and component selection
B1 Template	Deterministic template-based workflow automation
B2 Single-shot	Direct LLM workflow generation without iteration
FULL	Complete orchestration with planning, selection, recovery, and optimization
A1 No reasoning	Removes explicit component-selection reasoning
A2 No recovery	Removes retry and fallback behavior
A3 No optimization	Removes parallelization and redundancy elimination
A4 No coordination	Removes cross-service consistency mechanisms

The intent corpus was stratified to reflect realistic industrial variation. Tier 1 intents represented simple single-service tasks with one to three steps. Tier 2 intents represented medium-complexity workflows with four to six steps and moderate cross-service coordination. Tier 3 intents represented complex workflows with seven or more steps, multiple service domains, and richer dependency structures. This matters for deployment because the break-even point for orchestration is not the same across those tiers: simple tasks often favor deterministic templates, whereas higher-complexity tasks are where dynamic planning is expected to provide value.

Table 2: Intent Corpus Used in the Deployment Study

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+

The component library contained 25 services across six categories: code generation, repository and static-analysis tools, test and validation services, documentation and issue-management tools, CI/CD and deployment integrations, and supporting data services. Each entry carried capability descriptions, cost information, and quality metadata. That metadata was central to the full system because component selection depended on explicit trade-offs rather than simple first-match assignment.

We tracked four outcome dimensions: workflow correctness, component selection quality, output quality, and execution success. For the software engineering lens of this paper, execution success is the dominant outcome because a workflow with attractive benchmark scores is still unusable if the orchestration layer fails before tool execution begins.

3 Results: Benchmark Promise vs. Deployment Reality

This section contrasts strong benchmark performance—80.63% structural similarity, 88% component-selection accuracy, and 92% output quality—against complete execution failure in real deployments, where every automated condition recorded 0% success due to parse failures at the orchestration interface.

3.1 Benchmark Signals Before Deployment

The controlled benchmark suggested that dynamic orchestration is viable in principle. The full system reached 80.63% structural similarity to expert workflows and 88% component selection accuracy. It achieved 92% of manual output quality at 2.1× faster execution than manual design in the simulated setting. The ablation study further showed that component-selection reasoning was the single most important capability for quality preservation.

These benchmark-level results matter for industry because they explain why the orchestration approach is attractive in the first place. The system was not failing due to obviously weak planning or uniformly poor component choices. Under controlled conditions, it behaved well enough to justify pilot deployment. That point is important because the deployment failure was not a result of a hopeless underlying concept; it was a result of an implementation bottleneck at the interface between reasoning and execution.

Table 3 shows why the system appeared promising before deployment. These numbers reflect the *simulated* study only, in which component responses were controlled; they do not represent real API execution. The full orchestration configuration narrowed the gap to manual design, exceeded template-based quality, and delivered better output quality than single-shot generation. On medium-complexity tasks, where industry teams are most likely to seek flexibility beyond template libraries, the simulated results were

Table 3: Selected Benchmark Results from the Simulated Study Only. *In real deployment all automated conditions had 0% execution success; see Table 4.*

Condition	Similarity	Selection	Quality
B0 Manual	94.36	100%	100%
B1 Template	73.31	85%	75%
B2 Single-shot	74.59	75%	82%
FULL	80.63	88%	92%

especially encouraging. The next subsection explains why none of these gains carried over to real deployment.

3.2 Execution Failure in Real Deployment

Those benchmark results did not carry over to real deployment. In the real execution setting, only the manual baseline (B0) succeeded; all other conditions—AI-driven and template-based alike—recorded 0% real execution success. For AI-driven conditions the dominant failure was the orchestration layer's inability to transform LLM-generated workflow descriptions into valid, machine-consumable workflow objects; the parser expected strict JSON but received prose-wrapped fragments, causing every automated condition to fall back to placeholder workflows before any real component was invoked. The template baseline (B1) failed for a related reason: it shared the same schema-validation and component-invocation infrastructure, which also required a parse-valid workflow artifact before any tool execution could begin.

Figure 1 shows how execution success dropped from near-complete success in the benchmark setting to 0% for all AI-driven configurations in real deployment. These results indicate how the key signal

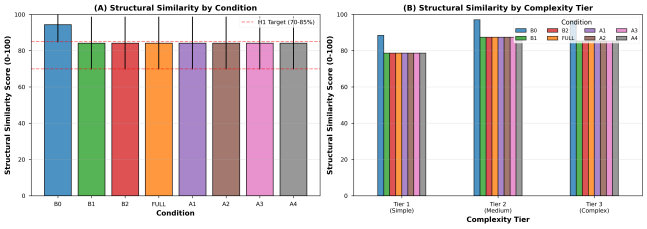

Figure 1: Structural Similarity Scores and Execution Success Rates in Real Deployments.

is not the fallback similarity score, but the complete failure of all automated conditions to achieve real execution. The LLMs often produced semantically reasonable workflow content, but they embedded that content inside explanatory prose. The parser expected strict JSON and therefore rejected responses that mixed natural language with structured fragments. Once parsing failed, the system fell back to placeholder workflows. This caused benchmark-like evaluation fields to remain populated while real execution never actually progressed through the intended orchestration pipeline.

Table 4 makes the gap explicit. In the controlled setting, execution rates were high enough to make the system appear robust. In the real setting, the orchestration boundary collapsed for every AI-driven condition. Recovery mechanisms had been designed for component-level failures such as retrying a tool or switching providers; they were not designed for orchestration-level failures

Table 4: Execution Success Rates in Simulated and Real Settings

Condition	Simulated	Real
B0 Manual	100.0%	100.0%
B1 Template	100.0%	0.0%
B2 Single-shot	96.7%	0.0%
FULL	99.3%	0.0%
A1–A4	88.7–98.0%	0.0%

in which the workflow itself never becomes executable. For practitioners, this changes how the system should be staged, budgeted, and validated.

3.3 Behavior Across Quality, Selection, and Cost Dimensions

Although execution failed, the fallback logic still produced enough metadata to inspect how the real deployment would have been interpreted by downstream evaluation pipelines. This is useful because it shows how misleading a deployment can appear if execution success is not treated as a primary gate.

Figure 2 shows that fallback defaults produced uniform selection scores across AI-driven conditions. The uniform AI behavior

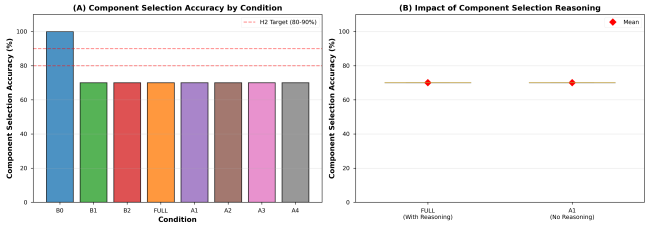

Figure 2: Component-selection Outcomes in Real Deployments.

reflects fallback defaults rather than successful orchestration. That uniformity is operationally significant because it masks the actual value of component-selection reasoning. A monitoring setup that only looked at final selection fields could easily misdiagnose the problem as acceptable degraded behavior rather than orchestration collapse.

Figure 3 shows a similar pattern for quality proxies. Several

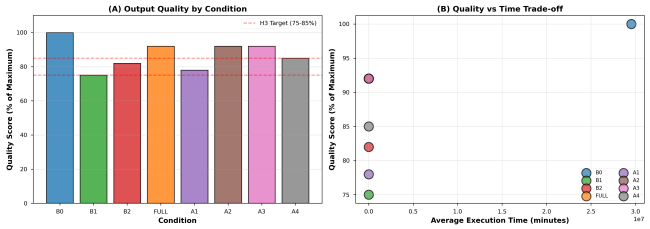

Figure 3: Output-quality Scores in Real Deployments.

AI-driven conditions appear to retain moderate quality relative to manual workflows, but those values come from fallback multipliers rather than generated outputs. These values are artifacts of fallback metadata and should not be treated as evidence of successful

execution, which is precisely why industrial evaluation must separate "looks plausible in the dashboard" from "actually executed and produced real artifacts."

The cost profile in Figure 4 adds a further deployment lesson.

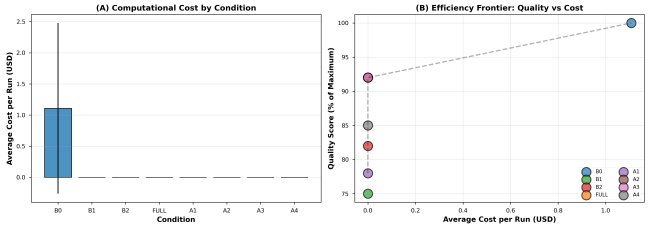

Figure 4: Cost Behavior in Real Deployments.

This figure shows how AI-driven conditions failed before incurring most execution cost, concentrating spending in the manual baseline. Real deployments consumed much less API cost for AI-driven conditions than predicted, not because the system became efficient, but because it failed before it could execute. Cost dashboards alone therefore cannot be interpreted as a success signal. Low cost may reflect orchestration failure just as easily as operational efficiency.

3.4 Capability Importance Under Failure

Figure 5 shows that component-selection reasoning still mattered in the fallback analysis. However, the broader lesson is architectural: once orchestration reliability collapses, the relative value of downstream optimizations becomes secondary. Component-

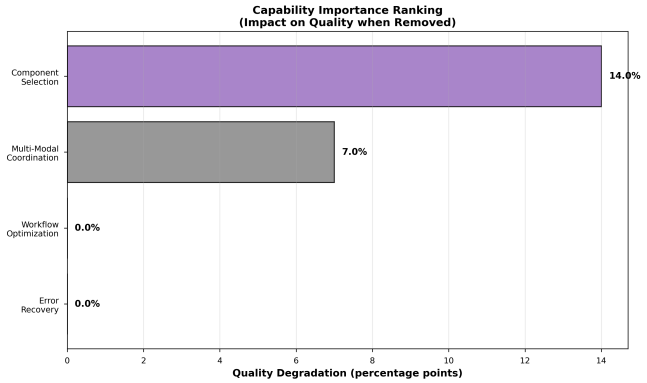

Figure 5: Ablation Findings in Real Deployments.

selection reasoning remains important, but execution-level insights are dominated by orchestration failure before workflow execution begins. This finding aligns with the simulated ablation study, where component-selection reasoning was the strongest contributor to quality, while optimization mainly improved latency and error recovery had limited value in stable settings.

4 Software Engineering Lessons for Deployment

This section summarizes the results of our deployment experience, which suggests the following five pragmatic lessons for teams introducing dynamic orchestration into products and internal platforms.

1. Use structured output guarantees from the start. Parsing free-form model text into machine-readable workflows is too fragile for production use. Systems should prefer function calling, tool-use APIs, schema-constrained generation, or JSON modes whenever the workflow representation must be executable [8].

2. Treat orchestration as its own failure domain. Recovery logic should not be limited to individual tool invocations. Production systems need orchestration-level validation, repair, and retry paths that handle malformed or partially valid workflow specifications.

3. Adopt a template-first rollout model. Template-based automation remains attractive for routine and high-frequency tasks because it provides predictable execution and bounded cost. Dynamic orchestration is better reserved for novel, higher-complexity tasks where template coverage is poor.

4. Prioritize component-selection quality after workflow validity is guaranteed. The benchmark study indicated that component-selection reasoning produced the largest quality gains. For engineering teams with limited implementation capacity, the priority order should be: make workflow generation machine-usable, improve component selection quality, and only then add secondary features such as optimization or richer coordination.

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

Taken together, these lessons argue against an autonomy-first rollout. Industrial teams should instead treat dynamic orchestration as an engineering system that must pass contract-level reliability checks before its higher-level reasoning quality matters. In practice, the deployment path should begin with deterministic templates for routine tasks, followed by narrow pilots that explicitly test orchestration-interface failures before broader rollout.

Table 5 summarizes the minimum release gates our deployment 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 and Validity

The divergence between simulated and real results highlights three concrete deployment challenges. First, **parsing robustness** remained inadequate: LLM responses often contained semantically useful workflow content, but not in a schema-conforming format that the runtime could consume directly. Second, **recovery scope** was mismatched to the real failure mode: the system handled component failures but not orchestration failures in which the workflow artifact itself never became executable. Third, **cost expectations** were overly optimistic: simulated cost estimates assumed successful orchestration, whereas real deployment showed that failed plans, retries, and validation overhead can dominate total cost of ownership before downstream execution even begins.

These observations suggest a three-phase validation path for industrial deployment. Teams should first benchmark under controlled conditions, then run a narrow real pilot explicitly targeted at orchestration-interface failures, and only then scale to broader execution studies. This sequence is a software engineering discipline rather than a modeling preference: it treats the workflow artifact, validator, and recovery loop as release-critical infrastructure.

Table 5: Recommended Release Gates for Dynamic Orchestration Deployment

Gate	Why it matters	Failure signal
Schema-valid workflow artifact	Ensures planning output can be consumed directly by the execution runtime	Response contains prose, partial JSON, or malformed fields
Executable dependency graph	Prevents the runtime from receiving unschedulable or incomplete workflows	Missing inputs, cycles, or unresolved component bindings
Orchestration-level recovery path	Recovers from malformed plans before any component execution begins	Repeated fallback to placeholder workflows or null plans
Execution-success monitoring	Distinguishes real workflow execution from dashboard artifacts	Similarity or quality fields populated while execution success is 0%
Template fallback policy	Constrains risk for routine requests when dynamic planning is unstable	Frequent fallback on common intents without bounded behavior

This paper also has clear validity limits. The observed failure mode is tied to one orchestration implementation and one parsing strategy, and other systems using stronger schema-constrained APIs may avoid this specific problem. In addition, some components in the broader study were simulated for budget control, which affects the absolute generalizability of cost and quality estimates. Those limitations do not change the central lesson, however, because the dominant deployment failure occurred before most downstream component execution took place.

6 Related Work

This section describes related work, which resides at the intersection of agent orchestration frameworks, workflow automation systems, structured-output reliability, and empirical evaluation of LLM-based systems.

Recent agent frameworks such as LangChain, Toolformer, and ReAct make multi-step reasoning and tool use more accessible [6, 11, 16]. Surveys of LLM agents emphasize the promise of multi-agent coordination and autonomous planning [15]. What they provide less often is deployment evidence about the reliability of the orchestration boundary itself. Our study complements that literature by showing that the failure point can occur before any individual tool has a chance to help.

Traditional workflow platforms such as Zapier and Apache Airflow emphasize deterministic integration, repeatability, and operational visibility [1, 3, 13]. They are less flexible than dynamic orchestration, but their control flow is explicit and machine-valid from the start. Our findings explain why these systems remain attractive in industry: deterministic workflow artifacts reduce a category of failure that agentic systems must still engineer around.

Recent empirical work on LLM-based agent frameworks has begun to characterize the reliability properties that matter most for production use. An ASE 2025 study of bugs in LLM agent workflow orchestration frameworks found that interface and coordination failures—not model output quality—were the dominant source of defects [4].

Likewise, WorkflowLLM provides a complementary benchmark perspective, evaluating orchestration capability across a broad task distribution and revealing that planning quality and execution reliability can diverge significantly [2]. Our deployment experience converges on the same diagnosis: the orchestration boundary, not the model, is the first point of failure.

The failure we observed—free-form text that is semantically plausible but not schema-valid—is directly addressed by structured-output and constrained-decoding research. JSONSchemaBench provides a systematic evaluation of how well current models satisfy JSON Schema constraints, finding that even capable models produce schema-invalid outputs at non-trivial rates under complex schemas [10]. Those findings strengthen the prescriptive recommendation in our lessons section: adopting schema-constrained generation is a reliability engineering requirement, not a stylistic preference. Work on LLM evaluation highlights the need for multi-dimensional measurement beyond model-centric accuracy [7, 9], and ASE work on execution-failure-aware evaluation reinforces our treatment of execution success as a hard gate rather than a secondary metric [12].

7 Concluding Remarks

This paper presented a deployment study of a dynamic intent-to-workflow orchestration system evaluated across 50 intents, 25 components, and eight conditions in both controlled and real-world settings. Strong benchmark performance—80.63% structural similarity, 88% component-selection accuracy, and 92% output quality—did not translate to real deployment, where every automated condition recorded 0% execution success due to orchestration-interface parse failures. The following lessons, grounded in that experience, offer practical guidance for teams deploying similar systems.

- **Interface reliability is the release gate.** Deployment hinges on orchestration-interface robustness, not model capability alone.
- **Use structured outputs.** Schema-constrained generation eliminates parse failures at the orchestration boundary.
- **Treat orchestration as its own failure domain.** Validate and repair plans before any component executes.
- **Deploy in stages.** Start with templates; add dynamic planning only where template coverage is poor.
- **Validity over sophistication.** Execution-ready plans matter more than complex planning features.

Data Availability Statement

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

References

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

[2] S. Bao et al. 2024. WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models. *arXiv preprint arXiv:2411.05451* (2024).

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

[4] Y. Chen et al. 2025. 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)*.

[5] Google Research. 2024. *Opal: Multimodal AI for Creative Tasks*. Technical Report. Google Research.

[6] LangChain Inc. 2023. LangChain: Building applications with LLMs. https://python.langchain.com/.

[7] 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. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).

[8] OpenAI. 2023. Function calling and other API updates. https://openai.com/blog/function-calling-and-other-api-updates. *OpenAI Blog* (2023).

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

[10] N. Peng et al. 2025. JSONSchemaBench: Benchmarking LLMs on JSON Schema-Constrained Generation. *arXiv preprint* (2025).

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

[12] M. Silva et al. 2024. 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)*.

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

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

[15] 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. 2023. A survey on large language model based autonomous agents. *arXiv preprint arXiv:2308.11432* (2023).

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