10.48047/jocaaa.2026.35.01.83

Architectural Patterns for Large Language Model–Driven Autonomous Agents: A Systematic Framework for Scalable and Robust Agentic Systems

Bhaskara Reddy Udaru
Anna University, Chennai, India

Abstract

Large Language Models have become efficient reasoning systems that allow autonomous agents who can plan, make decisions, interact with tools, and adapt to the environment. Nevertheless, most deployments are often monolithic in nature and have tightly bound control logic that restricts scalability, robustness, and reusability. This paper proposes a systematic architectural framework of autonomous agents based on LLM, which suggests a taxonomy of reusable patterns that separate reasoning, planning, memory, execution, and self-evaluation functions. The fundamental patterns presented are Planner-Executor to separate the strategic planning and action implementation, Reasoner-Critic to perform iterative selfassessment and refinement of outputs, Memory-Augmented Agent to maintain persistent knowledge management beyond the scope of the situation, Tool-Oriented Agent to combine external capabilities in an organized manner, and Hierarchical Agent Supervisor to coordinate multiple specialized agents. A reference architecture that comprises these patterns gives specific boundaries to the components, which facilitate modularity, fault tolerance, and scalability. Experimental analysis has shown that pattern-based architectures are much more effective at completing tasks, being able to reason accurately, and recover from failures than monolithic agent designs, especially with complex tasks with long horizons. The proposed structural designs offer fundamental infrastructure for the creation of scalable, supportable, and reliable agentic AI infrastructures within various areas of usage.

Keywords: Autonomous Agents, Large Language Models, Agentic AI Architecture, Multi-Agent Systems, Reasoning And Planning Patterns

1. Introduction

Another change that has occurred in the field of artificial intelligence is the advent of Large Language Models as the building blocks of autonomous agent systems. Such models have already established incredible capacities in reasoning, planning, and performing complex tasks in various domains, which have radically changed the manner in which intelligent systems perceive and deal with their environments. The transition to active autonomous behavior as opposed to passive text generation, has brought new automation opportunities in fields such as software engineering, scientific discovery, and enterprise operations never seen before.

The ReAct paradigm proposed a paradigm in which reasoning and acting are both synergized in language models, allowing agents to produce both verbal reasoning traces and task-specific action in an interlacing way [1]. This enables models to make, sustain, and modify high-level plans dynamically to act and, at the same time, interact with external environments to add more information in their reasoning process. Synergy between reasoning and acting is especially useful when dealing with tasks where both knowledge retrieval and multi-step decisions are necessary, since the reasoning component assists the model in inducing, tracking, and updating action plans, whereas the acting component allows the interface with external sources in order to retrieve information.

10.48047/jocaaa.2026.35.01.83

The extensive reviews of augmented language models have determined that augmentation of reasoning capabilities with tools and outside sources of knowledge is a core level improvement in the work of standard language models [2]. Experiments have shown that adding language models the capacity to reason about when and how to make use of external tools, access pertinent information, and be able to take actions on their surroundings can turn systems into much more capable entities. These enhanced techniques overcome systemic weaknesses of parametric knowledge and allow models to take advantage of specialization in terms of specialized tools via a structured interface. Through the systematic analysis of the strategies of augmentation, it is shown that the proper architecture will lead to those types of integrations to develop a dependable, effective autonomous behavior, or weak systems that may cause cascading failures.

2. Functional Decomposition and Core Architectural Principles

Effective autonomous agent systems construction requires strict functional decomposition into clear layers that can be developed in a modular fashion, tested systematically, and operated in a maintainable way. In this section, a detailed discussion of agent functional elements will be made, and the theoretical basis of pattern-based architecture design, which isolates the concerns and preserves uniform system behaviour, will be formed.

The basic system design of an LLM-based autonomous agent has a number of functional layers that work in a coordinated manner. The perception interface is the main interface between the agent and its environment of operation, which gets as inputs a mixture of natural language instructions up to structured representations of the environmental state. The reasoning and planning core makes use of the language model to produce action sequences, compare alternatives, and update strategies according to environmental feedback. Memory subsystems can offer the short term context retention and long term knowledge persistence, which help agents attempt to be coherent throughout prolonged sequence of interaction that would be impractical with context restrictions.

Chain-of-thought prompting has also been shown to enhance the performance of language models significantly on more complicated reasoning problems [3]. The chain-of-thought methodology allows models to break down multi-step problems into intermediary steps, in effect permitting adequate computation of problems that need more reasoning than can be offered by a direct answer generation method. Experiments in the area of arithmetic reasoning, commonsense reasoning, and symbolic manipulation exercises suggest that encouraging models to present their reasoning processes can be used to find solutions to non-solvable problems. The implications of this discovery are far-reaching to agent architecture, where explicit reasoning decomposition should not be an optional addition to an architectural principle, but instead a fundamental one.

The execution and tool interface layer deals with interactions of the agent with any external system, application programming interface, and computational resources. This layer should be able to deal with natural uncertainties of real-world tool invocation, such as network failures, return values otherwise, timeouts, and resource limitations. The correct separation of execution logic and reasoning components can be used to provide fault-tolerant operation and to provide a systematic process of dealing with errors to ensure that the reasoning state of the agent is not corrupted by cascading failures.

Extensive surveys of self-agent systems implemented using large language models offer detailed records of design guidelines and implementation approaches obtained through successful systems [4]. These studies discuss the construction of agents with separate profiling, memory, planning, and action modules, and conclude that the method of construction has a major impact on the agent's facilities and constraints. The survey results have shown that systems with explicit architectural separation exhibit better performance on complex tasks than monolithic counterparts in which all functionality is contained in the indistinguishable

10.48047/jocaaa.2026.35.01.83

prompt loops. The mechanisms by which agent personas are developed, memory systems which hold experiential knowledge, planning modules which break down goals, and action interfaces which realize the change of environment can all be approached with dedicated architectural consideration, as opposed to implicit treatment.

The coordination and control layer coordinates communication between functional components and controls the flow of execution, allocates resources, and sets termination criteria. The layer provides policies on agent behavior, such as safety limits, priority control, and recovery mechanisms in case of situations that the agent cannot deal with. Coordination takes into account timing, synchronization, and consistency of state in components with varying latencies and reliability properties to attain successful coordination.

Functional Layer	Primary Role	Key Components
Perception Interface	Environment boundary processing	Input parsing, state encoding
Reasoning & Planning	Strategy generation	LLM core, goal decomposition
Memory Subsystem	Knowledge persistence	Short-term context, long-term storage
Execution & Tools	Action implementation	API calls, tool invocation
Coordination & Control	Flow orchestration	Policy enforcement, synchronization

Table 1: Functional Layers of LLM-Driven Autonomous Agents [3, 4]

3. Planner-Executor and Reasoner-Critic Architectural Patterns

The separation of planning and execution is a fundamental architectural design that overcomes the dire constraints of monolithic agent designs. The Planner-Executor pattern is the method that creates a clear separation between high-level strategic thought and low-level action execution and allows optimizing every part of the system independently, while preserving the overall system behavior with a well-defined set of interfaces.

The Planner-Executor architecture has a planning part that takes specifications of tasks and produces a structured action sequence without worrying about the implementation of the execution. This aspect uses reasoning functions of the language model to break down complex objectives into solvable subobjectsives, determine the relationship between actions, create constraints on order, and predict the possibility of failure. The executor will be provided with action specifications by the planner and will deal with their execution, with the tool invocation, monitoring results, and result reporting back to the planning layer to allow a possible modification of the plan.

Experiments on hallucination in large language models offer fundamental findings on failure modes that an architecturally Planner-Executor model must deal with [5]. In-depth surveys define taxonomies of the types of hallucination, distinguishing between factuality hallucination, where the content that is generated conflicts with known facts, and faithfulness hallucination, where what is generated does not conform to the given context or instructions. The knowledge of these failure modes is involved in the architectural decision-making concerning the verification checkpoints, the estimation of confidence, and the recovery plan. The distinct separation between planning and execution sets up natural barriers beyond which the mechanisms of detecting hallucinations can be used to stop erroneous plans before they can be transferred to irreversible behaviours.

The advantages of the Planner-Executor pattern go beyond the increased task completion to increased debuggability and recovery of systematic failures. In the case of failures of execution, the system is able to

10.48047/jocaaa.2026.35.01.83

determine specifically which action has failed and whether or not it is required to do full replanning, or if local recovery can be carried out within the present plan structure. This feature is especially useful in long-horizon workloads in which a full restart would be costly in terms of computational and time expenditure. The pattern also supports caching and reuse of plans so that successful strategies can be used on other instances of the same task without the plans having to be created afresh.

Algorithm 1: Planner-Executor Pattern Execution Loop

```
INPUT:   Task T, ToolSet Tools, MaxRetries R
OUTPUT:  Result, ExecutionStatus

Complexity:
O(n × R) where n = |Actions|

PLANNER_EXECUTOR (T, Tools, R):
01  Plan ← LLM_GeneratePlan (T)        // Decompose task into action sequence
02  Actions[] ← ExtractActions (Plan)
03  Results[] ← ∅
04  FOR EACH action ∈ Actions[]:
05     retryCount ← 0
06     WHILE retryCount < R:
07        tool ← SelectTool (action, Tools)
08        outcome ← Executor.Run (tool, action.params)
09        IF outcome.success:
10           Results[]. append (outcome)
11           BREAK
12        ELSE:
13           action ← LLM_Replan (action, outcome.error)      // Local recovery
14           retryCount ← retryCount + 1
15  FinalResult ← AggregateResults (Results[])
16  RETURN FinalResult, ValidateCompletion (Results[])
```

Fig 1: Planner-Executor and Reasoner-Critic Architectural Patterns [5, 6]

Reasoner-Critic pattern Reasoner-Critic pattern is a complement of Planner-Executor, which adds systematic self-evaluation to the agent architecture by means of special criticism mechanisms. This trend is the embodiment of a critical part that determines the results of the reasoning system, before further actions for execution or ultimate delivery, judging of factual correctness, logical consistency, constraint compliance, and correspondence to declared task goals. The

Self-Refine framework shows that large language models can successfully critique and self-optimize their output in an iterative feedback loop without any further training or reinforcement learning being required [6]. Such a method produces an initial output, feeds back on the output with the same model, and subsequently improves the output based on the feedback and repeats this process until quality criteria are met. This self-refinement mechanism has been found to enhance performance in a variety of tasks such as dialogue response generation, code optimization, and mathematical reasoning. The architectural implication is that the ability to have critique can be affected with the same underlying model, but with a proper

10.48047/jocaaa.2026.35.01.83

prompting strategy, whereas more advanced implementations can use a special evaluation system. Application of the Reasoner-Critic pattern must be done with keen consideration of how the major reasoning and criticism parts relate to each other. The pattern encourages successive iterations of critique, with the ability to refine it over time until quality standards are met, or the inability of the process to enter an infinite loop led by iteration. Good critics should strike a balance between being sensitive to true errors and high rates of false positives, which would waste time in agent development as the agents undergo revision before reaching the correct developmental stage.

4. Memory-Augmented and Tool-Oriented Agent Patterns

The Memory-Augmented Agent pattern is used to overcome basic constraints in the context window of language models and knowledge currency by using external memory systems that are maintained across multiple calls to inference. The pattern allows agents to be able to act coherently over their prolonged task sequences, build experience-driven knowledge, and gain access to information that is beyond the ability to store in in-context learning methods.

Multi-store LLM-based agent memory architectures are usually a combination of various storage mechanisms that are optimized for different access patterns and content types. The short-term memory retains current context and mid-course reasoning outcomes in the context of the current task, which is often enacted in a form of organized context management where important information is prioritized in the face of restricted token budgets. The long term memory avails long term storage of knowledge, experiences, and acquired procedures that ought to guide future behavior even beyond the lines of work and even between separate conversation sessions.

Studies of retrieval-augmented generation formed the basis of incorporating external knowledge sources in language model generation [7]. The retrieval-augmented generation method entails a-priori training of a retrieval system and a language model, and these are jointly trained together, whereby the retriever is conditioned to retrieve documents that can enhance the quality of generation of knowledge-intensive tasks. This architecture shows that the retrieved information can be useful in question answering, fact checking, and coming up with knowledgeable answers that cannot be supported solely by its parametric memory. In the case of autonomous agents, these principles spread to gaining not only the factual knowledge but also the procedural memories, previous experiences, and contextual information used in the ongoing work.

Similarity-based retrieval over a large collection of documents and experience logs has become practiced through vector databases that have become the leading implementation technology of long term agent memory. The memory subsystem then creates dense vector representations of experiences, observations, and accumulated knowledge that can be used to find the nearest neighbor approximately in very little time. The findings of retrieval are integrated into the agent context by building up prompts attentively, which presents the background information to the reasoning and decision-making process. Management of memory should be done effectively by paying attention to the quality of encoding semantic contents, index maintenance as memory expands, and relevance ranking, which reveals memories most useful in the current situation.

10.48047/jocaaa.2026.35.01.83

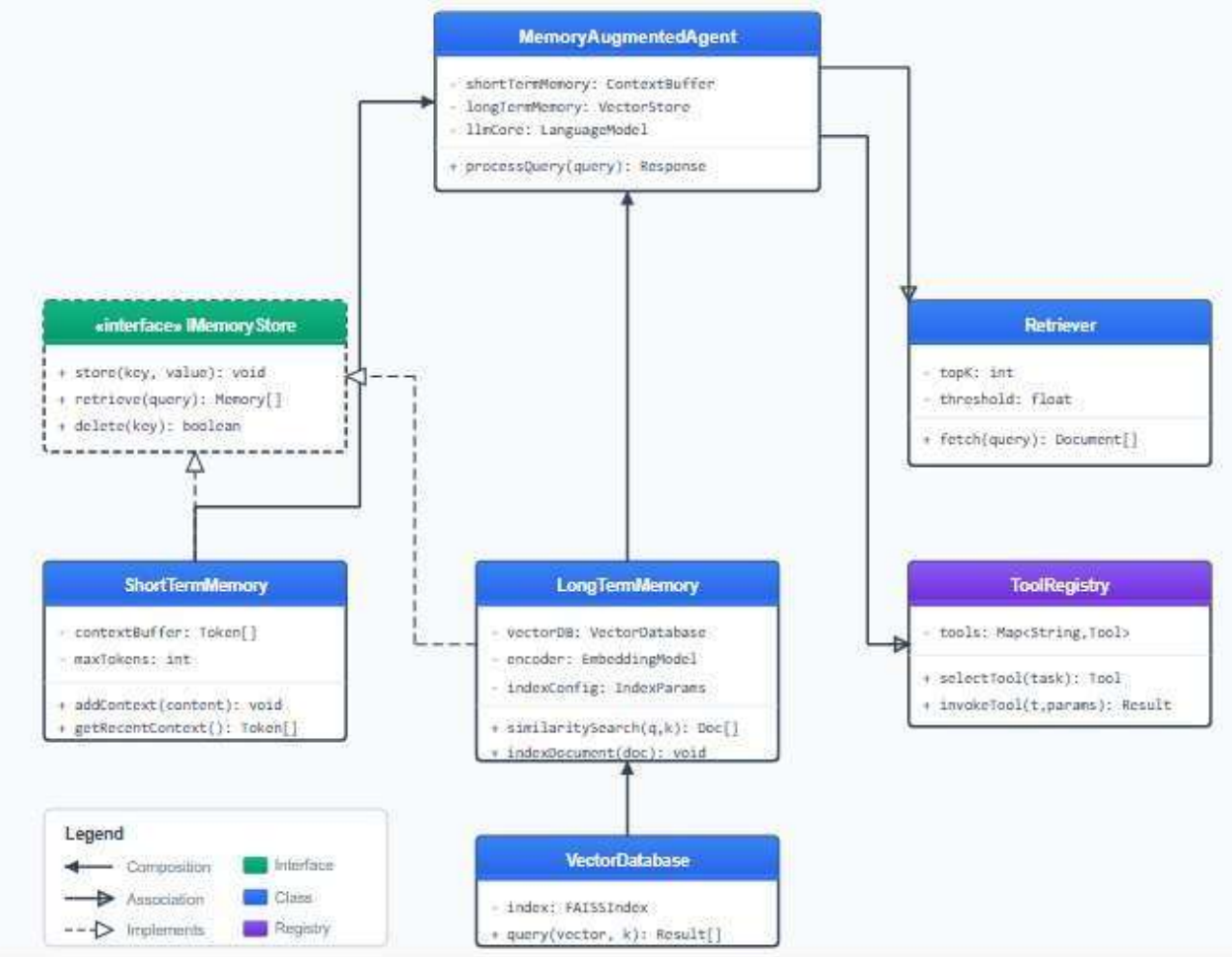

Fig 2: ML Class Diagram - Memory-Augmented Agent [7, 8]

The Tool-Oriented Agent pattern provides the systematic mechanisms that agents can increase their capabilities by invoking external tools, as language models were known to gain a lot of advantages when they have access to specialized tools to perform computation, information search, execute codes and operations in domains. This trend summarizes tool selection, parameter generation, invocation management, as well as result interpretation within a coherent architectural framework, throughout which adherence to safety and extensibility is encouraged.

Trying to do comprehensive research on learning tools by use of foundation models records varied methods by which successful use of tools can be made over language model systems [8]. This work presents paradigms of how models can be able to learn to use the tools using different mechanisms such as learning via demonstrations, learning via feedbacks and learning via tool documentation. The study finds that effective tool integration involves models knowing tool capabilities, identifying instances in which tools would be useful, creating relevant invocations using the right parameters, and interpreting the results of invocations in continuous thought processes. Various tools pose varying integration problems, between simple calculators whose output is predictable and complex APIs that have stateful interactions and may have side effects.

10.48047/jocaaa.2026.35.01.83

Pattern Type	Storage Mechanism	Access Method	Primary Advantage
Short-term Memory	Context buffer	Direct inclusion	Recent state retention
Long-term Memory	Vector database	Similarity search	Persistent knowledge
Tool Selection	Capability registry	Requirement matching	Appropriate tool choice
Tool Invocation	Parameter generation	Structured API calls	Safe execution
Result Interpretation	Output parsing	Context integration	Continued reasoning

Table 2: Memory and Tool Integration Patterns [7, 8]

The Tool-Oriented Agent pattern executes a number of functionalities, which guarantee the safe and efficient use of tools in autonomous systems. Mechanisms of tool selection compare the available tools with the task needs being addressed at a given moment and filter out the tools that would be useful in performing a task without the needless invocation of tools that would result in wastage or threat of failure. The abstract reasoning intentions are translated into invocations of concrete tools with the right values of argument parameters by parameter generation, typically necessitating prudent conversion of types and adherence to format. Result interpretation converts the results of tools into formats that can be used in further reasoning, and gracefully manages successful and error results.

5. Hierarchical Multi-Agent Supervision and Coordination

The Hierarchical Agent Supervisor pattern is also used to confront scalability and complexity management issues that emerge in more advanced agentic systems that have to face problems that are beyond the scope of individual agents. This trend presents architectural hierarchy in which the supervisor agents organize actions of several specialized subordinate agents in such a way that complex tasks can be decomposed into manageable units whilst preserving the overall behavior as an integrated whole through centralized control. Within hierarchical architectures, the supervisor agent gets a high-level task specification and chooses strategies that are suitable to achieve the decomposition, depending on the capabilities of the agents available, the features of the task, and the available resource constraints. The supervisor assigns subtasks to the specialized agents who are the most suitable to handle the specific subtask, monitors their progress in accomplishing the task, coordinates dependencies between multiple parallel tasks, resolves inconsistencies in the output of the agent, and integrates the results into consistent outcomes of tasks. It is a structure that allows the complex problems to be divided into the tasks of the division of labor, and at the same time has accountability because of the supervisory layer.

Survey on large language model-based multi-agent systems Research records coordination mechanisms, communication protocols, and organizational structures that are observed in collaborative agent implementation [9]. The surveys look at the possibility of multiple agents interacting with one another on more difficult tasks in many different interaction patterns, such as cooperative interactions where agents are associated with the same goals, competitive interactions where agents pursue incompatible goals, and integrative interactions involving both of these. The study points to a number of challenges that have to be looked at to achieve effective multi-agent systems - they need to be attentive to agent communication, conflict resolution in case of different recommendations, shared state representation, and timing coordination of interdependent activities. Hierarchical supervision is one of the forms of organization out of various options, such as peer-to-peer coordination and market-based coordination.

10.48047/jocaaa.2026.35.01.83

The supervisor component establishes a number of important coordination functions that provide effective multi-agent behaviour in a complex task environment. Task decomposition breaks down the problems received into a form that can be partitioned using the available specialization of agents without creating too much coordination overhead due to too many interdependencies. Subtasks are assigned to agents in a process referred to as agent selection, whereby the ability profiles are evaluated to provide an understanding of what each agent is good at, rather than the time constraints faced by the agent, which is an indicator of the workload, and a historical performance of similar tasks assigned to the agent offers an empirical guide. Progress monitoring will monitor the level of subtask completion, delays, or failures, which will need intervention and cause corrective measures like assigning tasks or deadline setting where necessary.

Specialized agents in hierarchical systems acquire specialized capabilities that are optimized in specific task types, domains, or modalities of operation. This specialization allows a level of expertise that general-purpose agents would not normally reach, and the hierarchies available access to a wide range of capabilities via the coordination layer, which directs tasks to the appropriate destinations. Specialization of agents can be created by domain-specific fine-tuning of data, limited access to special tools with special capabilities, or specific prompt engineering focusing on specific competencies and personas. Studies of making declarative language model calls into self-improving pipelines offer architectural support to organize complicated workflows consisting of multiple model calls and programmatic code [10]. The DSPy framework presents programming abstractions, which decouple the definition of what a language model system should achieve and the optimization of how it achieves those objectives by prompt engineering and fine-tuning. This method allows formal assembly of language model functionality into pipelines, which can be automatically optimized to accomplish a given task and evaluation criteria. In the case of multi-agent systems, these frameworks offer the basis of defining agent interactions on a declarative basis whilst also allowing the automatic optimization of coordination protocols and agent behaviors.

The coordination layer should focus on such demanding technical needs as resilient communication policies, state synchronization, and graceful recovery of failures. Mechanisms of communication facilitate the exchange of information among the agents and between the agents and supervisors, contributing to a structured exchange of data to support precise coordination and natural language communication to support flexible collaboration. It is state synchronization that maintains a uniform view of shared resources, task progress, and environment conditions when distributed components are operating with different latencies. Failure recovery deals with individual agent failures in a graceful manner, allowing incomplete work to be reassigned and partial results that may still be useful to be preserved.

10.48047/jocaaa.2026.35.01.83

Fig 3: Component Diagram - Integrated Reference Architecture [5, 6, 7, 8, 9]

6. Experimental Framework and Performance Analysis

The testing of pattern-based agent architectures needs systematic testing frameworks that evaluate various aspects of agent performance on varied tasks that test various components of the architecture. This part provides methodological reasons behind agent assessment and speaks about empirical evidence that supports the advantages of architectural modularity over monolithic design strategies.

The autonomous agent is experimentally evaluated in relation to a set of different dimensions of performance that are integrated to provide a description of system effectiveness that can be used in practice. Task completion rate is used to determine how many assigned tasks are completed by the agents within given constraints, which gives the main signs of usefulness in practice in the real world. Accuracy of reasoning is used to test the correctness of the intermediate reasoning processes and the conclusion; therefore, the reliability of agent cognition processes, which support all subsequent behavior. Recovery capability assesses an agent's capability to identify errors, diagnose the causes of errors, and take corrective measures, and is a measure of resilience to unforeseen circumstances that do not follow the expected cases. The token efficiency measures how many computational resources are used compared to the complexity of the task that it accomplished, which gives information on scalability and cost-effectiveness that is important in the deployment of production, where inference costs scale significantly with the length of operation. Pattern-based architectures are more efficient in tokens with narrow-pointed prompting, which provides only the relevant context, less accumulation of irrelevant historical data, and fewer redundant computations when past results are available in memory instead of having to be re-computed.

10.48047/jocaaa.2026.35.01.83

Metric	Monolithic	Planner-Executor	Reasoner-Critic	Memory-Augmented	Tool-Oriented	Hierarchical
Task Completion	Low	High	Medium-High	High	High	Very High
Reasoning Accuracy	Medium	Medium-High	High	Medium-High	Medium	High
Hallucination Rate	High	Medium	Low	Medium	Medium-Low	Low
Recovery Capability	Very Low	High	Medium	Medium-High	Medium	Very High
Token Efficiency	Low	Medium-High	Medium	High	Medium	Medium
Long-horizon Coherence	Very Low	Medium	Medium	Very High	Medium	High

Table 3: Performance Comparison - Pattern-Based vs Monolithic Architectures [8, 9, 10]

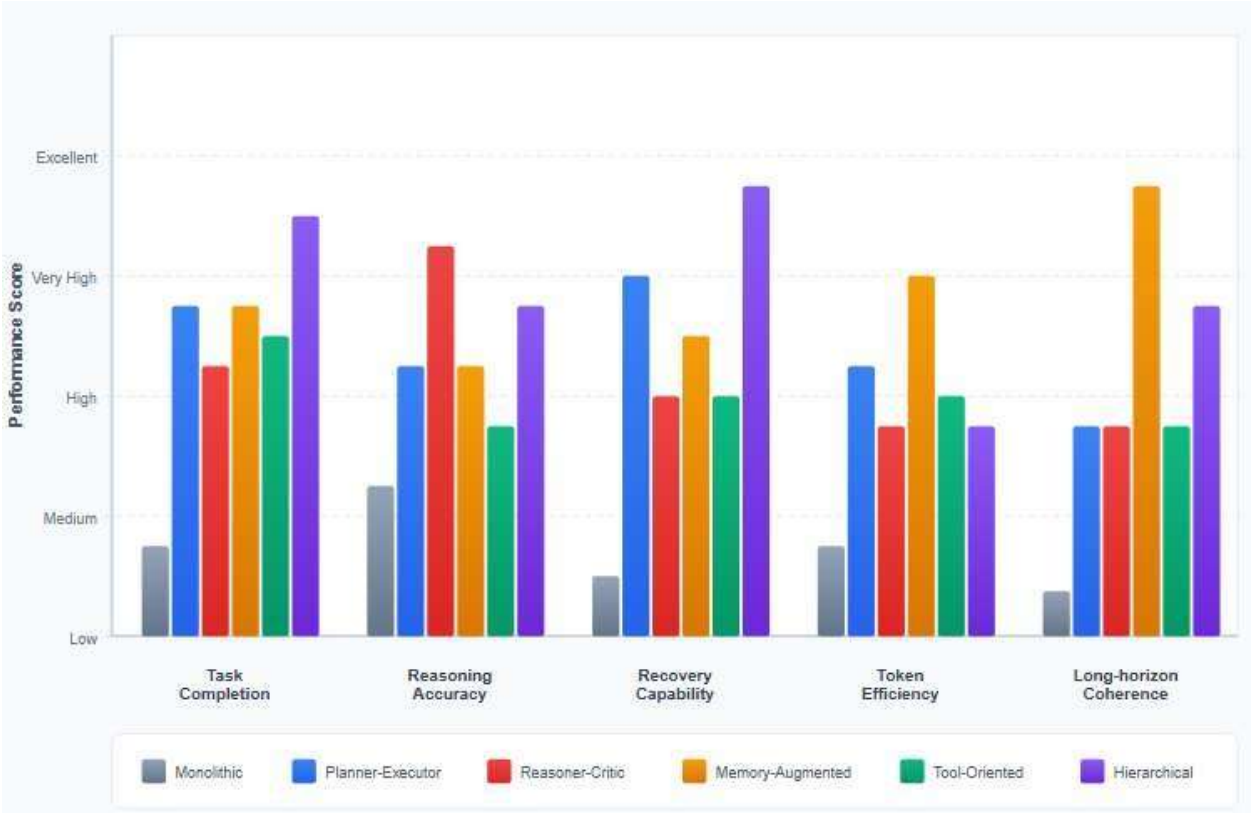

Fig 4: Performance Comparison: Pattern-Based vs Monolithic Architectures [8, 9, 10]

Relative assessment with monolithic baseline systems indicates that pattern-based architectures have regular benefits in evaluation dimensions and task categories. Planner-Executor separation demonstrated by

10.48047/jocaaa.2026.35.01.83

systems has shown better performance of tasks, especially those requiring multi-step completion, whose execution would otherwise be lost by monolithic systems, and also shows good recovery thereafter after intermediate errors invalidate the future reasoning. The clear distinction allows focused correction in case plans are not sufficient, without the need to restart the entire system.

Reasoner-Critic implementations display lower rates of hallucination and an enhanced level of factual accuracy as opposed to systems without self-evaluation systems, as the critique phase identifies errors before they propagate to the final outputs or irrevocable actions. The iterative refinement process is especially useful in tasks that have well-defined quality criteria, and the critic can give actionable feedback to improve it.

Memory-Augmented agents have been shown to possess significant benefits when it comes to the performance of tasks that involve the maintenance of knowledge over a longer interaction, or the operations on information that is beyond context window limits. These systems have consistent behavior over interaction sequences that cannot be sustained in non-augmented alternatives, due to the context overload that they impose.

Tool-Oriented agent patterns provide performance advantages on those tasks that demand computational accuracy, access to up-to-date information beyond that provided by training data, or specialized domain actions that are not provided by language model reasoning alone. Formalized methods of integrating tools minimize errors brought about by poor tool choice and ill-formed invocations to tools, in contrast with the lack of completeness of architectural tool use. Table 4 summarizes pattern effectiveness across task types and failure scenarios.

Task / Failure Type	Best Pattern	Key Advantage
Multi-step Reasoning	Planner-Executor	Structured decomposition
Factual Accuracy	Reasoner-Critic	Self-evaluation loop
Extended Interactions	Memory-Augmented	Persistent context
Computational Tasks	Tool-Oriented	External precision
Complex Projects	Hierarchical	Agent coordination
Execution Errors	Planner-Executor	Local retry vs full restart
Context Overflow	Memory-Augmented	Retrieval vs abandonment

Table 4: Pattern Effectiveness by Task Type and Failure Recovery [7, 8, 9, 10]

Conclusion

A systematic architectural base of LLM-driven autonomous agents has been developed using reusable patterns to deal with the underlying issues in agent design and deployment. The suggested taxonomy includes Planner-Executor, Reasoner-Critic, Memory-Augmented Agent, Tool-Oriented Agent, and Hierarchical Agent Supervisor patterns, each of which addresses particular architectural issues, but adds to the overall system robustness and scalability. The reference architecture that incorporates these patterns encourages modularity by having a clear definition of the component boundaries, fault isolation by separation of concerns, and extensibility by standardized interfaces. Pattern-based architectures show high

10.48047/jocaaa.2026.35.01.83

performance in several aspects compared to monolithic ones in terms of task completion, accuracy of reasoning, and recovery of failures. The maturation of agentic AI as a thorough experiment into viable production systems rather than as an experimental prototype becomes possible by making architectural decisions explicitly and offering reusable solutions to recurrent design issues. Future work involves the formal verification of agent architectures to ensure safety, the introduction of adaptive pattern selection techniques that would optimize configurations in a dynamically changing environment, and stronger connections with reinforcement learning to enhance the acquisition of better policies. Patterns and principles formulated here gain crucial grounds on how to develop agentic AI systems for reliable, scalable, and trustworthy autonomous operation.

References

[1] Shunyu Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," arXiv:2210.03629, 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[2] Grégoire Mialon et al., "Augmented Language Models: a Survey," arXiv:2302.07842, 2023. [Online]. Available: https://arxiv.org/abs/2302.07842

[3] Jason Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv:2201.11903, 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[4] Lei Wang et al., "A Survey on Large Language Model-based Autonomous Agents," arXiv:2308.11432, 2025. [Online]. Available: https://arxiv.org/abs/2308.11432

[5] Lei Huang et al., "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," arXiv:2311.05232, 2024. [Online]. Available: https://arxiv.org/abs/2311.05232

[6] Aman Madaan et al., "Self-Refine: Iterative Refinement with Self-Feedback," arXiv:2303.17651, 2023. [Online]. Available: https://arxiv.org/abs/2303.17651

[7] Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," arXiv:2005.11401, 2021. [Online]. Available: https://arxiv.org/abs/2005.11401

[8] Yujia Qin et al., "Tool Learning with Foundation Models," arXiv:2304.08354, 2024. [Online]. Available: https://arxiv.org/abs/2304.08354

[9] Taicheng Guo et al., “Large Language Model based Multi-Agents: A Survey of Progress and Challenges," arXiv:2402.01680, 2024. [Online]. Available: https://arxiv.org/abs/2402.01680

[10] Omar Khattab et al., "DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines," arXiv:2310.03714, 2023. [Online]. Available: https://arxiv.org/abs/2310.03714