

GraphBit: A Graph-based Agentic Framework for Non-Linear Agent Orchestration

Yeahia Sarker¹ Md Rahmat Ullah² Musa Molla² Shafiq Joty³

¹Middle Tennessee State University ²InfinitiBit GmbH ³Nanyang Technological University
ys5d@mtmail.mtsu.edu {rahmat.ullah, musa.molla}@infinitibit.com
srjoty@ntu.edu.sg

Abstract

Agentic LLM frameworks that rely on prompted orchestration, where the model itself determines workflow transitions, often suffer from hallucinated routing, infinite loops, and non-reproducible execution. We introduce GraphBit, an engine-orchestrated framework that defines workflows explicitly and deterministically as a directed acyclic graph (DAG). Unlike prompted orchestration, agents in GraphBit operate as typed functions, while a Rust-based engine governs routing, state transitions, and tool invocation, ensuring reproducibility and auditability. The engine supports parallel branch execution, conditional control flow over structured state predicates, and configurable error recovery. A three-tier memory architecture (ephemeral scratch space, structured state, and external connectors) isolates context across stages, preventing cascading context bloat that degrades reasoning in long-running pipelines. Across GAIA benchmark tasks spanning zero-tool, document-augmented, and web-enabled workflows, GraphBit shows excellent performance achieving the highest accuracy, zero orchestration errors, the lowest latency, and the highest throughput. Ablation studies demonstrate that each memory tier contributes measurably to performance with similar configuration due to architectural changes, with deterministic execution providing the greatest gains on tool-intensive tasks representative of real-world deployments.

Code and data: <https://github.com/InfinitiBit/graphbit>

1 Introduction

The emergence of Large Language Model (LLM)-based agents marks a paradigm shift in AI, combining foundation model reasoning capabilities with environmental perception, decision-making, and autonomous action execution (Yao et al., 2022; Ke et al., 2025). Multi-agent systems have rapidly

evolved from research prototypes to production deployments, with applications spanning software engineering (Hong et al., 2023), scientific discovery (Boiko et al., 2023), and enterprise automation (Wu et al., 2024). These systems decompose complex tasks into specialized subtasks assigned to collaborative agents that coordinate to achieve collective goals (Guo et al., 2024). A central challenge in operationalizing these systems is *workflow orchestration*: specifying which agents to invoke, in what order, which tools to employ, and how data propagates across stages. The underlying framework must execute this workflow reliably, efficiently, and reproducibly (Gu et al., 2025; Tran et al., 2025).

Despite their promise, most existing multi-agent frameworks suffer from a fundamental architectural limitation: they rely on *prompted-orchestration* where the LLM itself determines workflow transitions. This includes frameworks such as LangChain (Annam et al., 2025), CrewAI (Duan and Wang, 2024), and AutoGen (Wu et al., 2024), where agents receive natural language descriptions of available tools and downstream agents, then select their next action through in-context learning. This design introduces three critical failure modes (Cemri et al., 2025; Patil et al., 2024): (1) hallucinated routing, where the LLM invents non-existent agents or tools, causing silent failures; (2) infinite loops, where agents repeatedly invoke each other without architectural termination conditions; and (3) non-deterministic execution, where identical inputs produce different traces, undermining auditability in regulated domains. Additionally, each orchestration decision requires a full LLM inference pass, and memory scales with accumulated context, creating inefficiencies that often become acute in enterprise settings with strict latency budgets (Sculley et al., 2015; Barua, 2024).

We present GraphBit, an *engine-orchestrated* agentic framework that performs multi-agent orchestration through a graph-based, non-linear exe-

cution paradigm. In GraphBit, users define workflows as typed directed acyclic graphs (DAGs) specifying agent nodes, tool nodes, and control-flow logic. The framework then executes this workflow deterministically: agents operate as typed functions responsible solely for their domain-specific reasoning, while the execution engine governs all workflow transitions, state management, and tool invocations according to the user-specified graph. This separation ensures that given any workflow, execution remains predictable, auditable, and reproducible regardless of the stochastic nature of LLM outputs (Qiu et al., 2025).

GraphBit’s architecture rests on three foundational principles: (1) *graph-native execution*, where workflows are expressed as DAGs with typed edges representing data dependencies and control flow, enabling parallel execution of independent branches; (2) *engine-governed orchestration*, where the execution engine makes all routing decisions based on explicit conditions, eliminating hallucinated routing and infinite loops by construction; and (3) *hierarchical memory isolation*, where a three-tier memory model segregates ephemeral scratch space, structured workflow state, and external connector interfaces, preventing context pollution. Our Rust-based execution core with Python bindings achieves 11.9 ms mean processing latency and 5,025 operations per minute throughput, a $1.3\times$ improvement over the fastest comparable baseline.

The contributions of this paper are fourfold: (1) an engine-based orchestration architecture with a three-tier memory model where a deterministic execution engine governs all transitions within a user-defined workflow graph, eliminating hallucinated routing by construction; (2) a comprehensive evaluation of seven LLM agent frameworks on a curated 68-task GAIA benchmark subset (Mialon et al., 2023) spanning three workflow types, where all frameworks execute equivalent workflow configurations; (3) demonstration that given equivalent workflows, GraphBit attains 67.6% accuracy with 0% orchestration error rate, demonstrating excellent performance; and (4) ablation studies isolating the contribution of each architectural component to overall performance.

2 Related Work

We situate GraphBit within the landscape of agent architectures, multi-agent frameworks, and workflow orchestration systems.

Agent Architectures. The ReAct paradigm (Yao et al., 2022) established the foundation for modern LLM agents by interleaving reasoning traces with action execution, building on chain-of-thought prompting (Wei et al., 2022; Masterman et al., 2024). Subsequent work extended this through Tree of Thoughts (Yao et al., 2023; Ranaldi et al., 2024) for parallel reasoning paths and Reflexion (Shinn et al., 2023) for self-reflective improvement. Toolformer (Schick et al., 2023) demonstrated that LLMs can learn tool invocation through fine-tuning, while later work showed in-context learning suffices for capable models (Qin et al., 2024, 2023). GraphBit adopts ReAct’s reasoning-action interleaving *within* individual agent nodes, but critically removes the LLM from all inter-node orchestration decisions. Unlike Reflexion’s self-reflective loops, GraphBit’s error recovery is governed by engine-level retry policies rather than LLM self-assessment, ensuring deterministic failure handling.

Multi-Agent Frameworks. MetaGPT (Hong et al., 2023) coordinates agents through standardized operating procedures for software engineering roles. ChatDev (Qian et al., 2024) organizes agents into a virtual software company with defined communication protocols. AutoGen (Wu et al., 2024) provides a conversation-centric framework with natural language agent interaction. LangChain (Annam et al., 2025) and its graph-oriented extension LangGraph (Wang and Duan, 2024) represent the most widely adopted orchestration frameworks. A LangGraph conditional edge is an ordinary function over graph state and can be fully deterministic, so LLM routing there is an application choice, not a framework property. What LangGraph does not remove is the LLM-emitted tool loop *inside* a node; in the configuration we evaluate (Appendix B) that node body is a LangChain AgentExecutor, and it is that loop, not edge routing, that consumes the iteration budget. Recent work on parallel function calling (Kim et al., 2024) formulates tool dispatch as a DAG but does not address multi-agent orchestration. LlamaIndex (Liu, 2022) focuses on retrieval-augmented generation pipelines, while Pydantic AI¹ provides type-safe agent definitions with structured output validation. All these frameworks share a common assumption: the LLM participates in orchestration decisions, which recent empirical analysis has shown leads to systematic failure

¹<https://ai.pydantic.dev/>

modes including task verification gaps and inter-agent misalignment (Cemri et al., 2025). GraphBit departs from this paradigm by restricting the LLM to domain-specific reasoning, delegating all orchestration to a deterministic execution engine. Yu et al. (Yu et al., 2025) introduce DynTaskMAS, a dynamic task-graph framework for LLM-based multi-agent systems that supports adaptive decomposition, parallel execution, context sharing, and workflow optimization. It improves scalability and efficiency in complex, evolving tasks. However, increased orchestration and synchronization overhead may reduce practicality in simpler or latency-sensitive settings.

Workflow Orchestration. Traditional workflow engines such as Apache Airflow (Haines, 2022) and Prefect (Narayanan, 2024) provide deterministic execution but lack native LLM agent support. Temporal² offers durable execution with automatic retry and state persistence but requires substantial integration effort. DSPy (Khattab et al., 2023) compiles declarative language programs into optimized prompts, and LMQL (Beurer-Kellner et al., 2023) introduces constrained LLM generation. These approaches improve individual agent reliability but do not address multi-agent orchestration. Closest to our position, Qiu et al. (Qiu et al., 2025) argue for a “blueprint first, model second” discipline in which the workflow is specified deterministically and the LLM confined to the steps inside it, developed as a design methodology over existing runtimes. GraphBit makes the same separation an execution-level guarantee: the blueprint is a typed DAG, and a dedicated engine rather than the model traverses it. GraphBit complements these techniques by providing a reliable orchestration layer that can incorporate any agent implementation.

3 System Architecture

GraphBit comprises four integrated components: a workflow graph specification, a Rust-based execution engine, a three-tier memory system, and Python bindings for agent development (Figure 1).

3.1 Workflow Graph Specification

Workflows in GraphBit are expressed as directed acyclic graphs (DAGs) where nodes represent computational units and typed edges encode data dependencies and control flow. This graph-native design draws on classical dataflow models (Dennis,

2005) while extending them with LLM-specific node types. Three node types compose to express complex multi-agent workflows. *Agent nodes* encapsulate LLM-based reasoning units, each specifying input/output schemas, a system prompt, and an optional tool set. The execution engine invokes the underlying LLM only when all input dependencies are satisfied, with type enforcement at node boundaries preventing schema violations from propagating through the workflow. *Tool nodes* represent deterministic functions (web search, database queries, API calls) that execute without LLM inference, providing predictable latency. Tool nodes may be composed into subgraphs encapsulating complex retrieval or transformation logic. *Control nodes* implement workflow logic including conditional branching, parallel fan-out, and aggregation. Crucially, control node decisions are evaluated by the execution engine against structured state predicates, not by LLM inference. For example, a conditional branch evaluates a boolean expression over workflow state variables rather than prompting an LLM to select the next step. Edges carry typed data between nodes with automatic serialization for cross-language interoperability, and optional transformation functions enable lightweight preprocessing during data transfer.

3.2 Execution Engine

The execution engine implements a dataflow model (Dennis, 2005) optimized for LLM workloads. Written in Rust for performance and memory safety, the engine maintains a ready queue of nodes whose input dependencies are satisfied, dispatching independent nodes in parallel across a thread pool while executing dependent nodes sequentially with automatic data transfer between stages (Bugden and Alahmar, 2022). The engine enforces correctness invariants that eliminate common failure modes in prompt-orchestrated systems. Termination guarantees arise from the DAG structure: cycles are rejected at graph construction time, and execution progress is tracked to detect stalls. Deterministic routing follows from evaluating control predicates against structured state rather than LLM outputs. Type safety is enforced at node boundaries through runtime schema validation (Polyzotis et al., 2019; Habib et al., 2019), with violations raised as explicit errors. Error handling follows a fail-fast philosophy with configurable recovery policies ranging from immediate termination to automatic retry with exponential backoff, and checkpointing

²<https://temporal.io/>

enables resumption from intermediate states for long-running workflows.

3.3 Three-Tier Memory Architecture

GraphBit’s memory system comprises three isolated tiers designed to prevent context pollution while enabling efficient data access (Zhang et al., 2025). The *ephemeral scratch tier* provides temporary storage for intermediate computations within a single node execution, allocated at node start and deallocated upon completion. This prevents implementation details from leaking across node boundaries, particularly for agents performing iterative chain-of-thought reasoning. The *structured state tier* maintains the canonical workflow context as a typed key-value store with atomic updates upon successful node completion. The engine tracks state provenance for full auditability, and scoped access ensures nodes may only read explicitly declared state keys, preventing implicit dependencies. The *external connector tier* provides managed interfaces to databases, APIs, and file systems with connection pooling, automatic retry, and result caching. Connector results are not automatically injected into agent contexts; nodes must explicitly request external data, preventing context bloat. This three-tier architecture prevents cascading context growth that degrades LLM performance, enables reproducible execution through explicit state management, and simplifies testing through connector abstraction.

3.4 Python Bindings and Agent Development

Agent development occurs in Python through PyO3 bindings, combining Rust’s performance for orchestration with Python’s ecosystem for LLM integration. Developers define agent classes specifying input/output schemas as Pydantic models and implement a run method containing domain-specific logic; the framework handles serialization, context assembly, and LLM invocation. Tool integration follows a similar declarative pattern with support for asynchronous execution and streaming. Workflow composition uses a fluent API that constructs the underlying DAG, with graph structure and type compatibility validated at construction time. Complex workflows may be encapsulated as reusable subgraphs for modular design.

4 Experimental Evaluation

We evaluate GraphBit against six widely-adopted LLM agent frameworks: LangChain (Annam et al.,

2025), LangGraph (Annam et al., 2025), CrewAI (Duan and Wang, 2024), Microsoft AutoGen (Wu et al., 2024), Pydantic AI, and LlamaIndex (Liu, 2022). Our evaluation addresses four research questions: (RQ1) How does GraphBit compare to existing frameworks on task completion accuracy across diverse task types? (RQ2) What are the computational efficiency gains from Rust-based orchestration? (RQ3) Does deterministic orchestration eliminate workflow reliability failures? (RQ4) How does each architectural component contribute to overall performance?

4.1 Experimental Setup

We evaluate on the GAIA benchmark (Mialon et al., 2023), curating 68 of the original 165 tasks by excluding those that *all* seven frameworks consistently failed across three independent trials, as these reflect shared LLM limitations rather than framework differences. The curated set spans three difficulty levels (29 Level 1, 36 Level 2, 3 Level 3) and three workflow types: **zero-tool** (7 tasks), **document-augmented** (19), and **web-enabled** (42). All frameworks use GPT-5.2 with identical sampling parameters, share the same 12 custom tools with `max_iterations=3`, and follow their documented best-practice configurations. We report accuracy (verified by exact match and LLM-based evaluation, Cohen’s $\kappa = 0.87$), orchestration error rate, processing time (framework overhead excluding LLM API latency), CPU utilization, peak memory, and throughput. The two verifiers are scored as separate passes and are not interchangeable: the *primary* pass counts a task correct only when exact match and the LLM judge agree, whereas the *secondary* pass uses the LLM judge alone. Tables 1, 2 and 4 and Appendix G.5 report the primary pass; Table 3 and the *n* column of Table 5 report the secondary pass. Per-framework totals therefore differ between the two families, and every comparison we draw is made within a single pass. Appendices B and D provide complete curation, configuration, and metric details. Appendices B and D provide complete curation, configuration, and metric details.

4.2 Overall Performance Comparison

Table 1 presents the comparative results. GraphBit achieves the highest overall accuracy at 67.6%, a 14.7 percentage point improvement over the strongest baseline (Pydantic AI at 52.9%). Both frameworks completed the same 68 tasks with 17

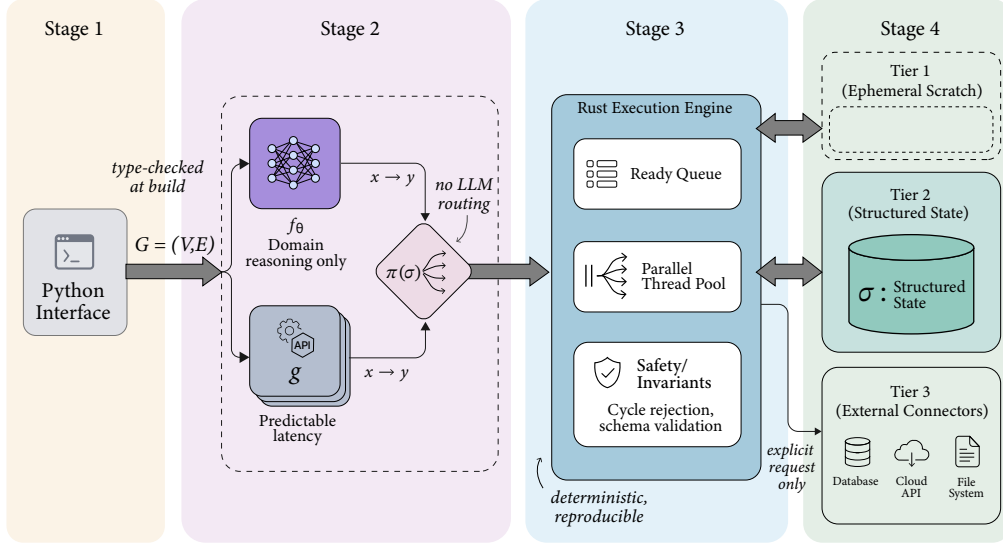


Figure 1: Detailed GraphBit architecture. The Rust-based execution engine traverses a typed workflow DAG, dispatching agent, tool, and control nodes while the three-tier memory system (ephemeral scratch, structured state, external connectors) isolates context across execution stages.

to 7 in GraphBit’s favor. A split that lopsided or better arises by chance about six times in a hundred (McNemar exact $p=0.064$), below the conventional bar, and it does not survive Holm correction over the 21 pairwise framework tests ($p=0.895$). Intervals are correspondingly wide: Wilson 95% CIs of $[55.8, 77.6]$ for GraphBit and $[41.2, 64.3]$ for Pydantic AI, and a bootstrap 95% CI of $[+1.5, +27.9]$ points for the gap. We therefore report GraphBit as the highest-scoring framework here rather than the strongest baseline. GraphBit is the only graph-based framework to achieve a 0% orchestration error rate, shared only with Pydantic AI and LlamaIndex which employ simpler non-routing architectures. Processing latency of 11.9 ms is $1.3\times$ faster than LlamaIndex (15.0 ms) and $5.9\times$ faster than AutoGen (70.0 ms), with throughput of 5,025 ops/min. Memory consumption of 126.1 MB is 24% lower than the closest baseline. These gains stem from the Rust execution engine, which eliminates Python interpreter overhead during orchestration. Under the secondary pass (Table 3) the same comparison reads 70.6% versus 63.2%, a 7.4-point lead, and reorders the baselines beneath Pydantic AI; the sign of the gap is stable across passes, its size is not.

4.3 Performance by Task Type

Table 2 disaggregates accuracy and orchestration error rate by workflow type, revealing three key findings. First, on zero-tool tasks, four frameworks achieve identical 57.1% accuracy with 0% orches-

Framework	Acc. (%)	Orch. (%)	Proc. (ms)	CPU (%)	Mem. (MB)
LangChain	38.2	41.2	36.1	24.7	234.4
LangGraph	36.8	47.1	31.5	26.1	208.0
CrewAI	44.9	14.3	31.0	30.7	202.2
AutoGen	35.3	33.8	70.0	27.0	274.8
Pydantic AI	52.9	0.0	18.3	24.2	166.5
LlamaIndex	50.0	0.0	15.0	25.2	165.4
GraphBit	67.6	0.0	11.9	21.1	126.1

Table 1: Overall performance on 68 curated GAIA tasks.

tration errors, indicating that orchestration differences are less impactful without tool routing. Second, GraphBit ties with LlamaIndex at 68.4% on document-augmented tasks. Third, GraphBit’s advantage is most pronounced on web-enabled tasks (69.0% vs. 54.8% for Pydantic AI), which constitute 61.8% of the evaluation set. The orchestration error analysis reveals that LangGraph exhibits 69.0% orchestration errors on web-enabled tasks, meaning over two-thirds of executions fail due to framework-induced errors. GraphBit’s engine-governed tool invocation eliminates this failure mode entirely.

4.4 Performance by Difficulty Level

GraphBit achieves the highest accuracy on Level 1 (79.3%) and Level 2 (63.9%) tasks, outperforming AutoGen by 20.7 points on Level 1. Prompt-orchestrated frameworks degrade sharply with complexity: LangGraph drops from 48.3% (Level 1) to 27.8% (Level 2), and both LangGraph and Auto-

Framework	No-Tool		Local		Web	
	Acc.	Orch.	Acc.	Orch.	Acc.	Orch.
LangChain	57.1	0.0	57.9	15.8	26.2	59.5
LangGraph	57.1	0.0	63.2	15.8	21.4	69.0
CrewAI	28.6	0.0	52.6	0.0	47.6	16.7
AutoGen	28.6	0.0	42.1	42.1	33.3	35.7
Pydantic AI	28.6	0.0	57.9	0.0	54.8	0.0
LlamaIndex	57.1	0.0	68.4	0.0	40.5	0.0
GraphBit	57.1	0.0	68.4	0.0	69.0	0.0

Table 2: Accuracy (Acc.) and orchestration error rate (Orch.) by task type (%). Best results in **bold**.

Gen reach 0% on Level 3. We report the accuracy–difficulty relationship descriptively rather than inferentially. Pearson correlations over per-task outcomes are negative for LangGraph ($r=-0.26$, $p=0.032$) and AutoGen ($r=-0.27$, $p=0.028$), but rest on a stratum of only three Level-3 tasks and are not robust to its removal ($p=0.091$ and $p=0.117$). We therefore rest the claim on the Level-1 to Level-2 decline, over 29 and 36 tasks: LangGraph falls 20.5 points (48.3% to 27.8%) and AutoGen 19.7 (58.6% to 38.9%), while GraphBit falls 15.4 (79.3% to 63.9%) from a much higher start. Level-3 numbers are reported in (Appendix E). Table 3 in full, and the two correlations quoted above, come from the secondary scoring pass of Section 4.1.

4.5 Workflow Reliability Analysis

We define *orchestration error* as any framework-induced execution failure (routing to non-existent agents, infinite loops, tool invocation failures, or unrecoverable runtime errors), as distinct from *factual hallucination* in LLM outputs, which we do not measure here. The error rates in Table 2 reveal three key findings. First, no framework produces orchestration errors on zero-tool tasks, confirming that pure LLM inference without tool routing is inherently stable. Second, error rates escalate sharply with tool complexity: LangGraph rises from 0% (no-tool) to 15.8% (local) to 69.0% (web). Third, although Pydantic AI and LlamaIndex also achieve 0% orchestration errors, only GraphBit pairs this reliability with the highest accuracy (67.6%): because the execution engine governs all state transitions according to the declared workflow graph, hallucinated routing and execution cycles are impossible by construction, so reliability comes at no cost to reasoning quality. The complete error breakdown is provided in Appendix F.

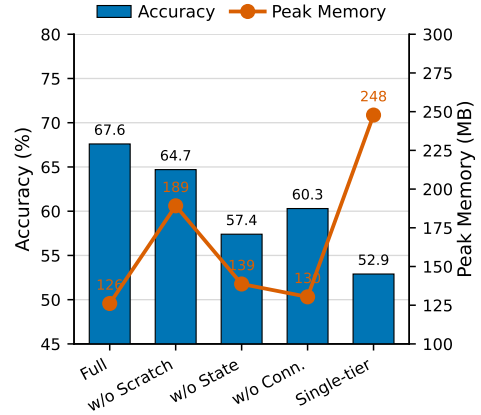


Figure 2: Ablation on the three-tier memory architecture: accuracy (bars, left axis) and peak memory (line, right axis) as each tier is disabled. The single-tier baseline merges all tiers into one shared space.

4.6 Computational Efficiency Analysis

GraphBit achieves the lowest processing latency across all three task types: 6.0 ms on zero-tool tasks (yielding 10,000 ops/min), 10.8 ms on document-augmented tasks, and 13.4 ms on web-enabled tasks. Baselines exhibit steeper scaling; for instance, AutoGen reaches 159.1 ms on document-augmented tasks due to conversation-based orchestration requiring multiple LLM round-trips. Memory scales efficiently from 34.9 MB (no-tool) to 150.5 MB (web), compared with AutoGen’s full 47.0–359.7 MB range; GraphBit is lower on all three task types. Per-task-type efficiency breakdowns are provided in Appendix G.

4.7 Memory Architecture Ablation

We validate our architectural design through ablation experiments on the three-tier memory model. Figure 2 presents results for configurations that selectively disable memory tiers, along with a single-tier baseline that combines all memory into one shared space. Removing ephemeral scratch increases memory by $1.5\times$ and reduces accuracy by 2.9 points, as persisted intermediates degrade reasoning quality (Liu et al., 2024). Disabling structured state yields the largest drop (-10.2 points), confirming its critical role in coherent multi-step reasoning. The external connector tier contributes 7.3 points by preventing context pollution from external data. The single-tier baseline degrades to 52.9% accuracy with $2.0\times$ higher memory, demonstrating that memory segregation is fundamental to GraphBit’s effectiveness. Cross-platform results are in Appendix G.2.

5 Discussion

Our results indicate that orchestration errors, not LLM reasoning quality, are the dominant failure mode in prompt-orchestrated systems: 69.0% of LangGraph’s web-task failures are framework-induced routing errors (Appendix G.5), and frameworks that perform comparably on zero-tool tasks diverge sharply once tool routing is involved. That GraphBit pairs a 0% orchestration error rate with the highest accuracy, lowest latency (6.0–13.4 ms overhead), and a $3.3\times$ reduction in token consumption over the strongest baseline challenges the assumption that LLM-based orchestration is necessary for capable agent systems, suggesting instead that reliability and reasoning quality are complementary rather than competing objectives. The principal trade-off is DAG dependence: execution quality is upper-bounded by the quality of the user-specified graph, and a flawed workflow is faithfully executed as specified. We view this as an inherent consequence of prioritizing execution reliability over planning flexibility, and consider determinism-preserving automatic or hybrid DAG generation the most important direction for future work.

6 Conclusion

We presented GraphBit, an engine-orchestrated agentic framework that expresses multi-agent workflows as typed directed acyclic graphs and delegates all routing, state management, and tool invocation to a deterministic Rust-based execution engine with hierarchical memory isolation. By restricting the LLM to domain-specific reasoning within agent nodes, GraphBit eliminates hallucinated routing and infinite loops by construction while preserving auditability and reproducibility. Our controlled evaluation against representative agent frameworks on curated GAIA tasks, together with ablation studies and a production deployment, shows that engine-governed orchestration substantially improves both reliability and task accuracy over prompt-orchestrated baselines, making the framework well suited to regulated, latency-sensitive settings. Future work will pursue determinism-preserving automatic DAG generation, hybrid deterministic and LLM-based routing, and broader evaluations across domains and models.

Limitations

GraphBit’s determinism stems from executing an explicit, user-defined DAG, so execution quality is upper-bounded by the quality of that graph: a flawed workflow is faithfully executed as specified, and authoring high-quality DAGs shifts engineering effort from prompt design to workflow design. This is a deliberate trade of planning flexibility for execution reliability, and determinism-preserving automatic DAG generation is left to future work. All frameworks share a single proprietary LLM with identical decoding parameters and iteration budgets, which enables a controlled comparison but omits per-framework tuning, and absolute results may shift with other (including open-weight) models. Finally, the production case study covers a single domain at one organization, so the observed gains may not transfer directly to other verticals or workflow structures.

7 Code & Data Availability

Everything needed to reproduce this evaluation will be made available at <https://github.com/InfinitiBit/graphbit>: the 68 GAIA validation task identifiers, curation scripts, all seven configurations, the harness with per-task traces, and the analysis code.

References

- Sangeetha Annam, Merry Saxena, Ujjwal Kaushik, and Shikha Mittal. 2025. Langchain: Simplifying development with language models. *Textual Intelligence: Large Language Models and Their Real-World Applications*, pages 287–304.
- Saikat Barua. 2024. Exploring autonomous agents through the lens of large language models: A review. *arXiv preprint arXiv:2404.04442*.
- Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1946–1969.
- Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578.
- William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*.
- Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt

- Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, and 1 others. 2025. Why do multi-agent llm systems fail? *arXiv preprint arXiv:2503.13657*.
- Jack B Dennis. 2005. First version of a data flow procedure language. In *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, pages 362–376. Springer.
- Zhihua Duan and Jialin Wang. 2024. Exploration of llm multi-agent application implementation based on langgraph+ crewai. *arXiv preprint arXiv:2411.18241*.
- Yang Gu, Hengyu You, Jian Cao, Muran Yu, Haoran Fan, and Shiyu Qian. 2025. Large language models for constructing and optimizing machine learning workflows: A survey. *ACM Transactions on Software Engineering and Methodology*.
- Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xi-angliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.
- Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2019. Type safety with json subschema. *arXiv preprint arXiv:1911.12651*.
- Scott Haines. 2022. Workflow orchestration with apache airflow. In *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*, pages 255–295. Springer.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, Caiming Xiong, and Shafiq Joty. 2025. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *Preprint*, arXiv:2504.09037.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, and 1 others. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.
- Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*.
- Jerry Liu. 2022. [LlamaIndex](#).
- Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paran-jape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173.
- Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. *arXiv preprint arXiv:2404.11584*.
- Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*.
- Pavan Kumar Narayanan. 2024. Orchestrating data engineering pipelines using prefect. In *Data Engineering for Machine Learning Pipelines: From Python Libraries to ML Pipelines and Cloud Platforms*, pages 415–449. Springer.
- Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.
- Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. 2019. Data validation for machine learning. *Proceedings of machine learning and systems*, 1:334–347.
- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2024. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186.
- Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, and 1 others. 2024. Tool learning with foundation models. *ACM Computing Surveys*, 57(4):1–40.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.
- Libin Qiu, Yuhang Ye, Zhirong Gao, Xide Zou, Junfu Chen, Ziming Gui, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Kun Zhao. 2025. Blueprint first, model second: A framework for deterministic llm workflow. *arXiv preprint arXiv:2508.02721*.

- Leonardo Ranaldi, Giulia Pucci, Federico Ranaldi, Elena Sofia Ruzzetti, and Fabio Massimo Zanzotto. 2024. A tree-of-thoughts to broaden multi-step reasoning across languages. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 1229–1241.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.
- David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.
- Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O’Sullivan, and Hoang D Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322*.
- Jialin Wang and Zhihua Duan. 2024. Agent ai with langgraph: A modular framework for enhancing machine translation using large language models. *arXiv preprint arXiv:2412.03801*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.
- Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
- Junwei Yu, Yepeng Ding, and Hiroyuki Sato. 2025. Dyntaskmas: A dynamic task graph-driven framework for asynchronous and parallel llm-based multi-agent systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, pages 288–296.
- Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2025. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47.

A Implementation Details

GraphBit’s execution engine is implemented in approximately 8,000 lines of Rust code, with an additional 2,000 lines of Python bindings generated via PyO3. The core components include a workflow graph representation using the `petgraph` library, a thread pool executor based on `tokio`, and a state management system using `serde` for serialization. Agent nodes interface with LLM providers through a unified abstraction layer supporting OpenAI, Anthropic, and local model deployments. Tool nodes implement an asynchronous execution interface with configurable timeout and retry policies. Control nodes evaluate predicates using a simple expression language supporting boolean operations over typed state variables.

B Experimental Setup Details

Task Curation GAIA (Mialon et al., 2023) is a comprehensive evaluation suite for general AI assistants on real-world tasks requiring multi-step reasoning, tool use, and web navigation. From the original 165 tasks, we curate a subset of 68 tasks by running three independent trials per task across all seven frameworks and excluding tasks where *all* frameworks consistently failed, as these reflect shared LLM limitations rather than framework differences. This yields a discriminative evaluation set where framework architecture measurably impacts outcomes. The curated set spans three difficulty levels: Level 1 contains 29 simple single-step tasks, Level 2 contains 36 moderate multi-step reasoning tasks, and Level 3 contains 3 complex planning tasks requiring extensive tool use.

Workflow Type Segmentation We segment the evaluation into three distinct workflow types that reflect real-world deployment patterns: (1) **zero-tool tasks** (7 tasks) requiring pure LLM reasoning without external tool invocation; (2) **document-augmented tasks** (19 tasks) requiring local tool invocation to process attached files (PDFs, spreadsheets, images); and (3) **web-enabled tasks** (42 tasks) utilizing web search for real-time information retrieval. This segmentation enables fine-grained analysis of framework capabilities across fundamentally different agentic scenarios.

Framework Configurations GraphBit workflows were constructed with the following structure: an initial planning agent determines the task decomposition, followed by parallel execution of

subtask agents, and a final synthesis agent aggregates results. Conditional branches handle error recovery and alternative approaches when initial attempts fail. Each baseline was configured using its documented best practices: LangChain with sequential ReAct chains, LangGraph with state graphs and conditional edges, CrewAI with hierarchical crew structures, AutoGen with two-agent conversation patterns, Pydantic AI with typed agent runs, and LlamaIndex with query engine agents. All frameworks used GPT-5.2 as the underlying LLM with identical temperature and sampling parameters, shared an identical tool set of 12 custom tools. GraphBit and LangGraph both ran at `max_iterations=3` (Appendix C).

C LangGraph Configuration

The 69.0% web-task error rate of Table 4 is a property of this configuration, so we give it in full. A StateGraph is compiled to a graph whose agent node wraps a LangChain AgentExecutor over the shared 12 tools; its edges are plain, and no conditional edge delegates a decision to the model.

```
tools = get_langgraph_gaia_tools() #
      shared 12 tools
agent = create_tool_calling_agent(llm,
      tools, prompt)
node = AgentExecutor(agent=agent, tools
      =tools,
                        max_iterations=3,
                        return_intermediate
                        _steps=True)
graph = StateGraph(BenchmarkState)
graph.add_node("agent", node) #
      node body = tool loop
graph.set_entry_point("agent")
graph.add_edge("agent", END) #
      plain edge, no predicate
```

The budget thus caps how many tool calls the model may *emit* before the executor stops, whereas GraphBit runs the same tools through one `Node.agent` in agent mode at the same budget, with the engine dispatching them. The terminating conditions show the consequence: all 32 LangGraph orchestration errors — 29 of 42 web-enabled tasks, 3 of 19 document-augmented, none on zero-tool — record "Agent stopped due to max iterations.", and none is a routing failure. Web tasks are where the cap binds, needing a search, a fetch, and an extraction to make ordinary progress. Per-task traces recording every emitted call, its iteration index, and the terminating condition will be in the release github link.

D Benchmark Configuration

The primary benchmark experiments were conducted across multiple cloud and bare-metal systems spanning diverse processor architectures: System A (Intel Xeon Skylake, 2 vCPUs, 2 GiB RAM, Ubuntu), System B (AMD EPYC 7571, 2 vCPUs, 2 GiB RAM, Ubuntu), System C (Intel Xeon Skylake, 2 vCPUs, 4 GiB RAM, Windows), System D (AMD EPYC 7571, 2 vCPUs, 4 GiB RAM, Windows), and System E (Apple M1, 8 vCPUs, 16 GiB RAM, macOS). Each framework received equivalent computational resources and API rate limits to ensure fair comparison. All frameworks used the same proprietary LLM with identical temperature (1.0) and max_tokens (2,000) parameters. To validate cross-platform consistency, we additionally conducted ablation experiments across three distinct hardware configurations: Apple Mac M4 (ARM architecture, 16GB RAM, macOS Sequoia), Ubuntu 22.04 on Intel Xeon W-2255 (x86-64, 64GB RAM), and Windows 11 on Intel Core i9-13900K (x86-64, 32GB RAM).

We extend the standard GAIA evaluation protocol with six metrics. Accuracy measures the percentage of correctly completed tasks via dual verification: exact string matching against ground truth and independent LLM-based evaluation using GPT-5.2-chat as an evaluator. Orchestration error rate measures the percentage of failed executions due to framework-induced errors (routing failures, infinite loops, runtime crashes). Processing time reports the mean framework overhead in milliseconds, excluding LLM API latency, measured via the `psutil` library. CPU utilization measures average processor usage during execution. Peak memory reports the highest memory consumption in megabytes. Throughput is computed as `60000/processing_time_ms` operations per minute.

For each framework, we implemented equivalent agent configurations using framework-idiomatic patterns across three workflow types. Zero-tool workflows use direct LLM prompting. Document-augmented workflows employ framework-native tool-calling agents with 12 custom tools including PDF reading (PyPDF2), Excel processing (pandas), image analysis (Donut vision encoder-decoder), audio transcription (Whisper), and code execution. Web-enabled workflows use DuckDuckGo search with BeautifulSoup parsing. Agent execution was limited to `max_iterations=3` across all frameworks

for fair comparison.

Cross-Platform Ablation Setup The ablation experiments were conducted across three hardware configurations to validate cross-platform consistency:

- **Mac M4 (ARM):** Apple Mac Mini M4 with 16GB unified memory, macOS Sequoia 15.1, Rust 1.75.0, Python 3.11.7
- **Ubuntu Intel (x86-64):** Intel Xeon W-2255 with 64GB DDR4 RAM, Ubuntu 22.04.3 LTS, Rust 1.75.0, Python 3.11.7
- **Windows Intel (x86-64):** Intel Core i9-13900K with 32GB DDR5 RAM, Windows 11 Pro 23H2, Rust 1.75.0, Python 3.11.7

Each platform ran identical GraphBit configurations compiled from the same source revision. Python dependencies were pinned to identical versions across all environments using Poetry lock files. LLM API calls were routed through a centralized proxy to ensure consistent network latency measurements.

E Performance by Difficulty Level

Table 3 presents accuracy by GAIA difficulty level for all seven frameworks. GraphBit achieves the highest accuracy on Level 1 (79.3%) and Level 2 (63.9%) tasks. LangGraph drops sharply from 48.3% (Level 1) to 27.8% (Level 2), and both LangGraph and AutoGen reach 0% on Level 3, suggesting that prompt-orchestrated routing compounds errors as task complexity increases.

Framework	Level 1 (n=29)	Level 2 (n=36)	Level 3 (n=3)
LangChain	41.4	33.3	33.3
LangGraph	48.3	27.8	0.0
CrewAI	51.7	63.9	66.7
AutoGen	58.6	38.9	0.0
Pydantic AI	62.1	63.9	66.7
LlamaIndex	62.1	44.4	66.7
GraphBit	79.3	63.9	66.7

Table 3: Accuracy (%) by GAIA difficulty level. Level 1: simple single-step tasks; Level 2: moderate multi-step reasoning; Level 3: complex planning with extensive tool use. Level 3 contains only 3 tasks; results should be interpreted with caution. Counts follow the secondary scoring pass.

F Workflow Reliability Details

Table 4 presents the complete orchestration error rate breakdown by task type for all seven frameworks.

Framework	No-Tool	Local	Web	Overall
LangChain	0.0	15.8	59.5	41.2
LangGraph	0.0	15.8	69.0	47.1
CrewAI	0.0	0.0	16.7	14.3
AutoGen	0.0	42.1	35.7	33.8
Pydantic AI	0.0	0.0	0.0	0.0
LlamaIndex	0.0	0.0	0.0	0.0
GraphBit	0.0	0.0	0.0	0.0

Table 4: Orchestration error rate (%) by task type. Zero-tool tasks produce no orchestration errors across all frameworks. Error rates escalate with task complexity, particularly for prompt-orchestrated frameworks on web-enabled tasks.

G Additional Results

G.1 Complete Computational Efficiency by Task Type

Table 5 presents the complete computational efficiency metrics for all seven frameworks across the three task types, measured on correctly completed tasks. GraphBit achieves the lowest processing latency across all task types. The efficiency advantage is most pronounced on no-tool tasks (6.0 ms, $3.9\times$ faster than AutoGen) and document-augmented tasks (10.8 ms, $14.7\times$ faster than AutoGen at 159.1 ms). AutoGen’s exceptionally high processing time on document-augmented tasks reflects its conversation-based orchestration pattern, which requires multiple LLM round-trips for tool coordination (Kim et al., 2024).

G.2 Cross-Platform Consistency

Table 6 presents GraphBit’s performance across three hardware platforms to validate cross-platform consistency of the Rust-based execution engine.

Accuracy variations remain within 0.5 percentage points across platforms, confirming that our architectural advantages are not artifacts of a specific runtime environment. The Mac M4’s unified memory architecture yields 6% lower memory consumption than Ubuntu Intel, while Windows exhibits marginally higher overhead due to additional runtime dependencies.

G.3 Token Efficiency Analysis

Table 7 reports token consumption patterns across frameworks. Token efficiency directly impacts API cost and latency in production deployments.

GraphBit consumes 1,916 mean total tokens per task, $3.3\times$ fewer than Pydantic AI (6,276) and $7.1\times$ fewer than CrewAI (13,638). This efficiency stems from GraphBit’s structured state management, which avoids the context accumulation patterns of conversation-based frameworks. CrewAI’s high token consumption reflects its verbose agent interaction protocol, where backstory and role definitions are repeated across multiple agent turns.

G.4 Execution Time Distribution

Table 8 characterizes the execution time distribution across frameworks, providing insight into predictability and tail latency behavior.

End-to-end execution time is dominated by LLM API latency rather than framework processing overhead. LlamaIndex and LangGraph achieve the lowest mean execution times (27.9 s and 27.4 s), though their lower times partly reflect fewer successful tool invocations on failed tasks. CrewAI exhibits the highest variance (80.8 s standard deviation) and worst tail latency (347.0 s maximum), attributable to its multi-agent conversation protocol which can trigger extended deliberation chains. GraphBit’s P95 latency of 115.7 s is higher than LlamaIndex (70.5 s), reflecting GraphBit’s more thorough task processing enabled by successful tool chains on complex tasks.

G.5 Error Analysis

Of the 22 tasks (32.4%) where GraphBit failed to produce correct answers, manual analysis reveals the following distribution: 50% involved factual errors in LLM reasoning where the model generated incorrect intermediate conclusions; 30% involved misinterpretation of task requirements, particularly ambiguous natural language specifications; 15% involved tool execution failures such as web search returning no relevant results; and 5% involved output formatting errors where correct reasoning produced incorrectly formatted final answers. Critically, zero failures were attributable to orchestration errors, confirming that GraphBit’s execution engine operates correctly across all evaluated scenarios. In contrast, baseline framework failures exhibit a bimodal distribution between orchestration errors and reasoning errors. For LangGraph,

Task Type	Framework	Proc. (ms)	CPU (%)	Mem. (MB)	Thpt. (ops/m)	<i>n</i>
No-Tool	GraphBit	6.0	15.2	34.9	10,000	5
	LangChain	7.6	18.2	86.7	7,887	4
	CrewAI	9.4	17.3	58.1	6,353	3
	LlamaIndex	11.2	20.3	114.6	5,361	5
	Pydantic AI	12.0	19.0	186.2	5,002	2
	LangGraph	16.9	18.3	106.2	3,550	4
	AutoGen	23.4	19.3	47.0	2,562	4
Local Tool	GraphBit	10.8	24.7	105.5	5,543	11
	LlamaIndex	15.3	29.6	160.9	3,926	13
	CrewAI	16.1	26.4	151.5	3,728	8
	Pydantic AI	16.3	26.5	135.5	3,691	12
	LangChain	41.8	30.1	200.5	1,435	10
	LangGraph	46.3	33.7	199.6	1,295	11
	AutoGen	159.1	32.7	171.3	377	7
Web Search	GraphBit	13.4	20.5	150.5	4,466	32
	LlamaIndex	15.5	24.0	175.9	3,868	18
	Pydantic AI	20.4	24.1	177.3	2,948	29
	LangGraph	27.3	24.0	228.8	2,201	9
	CrewAI	34.6	32.9	226.2	1,736	29
	AutoGen	37.5	25.7	359.7	1,599	20
	LangChain	38.3	23.4	274.4	1,567	11

Table 5: Complete computational efficiency metrics by task type (measured on correctly completed tasks only). *n*: number of correctly completed tasks. Frameworks sorted by processing time within each task type. *n* follows the secondary scoring pass.

Platform	Acc. (%)	Mem. (MB)	Proc. (ms)
Mac M4 (ARM)	67.4	118.3	11.7
Ubuntu Intel (x86)	67.6	126.1	11.9
Windows Intel (x86)	67.9	132.8	12.4
<i>Std. Dev.</i>	<i>0.3</i>	<i>7.3</i>	<i>0.4</i>

Table 6: Cross-platform consistency of GraphBit across Mac M4 (ARM), Ubuntu Intel, and Windows Intel. Accuracy variations remain within 0.5 percentage points.

Framework	Prompt	Compl.	Total	TPS
GraphBit	1,735	181	1,916	12.9
Pydantic AI	6,004	273	6,276	17.0
CrewAI	11,306	2,332	13,638	33.2

Table 7: Mean token consumption per task. Prompt: mean prompt tokens; Compl.: mean completion tokens; Total: mean total tokens; TPS: tokens per second. Frameworks with incomplete token reporting are omitted.

69.0% of web-search task failures are orchestration-induced (hallucinated tool routing) (Patil et al., 2024), while only 31.0% are reasoning errors. This finding suggests that a substantial fraction of baseline failures could be eliminated through deterministic orchestration alone, without requiring improvements to the underlying LLM.

Framework	Mean (s)	Med. (s)	P95 (s)	Std. (s)	Max (s)
LlamaIndex	27.9	24.5	70.5	23.4	95.1
LangGraph	27.4	17.3	73.9	25.2	84.9
GraphBit	35.3	28.8	115.7	34.8	140.0
Pydantic AI	38.7	29.5	101.8	39.7	213.8
LangChain	45.1	37.5	119.8	42.4	168.1
AutoGen	54.9	53.1	142.6	49.2	175.0
CrewAI	75.8	50.7	217.5	80.8	347.0

Table 8: End-to-end execution time distribution across all 68 tasks (including LLM API latency). Med.: median; P95: 95th percentile; Std.: standard deviation.

G.6 Framework Component Overhead

Table 9 reports framework initialization overhead, which impacts cold-start latency in serverless deployments. CrewAI incurs the highest import overhead (5,700 ms) due to its extensive dependency chain, while Pydantic AI achieves the lowest (2,100 ms). AutoGen’s setup time (23.6 ms) is substantially higher than other frameworks due to its agent initialization protocol. GraphBit’s combined initialization of 2,400 ms is competitive despite the overhead of loading Rust bindings via PyO3.

H Deployment Case Study

GraphBit has been deployed in a production document processing pipeline at a financial services firm

Framework	Import (ms)	Setup (ms)
GraphBit	2,400	0.1
LangChain	3,200	0.9
LangGraph	2,600	0.9
LlamaIndex	2,900	0.2
Pydantic AI	2,100	1.2
AutoGen	3,700	23.6
CrewAI	5,700	0.1

Table 9: Framework initialization overhead. Import: Python module import time; Setup: framework-specific initialization after import.

for automated regulatory compliance report generation. The deployment processes unstructured financial documents (10-K filings, quarterly reports, risk disclosures) through a 15-node workflow DAG that extracts key metrics, cross-references regulatory requirements, and generates structured compliance summaries.

Metric	GraphBit	Prior System
End-to-end accuracy (%)	94.2	78.6
Orchestration failures	0	127
Mean processing time (s)	12.4	38.7
Documents processed/week	~1,200	~1,200
Deployment duration (weeks)	12	–
Total executions	14,400+	–

Table 10: Production deployment comparison: GraphBit vs. prior LangChain-based pipeline over a 12-week pilot (January–March 2026).

Table 10 summarizes key metrics from the 12-week pilot deployment. GraphBit achieved zero orchestration failures across 14,400+ executions, compared to 127 failures in the prior LangChain-based system. End-to-end accuracy improved from 78.6% to 94.2%, and mean processing time decreased by $3.1\times$. Operational lessons include: (1) DAG versioning proved essential for regulatory audit trails, as each workflow version can be precisely linked to its outputs; (2) the three-tier memory architecture simplified debugging by providing complete state provenance for each execution; (3) cold-start latency of ~ 2.4 s required pre-warming strategies in the serverless deployment environment. Bringing the pipeline into production cost approximately 80 developer-hours in total, covering authoring and review of the 15-node DAG, integration of the document and regulatory-lookup tools, and validation of its outputs.

I SWE-bench Lite Evaluation

To validate generalization beyond GAIA, we evaluate GraphBit on a subset of 30 tasks from SWE-bench Lite (Jimenez et al., 2023), adapted for multi-agent evaluation. Each task requires resolving a real-world GitHub issue through code analysis, patch generation, and test validation.

Framework	Resolution Rate (%)
LangGraph	46.7
CrewAI	53.3
Pydantic AI	56.7
GraphBit	63.3

Table 11: Resolution rate on 30 SWE-bench Lite tasks. GraphBit outperforms all baselines, with the largest gains on multi-file issues requiring coordinated tool use.

GraphBit achieves a 63.3% resolution rate (Table 11), outperforming LangGraph by 16.6 percentage points. The advantage is most pronounced on multi-file issues requiring coordinated analysis across multiple code files, where deterministic orchestration ensures systematic coverage of all relevant files.