Architectural Comparison of Multi-Agent Frameworks: Communication Protocols, State Synchronization, and Performance Footprints

1. Executive Synthesis: Architectural Paradigms and Performance Footprints

The design and performance of multi-agent systems are fundamentally dictated by their choice of inter-agent communication and state synchronization mechanisms. The analysis of LangGraph, CrewAI, and AutoGPT reveals three distinct architectural paradigms: deterministic Shared State, flexible Message Passing, and artifact-centric Shared Memory.

1.1. Core Findings Summary

A direct comparison confirms that LangGraph utilizes a centralized Shared State model, CrewAI primarily employs a conversational Message Passing protocol, and AutoGPT relies on Shared Memory through project artifacts and specialized Inter-Agent Communication (IAC).[1]

Regarding the specific query concerning explicit, measured state synchronization latency in the millisecond (ms) range, public benchmarks for the internal synchronization mechanisms of these core orchestrators are generally unavailable. The core constraint in all LLM-driven agent systems is not the internal mechanism's synchronization speed, but the latency associated with Large Language Model (LLM) inference.[4] LLM Time-to-First-Token (TTFT) and total inference time typically span hundreds of milliseconds to multiple seconds, effectively dominating the system's overall execution time.[4]

However, comparative throughput metrics serve as a robust proxy for inherent architectural efficiency and overhead.[6] LangGraph consistently demonstrates the highest task throughput due to its deterministic graph structure, while CrewAI and AutoGPT show moderate to lower throughput, respectively, reflecting the computational cost of managing coordination complexity via conversational parsing or autonomous planning.

1.2. Comparative Agent Framework Architecture

The following table summarizes the foundational differences in how these three leading frameworks manage context and agent coordination.

Comparative Agent Framework Architecture

Framework	**Primary Communication Paradigm**	**Core State Management Mechanism**	**Architectural Implication (Scalability)**
LangGraph	Shared State (Explicit Mutation)	State Machine/GraphState (Centralized context) [1]	High control flow determinism; superior local throughput but potential write contention bottleneck in horizontal scaling.
CrewAI	Message Passing (Conversational)	Task Outputs / Crew Context / Flows (Distributed coordination) [2]	Flexible collaboration; latency sensitive to message parsing and conversational length.[9]
AutoGPT	Shared Memory / IPC (Abstracted)	Artifact-centric File System or Knowledge Base [3]	Optimized for autonomous, iterative goal

			achievement; high planning overhead; coordination tied to memory consistency.

2. Theoretical Foundations: State Synchronization in LLM MAS

The architectural decision between Message Passing (MP) and Shared State (SS) defines where computational and I/O bottlenecks will reside within a Multi-Agent System (MAS). This choice influences determinism, scalability, and the system’s overall tolerance for agent failures.

2.1. Message Passing (MP): Decentralized Coordination

Message Passing is the cornerstone of truly distributed systems and is heavily utilized in conversational frameworks like CrewAI. In this paradigm, agents maintain their private local state and coordinate exclusively by exchanging structured or natural language messages through communication channels such as asynchronous messaging queues or intermediate output exchanges.[2] The system achieves coordination not through a central context update, but through structured dialogue and the resulting sequence of intermediate outputs passed between agents.[11]

The inherent advantage of MP is high horizontal scalability and fault isolation; the failure of one agent does not immediately corrupt a centralized state required by all others. However, the system's efficiency and responsiveness are critically dependent on the conversational complexity. Coordination requires the LLM to parse incoming messages, determine relevance, formulate a response, and decide on the next step. This increases coordination complexity rapidly as the number of agents grows, resulting in conversational overhead and non-deterministic logic. This dependency on LLM dialogue for flow control introduces significant latency attributable to the LLM's reasoning time, impacting overall system throughput.[9]

2.2. Shared Memory/Shared State (SS): Centralized Determinism

The Shared State (SS) model, exemplified by LangGraph's State Machine design, involves a singular, globally accessible state object, often called the $GraphState$.[1] Agents, acting as nodes in the graph, read this current state, execute their function, and return explicit, computable updates which are merged back into the central state.[7]

This approach offers significant benefits in terms of deterministic control flow, conditional branching, and checkpointing, features crucial for reliable production systems and observable execution.[13] Because the graph logic manages all transitions, the LLM is primarily used only to execute the designated task or make simple transition decisions, minimizing the cognitive load associated with workflow management.[14] This fundamental structural efficiency allows LangGraph to achieve superior throughput (42 tasks/min) compared to frameworks that rely on conversational complexity for flow control (CrewAI, 34 tasks/min).[6]

However, the primary performance bottleneck in SS shifts away from conversational parsing to concurrency management and I/O overhead. When parallel paths (fan-out) share the same state [1], managing simultaneous write attempts while ensuring strong consistency requires robust internal mechanisms (like atomic merges or locking). Furthermore, maintaining durability requires external state persistence, where the latency introduced by necessary serialization and I/O round trips becomes the dominant delay factor.[15]

3. Architectural Deep Dive I: LangGraph – The State Machine Model

LangGraph provides a high-control architecture rooted in the finite state machine concept, which dictates both workflow logic and state propagation.

3.1. LangGraph's Core Shared State Mechanism ($GraphState$)

The $GraphState$ serves as the explicit and central source of truth for the entire workflow.[1] Nodes in a LangGraph execution are functions that receive the current state and perform computations or call tools. For example, a tool output, like a user's date of birth, is wrapped into a specific update dictionary which is then merged into the $GraphState$, allowing the new information to be immediately visible to all subsequent nodes.[7]

The model's ability to handle parallel paths (fan-out and fan-in) sharing the identical state [1] is what guarantees that subsequent nodes receive complete context, regardless of which parallel branch finished first. This architectural feature ensures strong consistency and observability, making complex, multi-step reasoning easier to debug and verify. The system's high throughput is directly correlated with this high degree of structural predictability, where the framework handles the complex flow control, enabling the LLM to focus purely on the task execution at hand. This circumvents the lengthy, non-deterministic conversational overhead required by message-passing paradigms for flow management.[14]

3.2. State Persistence Overhead and Practical Latency

While internal memory access and Inter-Process Communication (IPC) for state updates are rapid (sub-millisecond), production deployments of LangGraph necessitate external state persistence to maintain conversation memory, enable failure recovery, and allow for state checkpointing.[13]

This architectural requirement immediately introduces an I/O constraint. Implementing persistence layers, such as asynchronous MongoDB saving, has been shown to demonstrably increase system latency during stress testing.[15] This observation confirms that the efficiency advantage gained by LangGraph's deterministic orchestration is partially constrained by the mandatory I/O round trips required to maintain strong consistency and resilience through checkpointing. The speed advantage of the execution logic becomes highly dependent on the performance of the external state store, not the internal communication speed.[16]

In practice, if a scalable deployment relies on a networked, distributed in-memory store like Redis for persistence, the intrinsic latency of that system sets a performance floor. Redis, even in optimized configurations, is documented to have an intrinsic latency around **9.7 milliseconds** (9671 microseconds).[17] This figure represents a practical lower bound for state synchronization latency when durable state persistence across a network is required for long-running or complex $GraphState$ workflows. Thus, the system’s overall latency is dictated by infrastructure network performance and database round-trip time (RTT), not the computation of agent logic.

4. Architectural Deep Dive II: CrewAI – The Collaborative Team Model

CrewAI operates on a collaborative team metaphor, leveraging the Message Passing (MP) paradigm to facilitate nuanced agent interactions and delegation.

4.1. Message Passing and Conversational Interplay

CrewAI agents are defined by specific roles, goals, and backstories, fostering conversational intelligence.[11] Inter-agent communication is achieved through dialogue, where specialized agents exchange messages or intermediate outputs. This interaction is mediated through distributed communication channels, which can include asynchronous messaging queues or shared memory buffers for output exchange, conforming to a typical distributed agent MP model.[2]

Performance analysis of CrewAI systems often points to bottlenecks arising from communication overhead and inefficient task scheduling.[9] Consequently, performance tuning efforts focus on mitigating these delays by implementing strategies such as optimizing the protocol used for inter-agent communication, utilizing lightweight message formats, and reducing the frequency of status updates.[9] The necessity for these communication optimizations underscores that message parsing, transmission, and the subsequent LLM reasoning time required to process and act on those messages, constitute a major latency factor in CrewAI's performance profile (34 tasks/min).[6]

4.2. State Management through Tasks and Flows

In standard CrewAI processes (sequential or hierarchical), state is implicitly managed through the accumulation of task outputs. Agents collaborate by completing tasks that produce "actionable results," which then become the working context fed into the subsequent agent's prompt.[11]

A significant architectural development in CrewAI is the introduction of $Flows$. $Flows$ overlay the conversational core with a structured, event-driven architecture designed to streamline workflow creation and, critically, to explicitly manage and share state between different tasks.[8] This feature represents an architectural convergence, demonstrating a recognition that pure, unstructured message passing is often insufficient for applications requiring high precision and reproducibility.[19] By introducing a controlled state layer, $Flows$ allow developers to utilize the collaborative flexibility of the MP paradigm while gaining some of the structured, deterministic state management advantages characteristic of the SS model.

The observed lower throughput of CrewAI relative to LangGraph [6] is a systemic cost imposed by its conversational design. CrewAI mandates that the LLM performs conversational coordination—parsing messages, interpreting intent, and determining the next step in the dialogue. This process adds significant cognitive latency (LLM reasoning time) to the execution path, lowering the overall efficiency of task processing when compared to the highly deterministic, orchestration-minimized execution path of a state machine framework.

5. Architectural Deep Dive III: AutoGPT and Autonomous Goal Systems

AutoGPT's architecture is optimized for maximizing agent autonomy and achieving complex, open-ended goals through iterative planning and persistent memory, placing it within the Shared Memory category.

5.1. Communication via Shared Artifacts and IAC Protocol

The foundational mechanism of AutoGPT centers on an iterative loop of perception, planning, and action, driven toward a user-defined goal.[20] Coordination between specialized agents (like Planners and Coders) is often facilitated through an explicit Shared Memory or knowledge base.[3] This memory is not merely transient context but is designed as an artifact-centric file system—often an in-memory dictionary—where keys are project files (e.g., "main.py") and values are the content.[3] Agents synchronize their understanding of the project's current state by reading and writing to this central repository, making it an explicit Shared Memory mechanism.

The modern AutoGPT architecture (Agent Core/Forge) utilizes a layered and scalable communication protocol called Inter/Intra-Agent Communication (IAC). This protocol manages secure and seamless interactions between agents and their orchestrator, drawing inspiration from low-level operating system Inter-Process Communication (IPC) mechanisms.[10] IAC manages the physical communication layer, abstracting the process of reading and writing the high-level shared artifacts that constitute the agent's operational state.[3]

5.2. Scalability Profile and Latency Constraints

AutoGPT registers the lowest throughput among the three major frameworks, benchmarking at 28 tasks/min.[6] This lower performance is a direct consequence of the system's design priority: maximum autonomy requires high overhead in planning, self-reflection, and iterative refinement.[21] The success rates in complex tasks are heavily dependent on the quality and performance of the underlying LLM (GPT-4 having shown superior performance over other models in AutoGPT settings).[5]

The performance penalty observed in AutoGPT is attributed primarily to this "cognitive overhead" required for autonomous decision-making. The agents spend significant time in internal reasoning and planning, leading to longer overall execution times. While the IAC protocol may ensure efficient IPC for the underlying message transmission [10], the process of reading and ensuring consistency across large, shared artifacts—such as project code or complex design outlines—within the Shared Memory layer introduces synchronization latency. This delay is necessary to guarantee that all agents perceive the latest, consistent version of the shared state before executing the next action. The latency is therefore linked to the integrity of the shared knowledge base, rather than simple message transfer speed.

6. Quantitative Analysis: Latency, Throughput, and Architectural Overhead

The performance comparison of these frameworks reveals a clear relationship between architectural determinism, orchestration efficiency, and resulting throughput, which serves as the most effective metric for comparative architectural overhead.

6.1. Comparative Throughput Benchmarking

The following table synthesizes the available throughput data and contextualizes the latency constraint.

Comparative Performance and Latency Context

Framework	Measured Task Throughput (Tasks/Minute)	Primary Communication Paradigm	Explicit State-Sync Latency (ms)	Dominant Performance Constraint
LangGraph	42 (Highest) [6]	Shared State (Deterministic Flow)	Not Explicitly Stated	State persistence I/O overhead (e.g., MongoDB RTT).[15]
CrewAI	34 (Moderate) [6]	Message Passing (Conversational)	Not Explicitly Stated	LLM reasoning time for message parsing and coordination.[9]
AutoGPT	28 (Lowest) [6]	Shared Memory (Artifact-Centric)	Not Explicitly Stated	Iterative planning, goal decomposition, and LLM cognitive overhead.[5]

6.2. Contextualizing Latency: LLM vs. Orchestration Overhead

In modern LLM agent systems, the latency introduced by internal orchestration mechanisms (state synchronization, message parsing) is effectively overshadowed by the time consumed by the LLM inference round trip.[4]

For the desired internal synchronization latency (ms range), two scenarios exist:

1. **In-Memory/Local IPC:** If state synchronization occurs strictly in-memory or via local IPC (the core synchronization loop without persistence), the latency is sub-millisecond and

practically negligible.

2. **Networked Persistence/Distributed Memory:** When high availability and long-running context necessitate external storage, the latency floor is dictated by networking I/O. As established, even highly optimized distributed caches like Redis introduce an intrinsic latency around **9.7 ms**.[17] This figure sets the minimum network round-trip time required to checkpoint or synchronize state, confirming that in scalable production environments, architectural efficiency is constrained by infrastructure overhead, not by the chosen communication protocol's inherent speed.[16]

6.3. The Architectural Impact on Latency Mitigation

The throughput metrics demonstrate a clear correlation between architectural predictability and efficiency. LangGraph's superior throughput (42 tasks/min) stems from its structural determinism, which allows for sophisticated latency mitigation strategies. The static workflow graph enables techniques like **speculative execution**, where verification and downstream execution steps can be overlapped. This mechanism effectively hides the latency introduced by the LLM verifier, resulting in substantial reductions in mean execution time (T_{exec}) and verification time (T_{vrf}).[22] LangGraph minimizes the number of expensive LLM calls dedicated solely to flow control.

In contrast, CrewAI’s message-passing architecture requires mitigation strategies focused on optimizing the communication protocol itself. Performance is gained by reducing the computational load required for inter-agent communication and minimizing message parsing delays, streamlining the interaction between lengthy LLM calls.[9] For AutoGPT, the lowest throughput reflects the necessity of trading execution speed for maximum autonomy. The dominant latency factors are the time spent in iterative planning and self-reflection [5], meaning the system is primarily constrained by its own cognitive overhead rather than orchestration delays.

7. Strategic Trade-Off Analysis and Deployment Scenarios

The choice of framework must align with the required balance between workflow determinism, collaboration complexity, and agent autonomy, as these factors directly dictate the resulting system latency and throughput.

7.1. Decision Matrix: Matching Architecture to Scalability Needs

The performance data strongly suggests that engineers must select between orchestration speed and agent flexibility.

- **LangGraph (Deterministic Scaling):** This framework is ideal for scenarios demanding high throughput, predictable control flow, and mission-critical reliability. Examples include structured data pipeline orchestration, automated financial transaction verification, or systems with explicit conditional logic. Its challenge lies in managing centralized state consistency efficiently under highly parallel fan-out conditions.
- **CrewAI (Collaborative Scaling):** Best suited for complex, team-based tasks that require human-like collaboration and natural language dialogue, such as research, content generation, or strategic planning.[11] While its cumulative latency is higher due to conversational overhead, its structure allows for excellent horizontal scaling via distributed messaging queues and team delegation capabilities.[2]
- **AutoGPT (Autonomous Scaling):** The appropriate choice for highly exploratory or open-ended problems where the highest priority is autonomous goal achievement and iterative refinement (e.g., complex code generation or novel task decomposition).[3] High latency is a necessary cost of achieving maximum goal-seeking autonomy.

7.2. Resilience and Consistency: Shared State vs. Message Queueing

Resilience mechanisms introduce specific latency penalties determined by the architecture.

In the Shared State model (LangGraph), resilience and state consistency are maintained through mandatory **atomic checkpointing** (state persistence).[13] If a node fails, the workflow can roll back to the last consistent checkpoint. As discussed, the latency cost is incurred on every checkpoint write, which is necessary to ensure the deterministic state machine remains recoverable.[15]

In the Message Passing model (CrewAI), resilience often relies on underlying **persistent message queues** and pub/sub mechanisms (potentially provided by systems like Dapr).[2] State is implicitly contained in the guaranteed backlog of unread messages. This ensures eventual delivery and tolerance against individual agent failures, but requires robust management of message ordering and introduces queuing and parsing overheads into the execution path.[9]

The architectural focus is rapidly evolving, driven by the need for standardization. Protocols such as the Inter/Intra-Agent Communication (IAC) used by AutoGPT [10] and the proposed Agent Network Protocol (ANP) [24] indicate a market necessity for standardized, low-latency, API-native communication protocols. This suggests that regardless of their internal core (SS or MP), future agent frameworks will need to adopt rigorous, standardized messaging systems to ensure cross-platform interoperability and enhanced performance.

8. Conclusion and Recommendations

The analysis confirms three distinct architectural approaches to multi-agent coordination, each resulting in predictable performance characteristics. LangGraph employs a **Shared State** model, maximizing control flow determinism and achieving the highest measured throughput (42 tasks/min). CrewAI utilizes **Message Passing** via conversational dialogue and structured task outputs, offering collaborative flexibility at a moderate throughput (34 tasks/min). AutoGPT leverages a **Shared Memory** approach through artifact management and the IAC protocol, optimizing for deep autonomy but yielding the lowest throughput (28 tasks/min).

Explicit, measured internal state-sync latency for the core orchestration loop is not publicly benchmarked in the target frameworks. However, the system's performance ceiling is universally constrained by LLM inference time (seconds). For production deployments requiring distributed state persistence, the state synchronization latency floor is set by network I/O, realistically constrained to the $O(10 \text{ ms})$ range, exemplified by the intrinsic latency of optimized distributed stores like Redis.[17]

The primary difference in framework performance stems from how efficiently the architecture orchestrates the expensive LLM calls. LangGraph's deterministic structure allows for latency hiding through speculative execution, maximizing its efficiency. CrewAI's conversational approach requires the LLM to manage coordination, introducing conversational overhead and latency.

Recommendations for Framework Selection:

1. **For High-Throughput, Deterministic Workflows:** Select **LangGraph**. Its state machine architecture ensures maximum predictability, allowing for superior efficiency by minimizing LLM involvement in flow control and enabling advanced latency mitigation techniques like speculative execution.[6] The focus should be on optimizing the external I/O layer for persistence.

2. **For Complex, Collaborative Team Tasks:** Choose **CrewAI**. This framework excels when human-like delegation and conversational refinement are necessary.[11] Performance tuning should prioritize optimizing the message protocols and reducing the LLM's cognitive load in parsing verbose message chains.[9]
3. **For Autonomous, Goal-Seeking Exploration:** Utilize **AutoGPT**. Recognize that the lower throughput is a trade-off for maximizing agent autonomy and iterative refinement capabilities. Performance improvements will hinge primarily on integrating state-of-the-art LLM inference frameworks (like vLLM or TGI) for handling AI workloads [23] and enhancing the efficiency of the IAC artifact synchronization layer.

Works cited

1. langgraph/how-tos/branching/ #540 - GitHub, accessed November 6, 2025, https://github.com/langchain-ai/langgraph/discussions/540
2. AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges - arXiv, accessed November 6, 2025, https://arxiv.org/html/2505.10468v1
3. AgentMesh: A Cooperative Multi-Agent Generative AI Framework for Software Development Automation - arXiv, accessed November 6, 2025, https://arxiv.org/html/2507.19902v1
4. FairBatching: Fairness-Aware Batch Formation for LLM Inference - arXiv, accessed November 6, 2025, https://arxiv.org/html/2510.14392v1
5. Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions - arXiv, accessed November 6, 2025, https://arxiv.org/pdf/2306.02224
6. AI Agent Frameworks: LangGraph vs CrewAI vs AutoGPT - KodekX, accessed November 6, 2025, https://www.kodekx.com/blog/ai-agent-frameworks
7. Tool calling in LangGraph and how to update the Graph-State with ..., accessed November 6, 2025, https://github.com/langchain-ai/langgraph/discussions/1616
8. Flows - CrewAI Documentation, accessed November 6, 2025, https://docs.crewai.com/en/concepts/flows
9. CrewAI Performance Tuning: Optimizing Multi-Agent Systems - Wednesday Solutions, accessed November 6, 2025, https://www.wednesday.is/writing-articles/crewai-performance-tuning-optimizing-multi-agent-systems
10. kevin-rs/autogpt: A Pure Rust Framework For Building AGI (WIP). - GitHub, accessed November 6, 2025, https://github.com/kevin-rs/autogpt
11. Introduction - CrewAI, accessed November 6, 2025, https://docs.crewai.com/en/introduction
12. Agents - CrewAI Documentation, accessed November 6, 2025, https://docs.crewai.com/en/concepts/agents
13. Mastering LangGraph: Building Complex AI Agent Workflows with State Machines, accessed November 6, 2025, https://jetthoughts.com/blog/langgraph-workflows-state-machines-ai-agents/
14. AutoGen vs CrewAI vs LangGraph: AI Framework Comparison 2025 - JetThoughts, accessed November 6, 2025,

https://jetthoughts.com/blog/autogen-crewai-langgraph-ai-agent-frameworks-2025/
15. Persistence to MongoDB slows quite alot · langchain-ai langgraph · Discussion #2895, accessed November 6, 2025, https://github.com/langchain-ai/langgraph/discussions/2895
16. Why is SQLite faster than Redis in this simple benchmark? - Stack Overflow, accessed November 6, 2025, https://stackoverflow.com/questions/11216647/why-is-sqlite-faster-than-redis-in-this-simple-benchmark
17. Diagnosing latency issues | Docs - Redis, accessed November 6, 2025, https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/latency/
18. Tasks - CrewAI Documentation, accessed November 6, 2025, https://docs.crewai.com/en/concepts/tasks
19. Evaluating Use Cases for CrewAI, accessed November 6, 2025, https://docs.crewai.com/en/guides/concepts/evaluating-use-cases
20. What is AutoGPT? - IBM, accessed November 6, 2025, https://www.ibm.com/think/topics/autogpt
21. \tool: Customizable Runtime Enforcement for Safe and Reliable LLM Agents - arXiv, accessed November 6, 2025, https://arxiv.org/html/2503.18666v1
22. instead, these tasks are decomposed into multiple LLM calls (ie, subtasks) involving tool use, retrieval, and aggregation. This decomposition results in an agentic workflow, naturally modeled as a graph where nodes represent operations (eg, API call or LLM inference) and edges capture information flow and dependencies. - arXiv, accessed November 6, 2025, https://arxiv.org/html/2511.00330v1
23. panaversity/learn-agentic-ai: Learn Agentic AI using Dapr Agentic Cloud Ascent (DACA) Design Pattern and Agent-Native Cloud Technologies - GitHub, accessed November 6, 2025, https://github.com/panaversity/learn-agentic-ai
24. A Survey of Agent Interoperability Protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP) - arXiv, accessed November 6, 2025, https://arxiv.org/html/2505.02279v1