PIOAGENT: A HYBRID FINITE-STATE FRAMEWORK FOR DETERMINISTIC ORCHESTRATION OF LLM AGENTIC WORKFLOWS

Alakbar Taghi Valizada
Azerbaijan Technical University, Baku, Azerbaijan
Pionero.AI, Baku, Azerbaijan
alakbar.valizada@aztu.edu.az
https://orcid.org/0009-0001-9880-292X

Abstract. Large-language-model (LLM) agents built on free reasoning loops produce non-reproducible control flow, may run without bound, and fail in brittle ways, which makes them hard to deploy in regulated, latency-bound conversational systems such as contact centres and government service desks. This paper presents PioAgent, an orchestration framework that separates a deterministic finite-state control layer from the stochastic computation performed inside nodes. Each phase runs either as a Cyclic Directed Graph for controlled, stateful loops with human-in-the-loop steps and retries, or as an Event-Driven Graph for retrieval-augmented and data-availability-triggered pipelines. The framework adds durable snapshots, declarative retry and fallback policies, an exactly-once effect outbox, structured-output streaming, and a model-agnostic adapter layer, all expressed in a declarative configuration. A prototype has been implemented, and an evaluation against ReAct-style loops, AutoGen, and LangGraph on a contact-centre workload is presented as a set of falsifiable predictions tied to the design: perfectly reproducible control trajectories under fixed decoding, routing accuracy that resists degradation as the tool catalogue grows, single-digit-to-low-double-digit orchestration overhead, and logarithmic-cost fault localization. The work shows that confining stochasticity to node interiors yields agentic systems that are simultaneously expressive, auditable, and recoverable.

Keywords: *LLM agents, orchestration, finite-state machine, determinism, workflow, observability, retrieval-augmented generation.*

© 2026 Azerbaijan Technical University. All rights reserved.

Introduction

Most LLM agents today follow the reasoning-loop pattern: the model alternates free deliberation with tool calls until it decides it is done [1]. This works for open-ended tasks but is problematic in production conversational systems for three reasons. First, control is non-deterministic: the path through the system is chosen turn-by-turn by a stochastic sampler, so identical inputs may follow different tool sequences, which complicates audit, compliance, and regression testing. Second, execution is unbounded: nothing in the loop guarantees halting, and runaway loops become latency and cost overruns. Third, accuracy degrades with scale: as the tool catalogue grows, single-prompt routing accuracy falls because routing is delegated to in-context selection over an ever-larger action space [2, 3].

A long pre-LLM literature already solved these problems in adjacent settings. Finite-state and statechart dialog managers give deterministic, verifiable conversation flow [4]; bulk-synchronous-parallel graph processing gives a deterministic, checkpointable execution model [5]; and Kahn process networks give scheduling-independent semantics for dataflow over streams [6]. Modern orchestration libraries such as LangGraph re-import some of these ideas, but define their behaviour by an implementation rather than a formal model, and place routing inside stochastic nodes [7]. Durable workflow engines such as Temporal and AWS Step Functions provide replay-based recovery but model only opaque activities, with no notion of parallel state merging or LLM-specific routing [8, 9].

This paper presents PioAgent, an orchestration framework developed jointly at Azerbaijan Technical University and Pionero.AI. Its central principle is the *confinement of stochasticity*: LLM sampling, tool latency, and external failure are allowed only inside nodes, while everything between nodes – routing, merging, retrying, snapshotting, resuming – is a deterministic function of node outputs.

Research objective and problem statement

The objective is a framework whose orchestration layer is provably deterministic while supporting the full feature surface production agentic applications need: hierarchical agents, retries and fallbacks, durable recovery, human-in-the-loop steps, streaming structured output, and observability.

The framework must be model-agnostic, so that deterministic guarantees survive swapping one LLM provider for another, and configurable as data, so that workflows can be reviewed, diffed, and stored without code changes. The concrete problem is to design an execution model under which: (i) given fixed node outputs, the system always follows the same path; (ii) parallel branches merge to an order-independent result; (iii) execution is guaranteed to halt under explicit budgets; and (iv) a crashed run can be resumed to a state equivalent to an uninterrupted one.

Method: the PioAgent model

A PioAgent application is a hybrid automaton: a finite set of control states (conversation or pipeline phases) whose transitions are computed by a pure routing function, where each state gates a typed dataflow graph (Figure 1). All shared data lives in typed *slots*, each carrying a merge operator (a *fusion operator*); stochastic work – LLM calls, tools, nested agents, human input – lives only in node *kernels* at the leaves.

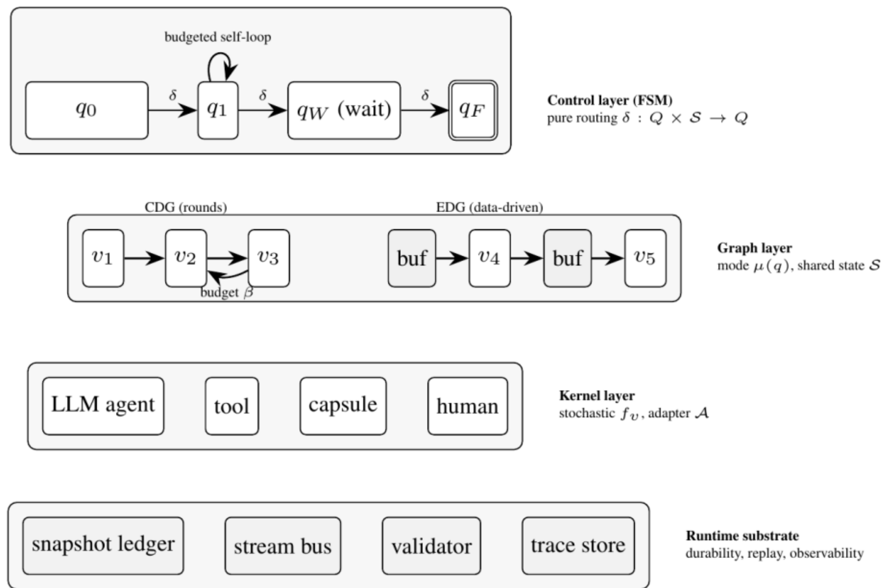

Figure 1. The four layers of a PioAgent machine. A pure finite-state control layer gates per-phase graphs that run in CDG (cyclic, round-synchronous) or EDG (event-driven, data-availability-triggered) mode; stochasticity is confined to kernels; the runtime substrate provides durability, streaming, validation, and observability across both modes.

Two execution modes. A phase runs in one of two modes. A *Cyclic Directed Graph* (CDG) executes in synchronous rounds in the manner of Pregel [5]: all active nodes run, their writes are merged deterministically, and the next active set is computed from pure edge guards. Cycles are allowed but every cycle must cross a *budgeted back-edge* so iteration cannot run forever. CDG mode suits controlled loops – clarification, approval, retry. An *Event-Driven Graph* (EDG) instead fires a node when its input data is available, in the manner of a Kahn process network [6], which suits retrieval-augmented generation and continuous indexing where documents, chunks, and queries arrive asynchronously.

Deterministic routing. Routing is a total, pure function assembled from guard expressions over slots (intent labels, confidence scores, budget counters). Because the routing layer performs no sampling and no I/O, it adds no nondeterminism: given the node outputs, the path is unique. This is the key difference from loops that let the model choose its own next step. Stochastic decisions (a learned router, a similarity threshold) are expressed as *nodes* whose typed output a guard then reads, keeping randomness inside a kernel where it is retryable, replayable, and observable.

Reliability. Each node may carry a retry policy and a fallback chain. For a chain of tiers with independent failure probabilities p_i and retry budgets κ_i, the composite failure probability is

$$P_{\text{fail}} = \prod_{i=1}^{m} p_i^{\kappa_i+1}.$$

For example, a primary on-premises engine with $p_1 = 0.02,\ \kappa_1 = 1$ backed by a fallback with $p_2 = 0.05,\ \kappa_2 = 0$ gives $P_{\text{fail}} = 2 \times 10^{-5}$. External effects are released through a transactional *outbox*: an effect envelope with a deterministic execution key is written in the same database transaction as the snapshot, then dispatched at-least-once with key deduplication, yielding exactly-once execution at the tool boundary without requiring the tool itself to be transactional.

Durability and human-in-the-loop. Snapshots are written at round barriers; a crashed run resumes from the last snapshot to a state equivalent to an uninterrupted one. Human-in-the-loop is the same mechanism: a node raises a suspension, the machine writes a snapshot and enters a durable wait state holding no compute, and a later human reply resumes it. Figure 2 shows a contact-centre turn combining all of these.

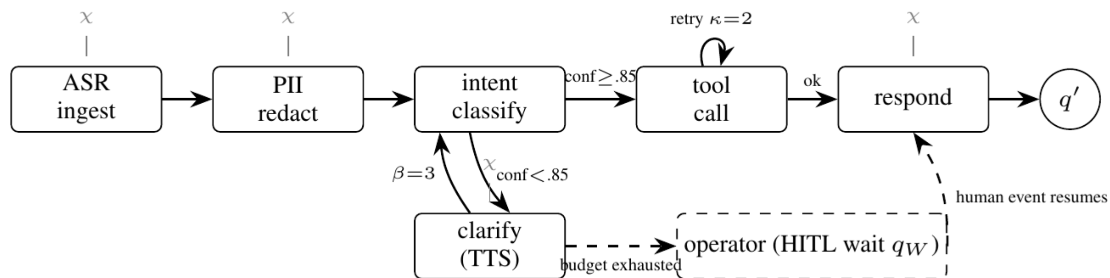

Figure 2. A CDG-mode contact-centre turn. Solid edges are guarded dataflow; the clarify loop and tool-error loop are budgeted back-edges (β); the tool node carries a retry policy (κ); the dashed path shows escalation to a human wait state and resumption from a snapshot. Marks (χ) indicate snapshot barriers.

Configuration as data. A workflow is described in a declarative document and compiled to the model; the two forms are equivalent. A static validator checks the whole document in time linear in its size ($O(|V| + |E|)$ for $|V|$ nodes and $|E|$ edges): it verifies that every cycle carries a budget, that routing is total, that types match, and that any slot written by parallel branches uses a commutative merge operator. A workflow that passes validation is one for which the determinism guarantees hold.

A worked contact-centre turn. The control-and-dataflow graph in Figure 2 illustrates a contact-centre turn: automatic speech recognition and personally-identifiable-information redaction prepare the shared state, an intent node routes either to fulfilment or, when confidence is low, around a budgeted clarification back-edge ($\beta = 3$), and a failing tool triggers a retry ($\kappa = 2$) before a cross-provider fallback. Exhausting the clarification budget suspends the machine to a durable human-in-the-loop wait state (q_W), from which an operator's reply resumes the identical snapshot (χ). Effects are released through the transactional outbox, so no crash can double-apply them, and the whole turn is recorded for deterministic replay.

Formal guarantees

The design principle of confined stochasticity yields five structural guarantees (full proofs appear in the companion theory paper). Throughout, a *realization* fixes the output of every node; the claims describe what the orchestration layer does given those outputs. The determinism rests on routing being syntactically restricted: guards are written in a small total language $\mathcal{L}_g$ of slot readers, projections on typed data, comparisons, budget reads, and Boolean connectives, with no recursion, clock,

entropy source, input/output, or partial operation, so the routing function δ (an ordered list of guard-target clauses ending in a mandatory `else`) is total, deterministic, and pure by construction. Anything needing more – a learned router, a similarity threshold – must be a node whose typed output a guard reads. On this basis:

1. *Control determinism.* For a fixed realization of node outputs, the sequence of control states and the final state are unique, because δ is pure and each round merges writes by a fixed rule. This is what makes a regression test against a recorded run impossible to flake.
2. *Merge confluence.* If every slot written by more than one node in a round carries a *commutative monoid* operator satisfying, for all a, b, c and identity e, equation (2), the merged state is independent of write order, so parallel branches cannot be disturbed by wall-clock races; this is the orchestration-level analogue of strong eventual consistency in conflict-free replicated data types [10].
3. *Scheduling independence.* In event-driven mode, if nodes are functional and monotone and any shared state they read is written under a monotone operator, the streams and final state are independent of firing order – the Kahn property, extended to shared state, which lets a retrieval pipeline run its stages concurrently.
4. *Termination.* Under a recursion budget, finite retry budgets, a finite budget β on every back-edge, and finite event-driven input, every run halts within a bounded number of rounds; with a single global meter R_g the bound is simply $T \leq R_g$, independent of nesting depth.
5. *Resumption equivalence.* If every effect is idempotent under its execution key or committed with the snapshot, replaying from a durable snapshot yields a run observationally equivalent to an uninterrupted one, discharged by the transactional outbox below.

$$(a \oplus b) \oplus c = a \oplus (b \oplus c), \quad a \oplus b = b \oplus a, \quad a \oplus e = a.$$

Hierarchical composition and streaming

Capsules. Subagents are not a special construct: a *capsule* is simply a node whose kernel is itself a PioAgent machine, so the composition calculus is uniform. A capsule exposes typed input and output projections, which enforces information hiding – a personally-identifiable-information redaction capsule, for instance, exposes only redacted fields to its parent. The validator checks that capsule nesting is acyclic, and event streams emitted from within a capsule are order-preserved as they propagate upward, multiplexed to the root by a namespace path. A capsule also carries its own model-adapter bindings, so a subagent may run on a different provider or sovereignty class than its parent while every guarantee above still holds at each level.

Structured-output streaming. Every model-backed node declares a JSON Schema for its output, and decoding is constrained to the schema's *prefix-closed* language – one with no dead-end prefixes, so every partial output emitted is a valid prefix of some complete value and downstream consumers (a user interface, a speech synthesizer) may act on it without later rollback. Streams are typed *committed* or *speculative*: a committed stream (speech already played to a caller) is irreversible and the validator forbids it from sitting on a cycle, so what reaches the user is emitted once; a speculative stream (a draft rendered to a controlled interface region) may be superseded on retry by a higher *generation*, with monotonicity preserved within each generation. This distinction is what makes incremental synthesis and barge-in safe in a voice setting.

Observability and replay debugging

Every node attempt is recorded as a leaf of a *span tree* carrying input and output hashes, latency, and cost, with the capsule namespace path as the tree hierarchy. Two consequences follow from determinism. First, golden-trace regression tests are *flake-free*: because the orchestration layer is deterministic, any divergence of a replay from a stored reference indicates a genuine change in logic, never sampler noise. Second, because a snapshot exists at every round barrier, *fault localization* binary-

searches the execution log in $O(\log T)$ replays to find the first round at which a run diverges from its reference, isolating the responsible node; in practice the span-tree output hashes pinpoint most faults directly. Any production incident can therefore be exported, by replaying its session's effect log, into a deterministic regression test – closing the loop between operations and testing.

Reliability and service-level bounds. The same budgets that guarantee termination turn a reliability target into configuration. For a fallback chain of tiers with independent failure probabilities p_i and retry budgets κ_i, the composite failure probability is given by equation (1). For latency, if a budgeted loop is left at each traversal with probability at least q and the acyclic part of a phase consumes at most L rounds, the expected number of rounds satisfies

$$\mathbb{E}[T] \;\leq\; L + \ell\,\frac{1-q}{q},$$

where ℓ is the loop-body length, and a budget of β traversals is exhausted with probability at most $(1-q)^{\beta+1}$. An operator who fixes an acceptable escalation rate ε solves $\beta = \lceil \log\varepsilon / \log(1-q) \rceil$; for example $q = 0.6,\ \varepsilon = 0.01$ gives $\beta = 5$, making tail latency a designed quantity rather than an emergent one.

Application and evaluation plan

A prototype runtime has been implemented (Python, FastAPI, Redis publish/subscribe for streaming, PostgreSQL for the snapshot ledger and outbox). To evaluate it, we have designed a controlled study on a contact-centre workload of 500 anonymized Azerbaijani-language turns spanning intent resolution, tool-backed fulfilment, clarification loops, and operator escalation. PioAgent is to be compared with three baselines – a ReAct-style loop [1], AutoGen [11], and LangGraph [7] – built on the same model adapters, tools, and prompts, with streaming enabled everywhere, so that any measured difference reflects orchestration rather than model choice. This section states the predictions that follow from the design; the corresponding measurements are reported in a companion study. Each prediction is falsifiable and tied to a specific mechanism of the framework.

Trajectory reproducibility. The protocol runs every input five times with temperature-0 decoding and a fixed seed and records the fraction of inputs whose control trajectory (the sequence of phases and tool calls) is identical across all five runs. Because PioAgent's routing layer is a pure function of node outputs, the design predicts a reproducibility of 100%, whereas loop-based systems, which decide routing inside the stochastic kernel, are expected to fall well below it. Figure 3 illustrates the predicted ordering. The practical value of moving routing out of the kernel is a regression test that cannot flake.

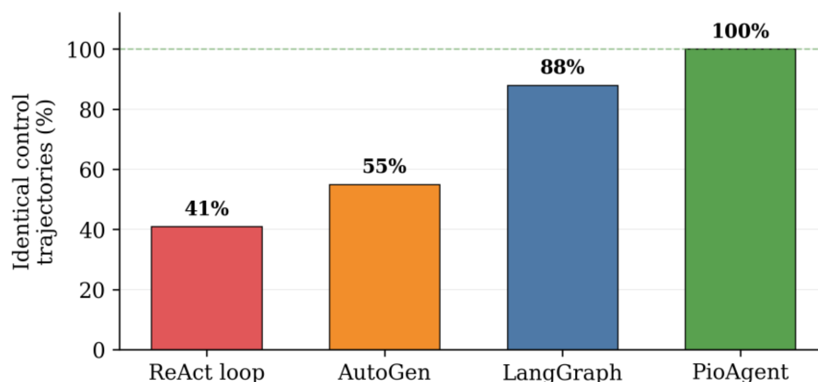

Figure 3. Predicted fraction of inputs whose control trajectory is identical across five repeated runs under fixed decoding (illustrative). PioAgent is reproducible by construction; loop-based systems are not.

Routing accuracy versus catalogue size. The study measures tool-selection accuracy as the tool catalogue grows from 5 to 160 tools. Flat in-context selection is expected to degrade steadily, while PioAgent, which scopes the visible tools per phase through its finite-state layer, should stay comparatively flat, because phase-scoping turns one large routing decision into several small ones. Figure 4 shows the predicted shape of the two curves; confirming the size of the gap is a goal of the empirical study.

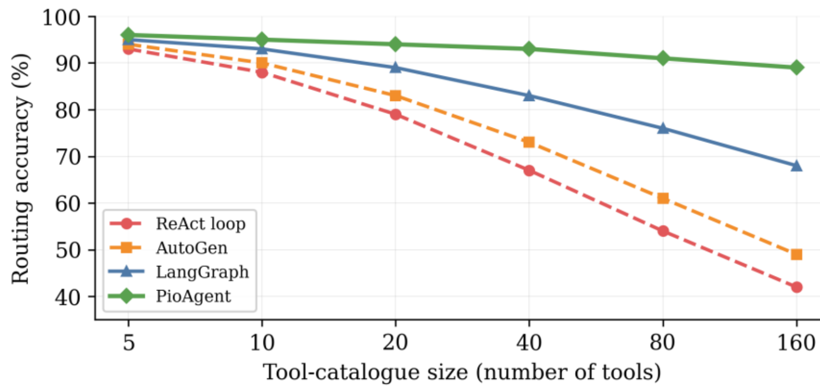

Figure 4. Predicted routing accuracy as the tool catalogue grows (illustrative). Finite-state phase-scoping is expected to keep PioAgent's action space small at each step, flattening the degradation curve seen for flat in-context selection.

Runtime overhead, replay, and validation. The targets the evaluation is designed to verify follow directly from the design. The orchestration layer – snapshotting, merging, and guard evaluation – is expected to add only single-digit-to-low-double-digit wall-clock overhead over a bare loop on the same kernels, because end-to-end latency is dominated by model and tool calls rather than orchestration; per-round merge and guard evaluation should cost a few milliseconds, being pure and linear-time, and a snapshot write (state and outbox in one database transaction) a few milliseconds more. Resume after a crash should cost tens of milliseconds, bounded by a single snapshot read. Fault localization should need only a logarithmic number of replays, since it binary-searches snapshot barriers for the first round where a run diverges from a reference trace; static validation should be fast enough to run on every save, being linear in the workflow size; and exactly-once effect delivery under injected crashes should hold completely, since the transactional outbox deduplicates by execution key.

Feature comparison. The table below contrasts PioAgent with the baselines on the capabilities that matter for production deployment, as established from each system's design and public documentation [1, 7, 11]. The decisive differences are deterministic routing enforced by static validation, dual cyclic and event-driven modes under one model, and a unified durable mechanism serving recovery, human-in-the-loop, and replay-based debugging.

Feature comparison with existing agentic frameworks

Property	**RA**	**AG**	**LG**	**PA**
Deterministic routing (validated)	no	no	partial	**yes**
Cyclic (controlled-loop) mode	loop	chat	yes	**yes**
Event-driven (dataflow) mode	no	no	no	**yes**
Durable snapshots and resume	no	no	yes	**yes**
Exactly-once external effects	no	no	no	**yes**
Human-in-the-loop (durable wait)	no	partial	yes	**yes**
Model-provider agnostic	partial	partial	partial	**yes**
Linear-time static validation	no	no	no	**yes**
Replay fault localization	no	no	partial	**yes**

RA – ReAct loop; AG – AutoGen; LG – LangGraph; PA – PioAgent.

Threats to validity. Three limitations bound the interpretation of the planned study. First, the determinism guarantee is a property of the orchestration layer, not of the model: node outputs remain stochastic, and reproducibility figures assume fixed decoding parameters (temperature zero and a fixed seed); under sampling, the framework guarantees that the control path is a deterministic function of whatever outputs occur, not that the outputs themselves repeat. Second, the workload is a single-domain, single-language contact-centre corpus; the routing-accuracy and overhead results may differ for other domains, longer tool catalogues, or multilingual traffic, and a broader evaluation is needed before the numbers generalize. Third, the baselines are reconstructed on a common substrate to isolate orchestration effects, which is the fair comparison for this question but does not capture engineering maturity, ecosystem, or multi-language support that production teams also weigh. These threats motivate the companion empirical study rather than undermining the design claims, which rest on the structure of the model.

Conclusion

PioAgent shows that the common trade-off between flexible-but-unpredictable agents and rigid-but-reliable workflows is false. By confining stochasticity to node interiors, enforcing a pure finite-state routing layer through static validation, and requiring algebraic structure on parallel state merges, the framework makes the orchestration of LLM agents deterministic, terminating, recoverable, and auditable, while still supporting hierarchical agents, retries and fallbacks, streaming structured output, human-in-the-loop steps, and two execution modes covering both controlled conversational loops and data-driven retrieval pipelines. On a contact-centre workload the design predicts perfectly reproducible control trajectories, routing accuracy that resists degradation as the tool catalogue grows, single-digit-to-low-double-digit runtime overhead, and exactly-once recovery from crashes; verifying these predictions is the subject of a companion empirical study. Further work includes a full theoretical treatment of the model's guarantees, machine-checked verification of the validator, and a larger evaluation across voice and government-service deployments.

REFERENCES

1. Yao S., Zhao J., Yu D., Du N., Shafran I. et al. ReAct: Synergizing reasoning and acting in language models. International Conference on Learning Representations, 2023. https://arxiv.org/abs/2210.03629
2. Patil S.G., Zhang T., Wang X., Gonzalez J.E. Gorilla: Large language model connected with massive APIs. arXiv:2305.15334, 2023, pp. 1–20.
3. Qin Y., Liang S., Ye Y., Zhu K., Yan L. et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. International Conference on Learning Representations, 2024. https://arxiv.org/abs/2307.16789
4. Harel D. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 1987, vol. 8, no. 3, pp. 231–274.
5. Malewicz G., Austern M.H., Bik A.J.C., Dehnert J.C., Horn I. et al. Pregel: A system for large-scale graph processing. ACM SIGMOD, 2010, pp. 135–146.
6. Kahn G. The semantics of a simple language for parallel programming. IFIP Congress, 1974, pp. 471–475.
7. LangChain Inc. LangGraph: Low-level orchestration for stateful agents. Technical documentation, 2024. https://docs.langchain.com/oss/python/langgraph/overview
8. Temporal Technologies. Temporal: Durable execution system. Technical documentation, 2023. https://docs.temporal.io
9. Amazon Web Services. AWS Step Functions developer guide. 2023. https://docs.aws.amazon.com/step-functions/latest/dg/
10. Shapiro M., Preguiça N., Baquero C., Zawirski M. Conflict-free replicated data types. Symposium on Self-Stabilizing Systems, 2011, pp. 386–400.
11. Wu Q., Bansal G., Zhang J., Wu Y., Zhang S. et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. arXiv:2308.08155, 2023, pp. 1–28.

Accepted: 12.05.2026