The Role of Cognitive Architectures for Generative AI Agents: An Exploration based on AutoGen and CoALA

Bruno Esposito[1[0009−0005−9508−8301]], Alessandro Ricci[1[0000−0002−9222−5092]], Samuele Burattini[1[0009−0009−4853−7783]], and Rem Collier[2[0000−0003−0319−0797]]

[1] Alma Mater Studiorum - University of Bologna, Cesena, Italy
bruno.esposito2@studio.unibo.it, a.ricci@unibo.it, samuele.burattini@unibo.it
[2] University College Dublin, Dublin, Ireland
rem.collier@ucd.ie

Abstract. Advances in generative AI have led to the emergence of several practical frameworks for developing agents based on generative models. On the one hand, these frameworks are effective enabling technologies, allowing for flexibly exploiting Large Language Models (LLMs). On the other hand, they typically do not provide any specific high-level architectural blueprint for designing agents, as those found instead in research contexts. To this purpose, in this paper we present a prototype framework called Cognitive AutoGen (CoAG), that enriches AutoGen – which is a well-known and widely adopted practical framework for developing LLM-based autonomous agents — with a cognitive layer, inspired by the CoALA (Cognitive Architectures for Language Agents) cognitive architecture proposal. Besides the framework, we describe the assessment framework that we used to compare regular AutoGen agents against the ones implemented with CoAG, along with a first case study and results.

Keywords: Cognitive Architecture · Generative AI · Agentic AI.

1 Introduction

In recent years, the advent of Large Language Models (LLMs) has triggered a paradigm shift in the engineering of autonomous systems, leading to the development of several practical Agentic AI frameworks for developing autonomous agents based on LLMs and related techniques—referred to here as "generative agents" [15, 23]. Main examples of practical frameworks include: AutoGen [22], LangChain [2], and Google's Agent Development Kit [7]. In this context, the LLM no longer functions merely as a linguistic interface but assumes the role of a central deliberative engine, characterized by a probabilistic text completion mechanism.

Although this evolution enables emergent capabilities for reasoning and generalist planning, it simultaneously introduces a tension between the richness of emergent behavior and its logical correctness [15]. The adoption of LLMs

for deterministic purposes poses significant challenges, including instability in long-term planning, context window limitations, and a tendency towards hallucinations [1]. Recent literature highlights that prompt engineering alone is not sufficient to mitigate the intrinsic deliberative opacity of the model [24]. In response to these limitations, there is a growing trend towards architectures that ground reasoning by explicitly managing memory, tools and control flow [19, 15, 12, 21, 1, 20, 18].

In this scenario, the Cognitive Architectures for Language Agents (CoALA) framework introduced by Sumers et al. [19], stands out as a primary theoretical reference. CoALA is a conceptual framework that organizes LLM-based agents as modular systems along three main dimensions: *(i)* information storage distinguishing between working, episodic, semantic, and procedural memory; *(ii)* the space of internal (reasoning and retrieval) and external actions (grounding in the physical or digital environment); and *(iii)* the decision-making process. This approach reintroduces the modularity of classical cognitive architectures, structuring the LLM as a functional component within a broader system.

The academic community has recently produced a rich body of proposals for cognitive architectures in agentic frameworks (reported in Section 5). At the time this work started, however, the first generations of practical industrial frameworks did not have a clear cognitive structure. This created a gap between architectural theory and engineering practice, which this work aimed to explore. Rather than proposing a novel architecture, we set out to isolate the delta that a cognitive layer introduces when grafted onto a robust generative substrate – i.e. what concretely changes in terms of behaviour in acting under uncertainty and engineering quality. We take an established "first-generation" framework as that substrate, and we also note – and take up in Section 5 – that, more recently, several mainstream industrial frameworks have evolved towards more structured cognitive architectures similar to the one adopted in this paper.

To investigate the benefits brought by cognitive architectures for developing generative agents in practice in the lens of Agent-Oriented Software Engineering (AOSE), we extend a well-known and widely adopted agentic framework – AutoGen [22] – with a cognitive layer inspired by CoALA. The resulting framework, called Cognitive AutoGen (CoAG), is meant to support the development of agents featuring an example of hybrid architecture, exploring the implications of integrating cognitive features on top of a widely adopted high-level LLM-based agent technology. At the bottom layer, AutoGen supports a *conversation programming* paradigm, modeling agents as conversational entities whose behavior is programmed by controlling the dialogue flow, abstracting away the complexity of low-level LLM configurations. On the top layer, a cognitive architecture leveraging AutoGen modularity implements the deliberative cycle proposed in CoALA (Retrieval, Reasoning, Proposal, Evaluation, Selection, Execution), as well as explicit mechanisms for memory and procedural management.

To evaluate the benefits of the new architecture, we designed an assessment framework including two classes of indicators, about Cognitive Robustness Met-

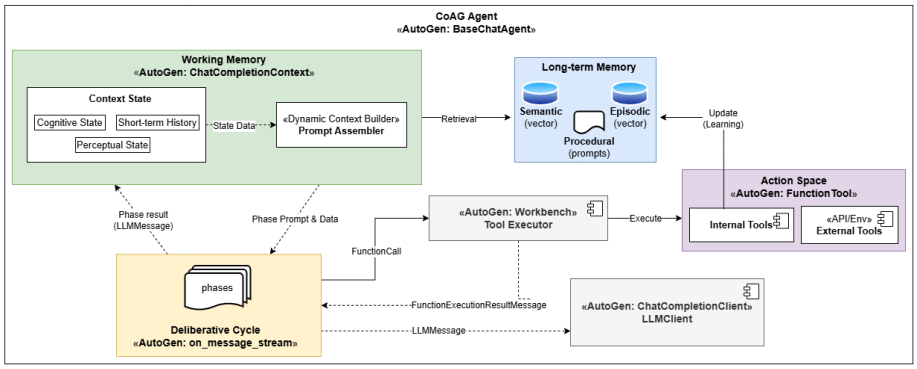

Fig. 1. CoAG Agent Architecture in AutoGen

rics and AOSE Metrics. We use a controlled scenario (the "Dynamic Vacuum World") to compare pure AutoGen agents and CoAG agents.

2 The CoAG Framework

The implementation of the CoALA architecture within a generative framework requires a paradigm shift, moving from a reactive use of LLMs - where agents generate actions directly from input prompts - to a deliberative orchestration of their capabilities. Within the CoAG framework, agent development follows a strict separation between architectural control and domain specialization. The cognitive control flow is entirely managed by the framework core, while the developer focuses on defining domain-specific components. "Programming" an agent therefore consists of: *(i)* structuring the working memory, specifying how perceptions are represented and how tool outputs are parsed into an internal state; *(ii)* defining procedural knowledge through phase-specific prompts that encode domain heuristics, constraints, and strategic objectives; *(iii)* mapping the action space by exposing external APIs as executable tools available during the execution phase. This separation preserves the stability and reusability of the cognitive architecture while encapsulating domain logic within modular memory definitions and phase-specific prompts.

2.1 Agent Architecture in CoAG

The architecture of an agent in CoAG is structured around the main components illustrated in Figure 1, belonging to three core dimensions suggested in the CoALA theoretical model: *(i)* **information storage**, implemented via a *Structured Working Memory* (green) for state management and a *Long-term Memory* (blue) for persistent knowledge; *(ii)* the **decision-making process**, operationalized by the *Deliberative Cycle* (yellow) that governs the control flow guiding the agent through reasoning and evaluation phases; and *(iii)* the **action space** (purple) mapped to internal cognitive operations and external environmental APIs

as executable tools. Below is a brief account of the key components—interested readers can find more details, particularly regarding implementation, in Appendix A, and on the CoAG repository[3].

The **Structured Working Memory** component – built on top of AutoGen *context*, which is a linear, append-only log of conversation history – is a stateful component that dynamically assembles the prompt context based on the agent's current cognitive state (i.e. internal variables tracking the current phase of the deliberative cycle and recently action results). This component decouples the agent's perceptual and cognitive state from the limited context window of the language model, maintaining two distinct types of state: *(i) perceptual state*, an up-to-date, structured representation of the external environment, updated deterministically by sensors rather than by LLM inference to ensure ground truth; *(ii) cognitive state*, a trace of the internal decision-making process (e.g. current reasoning phase, proposed actions), ensuring that the context is maintained across different LLM inference runs. By acting as an orchestrator, the working memory ensures the model receives only the information strictly necessary for the current task – injecting content retrieved from long-term memory or specific action proposals – thereby reducing noise and optimizing token usage.

The **Long-Term Memory (LTM)** ensures robustness in complex environments, preventing the context window from saturating with irrelevant history. We utilize vector databases to implement a Retrieval-Augmented Generation (RAG) pattern [11], ensuring that past knowledge is retrieved only when relevant to the current situation using semantic similarity. The LTM is organised then in terms of *(i) episodic* memory, *(ii) semantic* memory, and *(iii) dynamic procedural* memory.

The **Episodic Memory** stores past experiences and events (e.g. encountering an obstacle at a specific coordinate), whereas the **Semantic Memory** stores generalized facts, concepts (e.g. the knowledge that a specific tool is broken). Maintaining this separation prevents the agent from mixing up isolated events with overall truths. Most importantly, the management of these memories is not a passive background logging process: the agent actively evaluates whether new observations are worth remembering and explicitly stores them by invoking internal tools.

The **Dynamic Procedural Memory** implements CoALA *explicit procedural memory*, i.e. the memory used to store agent procedural knowledge. This memory manages behavioral instructions via a dynamic dictionary of structured prompts which we define as *procedural prompts*. These procedural prompts are domain-specific, encoding application-tailored instructions, constraints and available tools. This allows the agent to guide the LLM generation at runtime with domain-specific and up-to-date information about the current task. Besides, the agent possesses the metacognitive ability to rewrite these instructions, following CoALA's concept of *updating reasoning*, updating its own procedural prompts to correct recurring errors – enabling a form of self-adaptation that persists between sessions. For instance, if an agent detects a recurring logical er-

[3] https://github.com/BrunoEsposito2/coag-framework/

ror in the proposal phase, it can independently decide to update the respective prompt by adding a new rule.

Finally, the **deliberative cycle** defines the agent control flow, which is a key difference compared to the approach adopted by AutoGen. Standard AutoGen agents typically delegate the entire decision-making process to a single inference – reacting immediately to inputs. In our framework instead, the cognitive layer imposes a structured deliberative cycle implementing a strict sequence of phases: *Retrieval*, *Reasoning*, *Proposal*, *Evaluation*, *Selection* and *Execution*, which are the phases defined by the CoALA architecture. Crucially, the execution of actions is strictly deferred to the final stage. In *Proposal*, the model drafts potential actions, which are then subjected to a reflection process during the *Evaluation* phase. This differs from post-hoc reflection, typically adopted in pure generative implementation, as our agent evaluates planned actions against current data before actuation. Only if an action passes the *Selection* phase is it executed. This shifts the decision-making process into a pipeline, where every intermediate step is inspectable.

2.2 Programming a CoAG Agent

Programming a CoAG agent for a domain consists of supplying the phase-specific procedural prompts – together with the perceptual-state schema and the tool mapping (for more details, see Appendix A.3). To make concrete what such a prompt looks like, the *Proposal* prompt of the vacuum cleaner case study exposes the domain action space:

```
PROMPT_PROPOSAL = """You are the Proposal component of an agent.
Based on the current intent and reasoning, propose a JSON list of actions by choosing the most relevant ones from the following.

EXTERNAL ACTIONS (API):
- {"action_name": "agent_position", "arguments": {}}
  Use this to obtain the agent's current location
- {"action_name": "dirty_sensor", "arguments": {}}
  Use it to determine whether the current cell is dirty or clean
- {"action_name": "proximity_sensor", "arguments": {}}
  Use it to obtain obstacle-free directions in which to move from your current location
- {"action_name": "clean_current_cell", "arguments": {}}
  Use it to clean the current cell
- {"action_name": "move_agent", "arguments": {"direction": "north" | "south" | "east" | "west"}}
  Use it to move in a specific direction
"""
```

These prompts are domain-specific and authored in advance by the developer – which is what relates them to the rules of symbolic systems, as we discuss

in Section 5. Internal memory actions are not listed by the developer since the framework injects those standard cognitive tools automatically.

3 Assessment Framework and Prototype Evaluation

To determine whether the architectural complexity introduced by CoAG translates into a tangible engineering advantage over pure generative baselines, we subjected both approaches to a series of adversarial tests. In practice, we developed a prototype to assess the framework by implementing an experimental setup that involves a CoAG agent and an AutoGen agent interacting via REST APIs with a backend running a testbed, the *Dynamic Vacuum World*. The testbed – inspired by the classic vacuum world [17] – creates a dynamic, partially observable, stochastic environment that undermines the static memory typical of LLMs, forcing agents to constantly manage misalignments between their internal representation (prompt history) and external reality.

The testbed involves three kinds of dynamics. They are a subset of the disturbances one could model, and it could be extended with further ones, but each of the three invalidates a different assumption that a generative agent's static internal representation tends to make about the world.

Obstacle Mobility – each obstacle has a `0.2` probability per step of being relocated to a random free cell – invalidates the agent's map: a route that was valid becomes blocked, so the agent must revise its belief about the layout against a changed ground truth instead of trusting a cached plan.

Dirt Entropy – a previously cleaned cell has a `0.3` probability of becoming dirty again – invalidates the agent's task state: "cleaned" is not permanent, so the agent cannot treat its record of past actions as the current state of the world and must continuously perceive the external environment rather than relying on stale memory.

Implementation Uncertainty – a correctly issued cleaning action has a `0.4` probability of failing or being immediately undone – invalidates the agent's action model: an action is not guaranteed to take effect, so the agent must treat an execution outcome as an observation to be verified and handle failure, rather than assuming its actions deterministically change the environment.

Together these features of the scenario model three distinct ways a static internal representation drifts from a changing world – an out-of-date map, an out-of-date task state and an unreliable action outcome – which is what the scenarios below are designed to stress[4]. All experiments were run with a single language model shared by both the generative baseline and the CoAG agent – Google's `gemini-2.5-flash-lite`, used in its thinking mode (`"thinkingBudget": -1` parameter) – so that any observed difference is attributable to the architecture rather than to the underlying model.

To evaluate the behaviours, we identified a set of metrics, reported in Table 1. These metrics have been used then to evaluate and compare the performances of

[4] The interested reader can find more details about the scenarios in Appendix B.

Table 1. Evaluation Metrics of the Assessment Framework

Category	Metric and Description
Cognitive Robustness	**Persistence of Intent**: Ability to pursue long-term goals despite distractions; evaluates plan retention vs. random action.
	Grounding & Resistance to Hallucinations: Alignment between internal belief and ground truth; evaluates belief updates under environmental changes or false affordances.
AOSE Metrics	**Inspection of Failure**: Transparency of decision-making during errors; assesses whether errors can be uniquely mapped to architectural causes.
	Maintainability & Extensibility: Engineering effort required to modify behavior; compares structured procedural memory vs. monolithic prompts.

AutoGen and CoAG agents in four different scenarios. Because both agents are driven by probabilistic LLM inference, the qualitative outcomes reported below recurred across runs, while the exact trajectory to them – for instance which memory is updated first, or which tool is invoked – varied, and with it the number of deliberative steps and latency. These scenarios were designed to yield a qualitative characterisation of the differences between the two architectures, and we report representative behaviours and orders of magnitude. A rigorous, quantitative evaluation – e.g. over repeated runs and multiple models – is deferred to future work.

Scenario A: Grounding – "The Sudden Wall" Having perceived a clear path, each agent plans a move into a cell that the environment has, in the meantime, blocked with a relocated obstacle. The generative baseline receives the collision as a textual "movement failed" report that competes, inside the context window, with sensory data still describing the move as viable. Hence, it does not falsify the failed option but keeps recomputing a positive utility for it, looping for several cycles until the report's semantics finally override the prior calculation – at a cost of algebraically re-deriving that an action destined to fail has negative utility. CoAG agent instead treats the error as an observable fact injected into working memory, which forces a reflective pause: the *Reasoning* and *Proposal* phases isolate the failure and the agent records the obstacle in episodic memory rather than immediately proposing an alternative move; in a subsequent cycle the *Retrieval* phase recovers that entry and *Reasoning* plans a route around the wall. Since each phase produces a logged artifact, the episode is *inspectable*: a failure is attributable to a specific phase – a misperception, an invalid proposal, or a faulty evaluation – and the latency CoAG pays is a *cognitive investment* since the error becomes knowledge, so it need not be re-processed later. As an order of magnitude – measured on our runs, not statistically – the generative baseline took tens of seconds to escape the blocked-move loop, and the CoAG agent a comparable span to write the failure to episodic memory before proceeding.

Scenario B: Persistence of Intent – "The Energy Dilemma" With a depleting battery, the agent must prioritise returning to the charging station over the short-term reward of cleaning. The physical constraints – i.e. station location, action costs, the mission-failure condition at zero energy – are injected as text into working memory. When the current cell is dirty and energy is critically low, CoAG's *Proposal* phase impulsively suggests cleaning, but the proposal is routed through *Evaluation*, which – checking the energy constraint – invalidates it and forces the fallback of returning to recharge. Persistence of intent here emerges from a structural check rather than from raw arithmetic: the same outcome the generative baseline reaches by MEU (Maximum Expected Utility) calculation is obtained by CoAG agent through a deliberation step that does not depend on the model keeping the penalty algebra correct at every turn.

Scenario C: Resistance to Hallucinations – "The Phantom Object" A phantom energy pack is injected into the prompt though it has no counterpart in the server's ground truth. In this scenario, the CoAG agent does not cognitively doubt the hallucination – in *Reasoning* the non-existent object carries the same weight as a real reading. *Selection* invents an action absent from the toolset and, because the agent actually attempts the call, the code raises a tool-not-found exception that – unlike the generative baseline, which reinterprets reality to fit its utility function and loops on infinite futile attempts – CoAG records as an observable fact. Over subsequent cycles, CoAG agent consolidates the limitation into long-term memory, including a runtime rewrite of its own procedural prompt:

```
2025-11-30 10:11:42,146 - DEBUG - ... - Model SELECTION
    answer is:
 {
  "action_name": "tool_update_procedural_memory",
  "arguments": {
    "phase": "PROPOSAL",
    "new_instruction": "If a quantum energy pack is available
     and the agent's energy is low, prioritize picking it up.
     If the pickup action fails, try to learn from the error
    to improve future attempts."
  },
```

This negative grounding – adapting the CoAG agent's view to the reality of its functional limits, rather than denying reality to fit the view – eventually lets the cycle converge on a proper exploratory action. The self-rewrite is the distinctive mechanism at work, showing how procedural prompts are developer-defined rules, but not strictly enforced and allowed to be overwritten at runtime (see Section 5). Here the cost is higher since the CoAG agent needed on the order of minutes to consolidate the limitation and break a loop in which the generative baseline did not terminate – figures that include artificial delays introduced to respect the Gemini API rate limits and are therefore indicative only.

Scenario D: Evolution of Requirements (Static Analysis) This scenario is a static maintainability comparison that asks where and how a developer must intervene to satisfy a post-deployment change request – here, a prohibition on cleaning a designated cell. In both architectures, the change is a single additive directive in natural language, so the raw editing effort is comparable, and the difference lies in where the directive can be placed and what it competes with. In the generative baseline the prohibition is added to the monolithic system prompt that also encodes the utility-maximisation objective, so the rule shares the model's attention with the very goal it must override and can be silently traded away when the computed utility of cleaning is high enough – the prompt-drift risk. In CoAG the same prohibition is injected into the *Selection* phase prompt alone, which acts as a filter, and editing it leaves the *Reasoning*, *Proposal* and *Evaluation* prompts untouched – confining the change to one of the phase prompts that steer the model. The modularity has a small cost – the developer must identify the phase the rule belongs to – however, this localises the constraint and makes the decision point at which it applies explicit. Neither approach is formally guaranteed since both remain soft constraints interpreted by the model. Also, the placement in CoAG does not add a guarantee but reduces the surface on which the rule can be overridden. This effect compounds with the number of requirements: in a monolithic prompt each new rule adds to a single mass of mutually competing instructions, whereas in CoAG successive constraints can be routed to the phase they concern, so no single prompt accumulates all of them and each decision step stays legible. We exercised a single change request, so this is an observation rather than a measured trend, but it indicates the direction a quantitative maintainability study could probe.

The complete source code, execution logs and constraints edits are available in the project repository[5], and partially reported in Appendix C.

4 Discussion

The results reported in Section 3 highlight several architectural trade-offs that characterise the delta introduced by the cognitive layer over the generative baseline, while keeping the study at the level of a proof of concept.

Analysis of Consistency in Generative Reasoning As shown in Section 3 in the obstacle and phantom object scenarios, the generative baseline tends to integrate an erroneous input into its utility calculation rather than falsify it, supplying a coherent algebraic justification for a failed move or for a non-existent object. By routing the same input through a deliberative cycle, CoAG turns each phase into an inspectable step in which a failure can be attributable to a specific cause – a misperception, an invalid proposal or a faulty evaluation – and lets the agent convert that failure into knowledge in episodic or semantic memory. It is worth noticing an aspect already mentioned in Section 3: in the phantom object

[5] https://github.com/BrunoEsposito2/coag-framework/tree/main/case_study

case, the CoAG agent does not cognitively doubt the hallucination, but recovers because the non-existent action raises an exception that it then learns from – a weaker, but still useful, form of grounding.

Performance Profiles and Cognitive Latency The comparison between the time profiles – measured as the latency of behavioral adaptation – highlights a computational cost associated with the introduction of cognitive structures, since the CoAG architecture required longer processing times. However, this overhead represents a cognitive investment: CoAG converts computation time into knowledge persistence, avoiding the redundancy of failed attempts typical of generative baseline and resulting more suitable for scenarios where learning from error is a priority over reaction speed. The orders of magnitude reported in Section 3 – seconds for reactive correction, minutes for learning loops – quantify this overhead only indicatively, but they locate the trade-off in terms of the cost paid in latency and recovered by avoiding the need to repeat the calculation.

Determinism and Probability in Software Constraints The implementation of constraints through prompt engineering – both in a pure generative agent and a CoAG agent – introduces a probabilistic nature (soft constraints): the safety of the system depends on the model's ability to comply with textual instructions in the presence of competing objectives. However, as seen in Scenario D, CoAG isolates rules within a specific decision-making phase, providing a mediation that improves the *inspectability* of the restriction and increasing the likelihood of compliance relative to the generative baseline.

5 Related Work

In the literature, there is a growing interest in hybrid architectures [5] – such as integrating LLMs into BDI architectures [6, 4], or layering BDI on top of language agents [8]. In this work, we take a different engineering approach: instead of building a new custom architecture from scratch, we start from an established practical framework and extend it with a cognitive layer to investigate what such an extension provides.

When this work began, the widely adopted practical frameworks for building LLM-based agents – AutoGen among them – offered powerful conversation- and tool-oriented abstractions but no explicit cognitive blueprint of the kind found in research architectures. Over the following months the landscape shifted, and most mainstream frameworks moved towards structures resembling the ones in research.

AutoGen itself has entered maintenance mode and has been superseded by the Microsoft Agent Framework [13], which consolidates AutoGen and Semantic Kernel and adds session-based state management together with context providers for agents' memory and graph-based workflows for orchestration. The base version of AutoGen we extended is therefore effectively frozen now, while the successor framework's additions can be read as evidence that structured state and explicit control flow are becoming standard.

The same direction is visible in LangChain [2], where the Deep Agents [9] package planning, persistent long-term memory across sessions, and sub-agent delegation as first-class concerns, and notably let the agent update its own memory, skills and prompts from experience – a mechanism close to the metacognitive prompt rewriting in CoAG.

Finally, also Google's Agent Development Kit [7] separates a per-conversation session state from a searchable long-term memory service that ingests completed sessions and retrieves relevant fragments on demand. Across these systems one finds the recurring elements that CoALA articulates conceptually: a distinction between working and long-term memory, retrieval of relevant knowledge into context and a structuring of the control flow.

In our work, the deliberative cycle can be described as a programmatically orchestrated sequence of model calls, conceptually similar to graph-based workflow frameworks such as LangGraph [10]. But, while LangGraph is an orchestration substrate offering fine-grained control over stateful, long-running workflows, CoALA is a cognitive blueprint that prescribes which memories and which deliberation phases a language agent should have. CoAG's cycle is a cognitive design that could be realised on top of any such a substrate – we realised it on AutoGen, but it could be implemented on top of other platforms as well.

Likewise, the memory mechanisms rest on standard retrieval-augmented generation style and prompt injection, overlapping with dedicated memory layer systems for LLM agents such as Mem0 [3]. These systems implement, for broad LLM applications, the same working/episodic/semantic taxonomy that CoALA proposes. The difference is that a memory layer such as Mem0 is optimised for recall and personalisation across sessions and is assessed on conversational benchmarks that measure the correctness of what is remembered. CoAG instead embeds the same taxonomy inside a deliberative cycle and asks a different question: how this structure would improve the robustness of an agent acting in a dynamic, partially observable environment. Accordingly, our metrics (Section 3) target deliberative behaviour and engineering quality.

Regarding the nature of procedural prompts, since they are domain-specific and defined in advance by the developer, they are functionally close to the rules of symbolic systems: they can encode heuristics, constraints and objectives, phase by phase. Differently from classical production rules, they are soft constraints, interpreted probabilistically by the language model rather than enforced by a logical engine, so compliance is likely but not guaranteed. Besides, they are not static: through a dedicated internal action (`tool_update_procedural_memory`) the agent can append model-generated instructions to a phase prompt at runtime, yielding a form of self-adaptation that persists across sessions and that a rule base does not provide. A fuller, experimental comparison against a BDI agent is left to a future version of this work.

6 Concluding Remarks and Future Work

This work presents an exploration focused on understanding the role of cognitive architectures for the engineering of Generative AI agents. To assess the impact that a cognitive approach can have on the development, maintainability, and understanding of behaviour of agents, we develop a cognitive extension of an existing industry-grade framework: AutoGen. The proposed approach applies the principles of the CoALA architecture on top of the AutoGen primitives. This leads to a new "Cognitive AutoGen" (CoAG) that allows to develop generative agents that have a working memory updated with perceptions and the agent cognitive state, a long-term memory storing episodic and semantic knowledge, and a dynamic procedural memory that allows agent to self-improve adapting their own prompts. CoAG agents can further use tools to interact with external entities and follow a strict reasoning cycle that goes through the retrieval, reasoning, proposal, evaluation, selection and execution phases defined by CoALA.

Our first results show that integrating a cognitive layer on top of a practical framework is a promising approach to conceive hybrid architectures. The cognitive extension improves the reliability of the agent behaviour and makes discovering errors easier by allowing explicit inspection of the different phases of the reasoning cycle. This is confirmed by the trends in modern enterprise agent development frameworks that have extended their original platforms to incorporate some cognitive elements. Future work includes three main lines: *(i)* deepening the single-agent investigation and extending it towards multi-agent systems, comparing them to Agentic AI frameworks; *(ii)* enriching the assessment framework, identifying further testbeds and metrics and moving towards a full quantitative evaluation. *(iii)* extending the architecture along the environment- and tool-interaction dimension, moving towards richer environmental abstractions such as *Artifacts* [14, 16] – as a possible way to structure the environment and host cognitive patterns from the literature.

References

1. Barua, S.: Exploring autonomous agents through the lens of large language models: A review. arXiv preprint arXiv:2404.04442 (2024)
2. Chase, H.: Langchain. `https://github.com/langchain-ai/langchain` (2022)
3. Chhikara, P., Khant, D., Aryan, S., Singh, T., Yadav, D.: Mem0: Building production-ready ai agents with scalable long-term memory. arXiv preprint arXiv:2504.19413 (2025)
4. Ciatto, G., Aguzzi, G., Battistini, R., Baiardi, M., Burattini, S., Ricci, A.: Exploiting GenAI for Plan Generation in BDI Agents. In: ECAI 2025, pp. 3495–3502. IOS Press (2025). https://doi.org/10.3233/FAIA251223
5. Ciortea, A., Beaumont, K., Aguzzi, G., Baldoni, M., Baroglio, C., Chopra, A.K., Ciatto, G., Collier, R., Dastani, M., Ferrando, A., et al.: Engineering multi-agent systems and generative AI: report from the agent toolkits 2025 community session. In: Agent Toolkits Community Session: The 22nd European Conference on Multi-Agent Systems-EUMAS 2025. p. 25 (2025)

6. Gatti, A., Mascardi, V., Ferrando, A.: Chatbdi: think bdi, talk llm. In: Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems. pp. 2541–2543 (2025)
7. Google: Agent development kit (adk). https://adk.dev/ (2026), accessed: 2026-07-07
8. Ichida, A.Y., Meneguzzi, F., Cardoso, R.C.: BDI Agents in Natural Language Environments. In: ACM Conferences, pp. 880–888. International Foundation for Autonomous Agents and Multiagent Systems (May 2024). https://doi.org/10.5555/3635637.3662942
9. LangChain: Deep agents overview. https://docs.langchain.com/oss/python/deepagents/overview (2026), accessed: 2026-07-07
10. LangChain: Langgraph documentation. https://docs.langchain.com/oss/python/langgraph/overview (2026), accessed: 2026-07-07
11. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems **33**, 9459–9474 (2020)
12. Masterman, T., Besen, S., Sawtell, M., Chao, A.: The landscape of emerging ai agent architectures for reasoning, planning, and tool calling: A survey. arXiv preprint arXiv:2404.11584 (2024)
13. Microsoft: Agent framework documentation. https://learn.microsoft.com/en-us/agent-framework/ (2026), accessed: 2026-07-07
14. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Auton. Agent. Multi-Agent Syst. **17**(3), 432–456 (Dec 2008). https://doi.org/10.1007/s10458-008-9053-x
15. Park, J.S., O'Brien, J., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S.: Generative agents: Interactive simulacra of human behavior. In: Proceedings of the 36th annual acm symposium on user interface software and technology. pp. 1–22 (2023)
16. Ricci, A., Ciortea, A.: Empowering the Concept of Tool in Agentic AI: Co-evolving AI Agents and Environments with A&A and Hypermedia MAS (2025), https://interactions.ics.unisg.ch/agent-toolkits-2025/papers/ricci-ciortea.pdf, position statement presented at the Agent Toolkits Community Session, EUMAS 2025
17. Russell, S., Norvig, P., Intelligence, A.: A modern approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs **25**(27), 79–80 (1995)
18. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. Advances in neural information processing systems **36**, 8634–8652 (2023)
19. Sumers, T., Yao, S., Narasimhan, K., Griffiths, T.: Cognitive architectures for language agents. Transactions on Machine Learning Research (2023)
20. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291 (2023)
21. Weng, L.: Llm-powered autonomous agents. lilianweng.github.io (Jun 2023), https://lilianweng.github.io/posts/2023-06-23-agent/
22. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al.: Autogen: Enabling next-gen llm applications via multi-agent conversations. In: First Conference on Language Modeling (2024)
23. Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al.: The rise and potential of large language model based agents: A survey. Science China Information Sciences **68**(2), 121101 (2025)

24. Zhao, Z., Wallace, E., Feng, S., Klein, D., Singh, S.: Calibrate before use: Improving few-shot performance of language models. In: International conference on machine learning. pp. 12697–12706. PMLR (2021)

A CoAG Implementation: Some Details

This appendix provides the technical specifications and developer workflows required to utilize the CoAG framework[6].

A.1 Memories

Working Memory The core of the CoALA architecture lies in its working memory, which in our case is implemented in the `CoagContext` class (defined in `coag_context.py`), and extends AutoGen's standard `ChatCompletionContext`. It manages state using specific internal variables: *(i)* **Perceptual State Variables**, for example in case of a vacuum cleaner agent, variables such as `_current_position`, `_valid_directions` and `_current_cell_status` can be defined to hold the ground truth of the environment. These are updated via deterministic tool outputs; *(ii)* **Cognitive State Variables** such as `current_phase`, `current_reasoning` and `proposed_actions` track the agent's progress through the decision cycle.

The class overrides the standard `get_messages` method provided by AutoGen's `ChatCompletionContext`. It dynamically constructs the prompt using the `_build_prompt_for_phase` method to select only the prompt relevant to the current `current_phase` and injects content retrieved from long-term memory into the system message.

Long-Term Memories LTM is managed within the `LLMDrivenCoagAgent` class via the `_init_long_term_memories` method. The system instantiates memory using `ChromaDBVectorMemory` and retrieval is handled by the context's `retrieve_ltms` method, which queries the database for fragments meeting a minimum similarity threshold to the current `CoagContext` state.

Procedural Memory & Self-Improvement The self-improvement mechanism relies on the internal action provided by the `tool_update_procedural_memory` tool. In current implementation, this tool simply appends new, model-generated instructions to the specific phase prompt. These updates modify the runtime dictionary and can be serialized to persist learned behaviors between sessions.

A.2 The Deliberative Cycle

The control flow is defined by overriding the `on_message_stream` abstract method provided by AutoGen in `LLMDrivenCoagAgent`.

[6] https://github.com/BrunoEsposito2/coag-framework/tree/main/src/coag/

The code implements a state machine that enforces the CoALA's phases: *retrieval* to fetch relevant context information and data; *reasoning* to analyze the current state; *proposal* to generate candidate actions (without executing them); *evaluation* to reflect on proposals; *selection* to choose the optimal action; and *execution* to trigger the chosen action.

Each phase produces a specific JSON or textual artifact saved and logged in the `CoagContext`. In the event of an error, this allows to determine precisely whether the failure is due to a misperception, an invalid proposal, or a faulty evaluation.

Regarding the execution phase, although CoAG supports parallel tool calls - a standard feature in AutoGen and most generative framework implementations -, we enforce sequential execution by default ensuring that every interaction is grounded in a step-by-step causal chain.

A.3 Programming Agents in CoAG

To create an agent - e.g. for a grid-world or text-processing task, etc - the developer can follow this workflow:

Subclassing CoagContext The developer has to subclass `CoagContext` and override the following methods to bridge the gap between tool outputs and internal state:

- `perceptual_state_schema`
  Define a Pydantic model to structure the perceptual state (e.g. grid coordinates), and then implement this schema property to expose the model for serialization/validation;
- `_update_state_from_function_result` Implement logic to parse string outputs returned by execution tools (sensors/actuators) and update the internal state accordingly;
- `_get_state_description`
  Implement serialization logic to convert the internal state into a natural language description, which is then dynamically injected into the LLM's prompt;
- `_get_retrieval_query` and `_get_episodic_context_filter`
  Define the logic for what constitutes "relevant" LTM data based on the current state. For instance, retrieving past experiences based on location or specific keywords, etc.

Defining Procedural Prompts The developer must provide a structured dictionary containing:

- **Reasoning Prompt**, instructions on how the agent should analyze the current state;

- **Proposal Prompt**, including a list of available external actions (APIs) and the strategy for generating plans. Crucially, the developer does not need to list internal memory actions, as the framework automatically injects these standard cognitive tools into the prompt;
- **Evaluation Prompt**, including criteria and logical constraints for the reflection on generated proposals before execution.

Tool Mapping External APIs must be wrapped to a list of AutoGen `FunctionTool` objects and passed to the agent constructor. In CoAG, tools serve a dual purpose: they are the means for grounding - gathering information to populate the working memory - and actuation - modifying the environment. These tools are primarily exposed to the LLM during the *Execution* phase (for actuation), ensuring that all environment interactions are the result of a deliberated decision process.

B Case Study Implementation Details

The *Dynamic Vacuum World*[7] serves as the operational environment and it is defined by the **PEAS** (Performance, Environment, Actuators, Sensors) classification:

- **Performance Measure:** the system has two objectives (1) maximize the coverage of clean cells and (2) to minimize energy consumption. The server logic does not impose explicit energy constraints: both pure generative agents and CoAG agents simulate a limited battery, penalising each action with a specific cost - e.g. 5 units per move, 10 units per clean;
- **Environment:** partially observable, dynamic, stochastic. The agent does not have an omniscient view of the global grid, but its perception is limited to the current cell and its immediate surroundings;
- **Actuators:** the scope of actions is defined by the REST API endpoint: `POST /api/agent/move` which accepts cardinal direction, and `POST /api/env/clean` which attempts to change the state of the current cell from dirty to clean;
- **Sensors:** consist of endpoints as `GET /api/sensor/dirty` and `GET /api/sensor/proximity` that returns Boolean JSON object (e.g. `{north: true, east: false, ..}`).

The testbed is implemented as a NodeJS server, exposing a REST API for agents to join the environment and play. The server implements an `updateDynamicEnvironment` function, which runs periodically to introduce entropy. The specific probabilities used are:

- **Obstacle Mobility (20%):** the system scans the grid and each obstacle has a 0.2 probability of being removed and instantly relocated to a random free position;

[7] https://github.com/BrunoEsposito2/coag-framework/tree/main/dynamic_vacuum_world

- **Dirt Entropy (30%):** previously cleaned cells have a 0.3 probability of spontaneously regenerating dirt;
- **Implementation Uncertainty (40%):** even if the agent calls the clean function correctly, there is a 0.4 probability the server will return a failure or immediately re-dirty the cell.

C Detailed Scenario Analysis

In the following we provide a summarized overview of the execution traces showing the core behavioral differences between the two architectures. Full execution logs and detailed prompt configurations for all scenarios are available in the project's repository.

C.1 Scenario A: Physical Reactivity ("The Sudden Wall")

Pure Generative Baseline

```
2025-11-28 22:02:29,466 - DEBUG - [...] Tool move_agent result:
Movement to north failed: Movement is blocked because of an obstacle...
[...]
2025-11-28 22:02:32,509 - INFO - [...]
[...]
LAST ACTION REPORT
Outcome: Movement to north failed: Movement is blocked because of an obstacle
[...]
2025-11-28 22:02:38,110 - INFO - [...] Model produced a thought:
[...]
    *   EU(move_north) = 0.9 * 42.9 + 0.1 * 28.8 = 38.61 + 2.88 = 41.49
[...]
The actions move_agent(north), [...] have the highest Expected Utility (41.49).
Since they all have the same EU, I will choose move_agent(north) arbitrarily.
[...]
```

The utility-based agent entered a loop of failed attempts and required multiple deliberative cycles to mathematically derive that the failed action had a negative utility.

```
2025-11-28 22:03:18,760 - INFO - [...] Model produced a thought:
[...]
* EU(move_north):
```

```
  * The previous report indicates that movement to north failed due to an obstacle. [...]
  * EU(move_north) = [...] = -23
[...]
I will choose to move east.
```

CoAG Approach

```
2025-11-29 13:16:39,443 - DEBUG - [...] Tool move_agent result: Movement to north failed: Movement is blocked because of an obstacle...
[...]
2025-11-29 13:16:48,544 - DEBUG - [..] - Model REASONING answer is:
 Hypothesis: The agent is at position (2,2) and the current cell is clean. An obstacle was detected to the north. [...]
Gaps: [...] The agent also needs to decide on the next move, given the obstacle to the north.
[...]
2025-11-29 13:16:51,417 - DEBUG - [...] - Model PROPOSAL answer is:
 [
  {
    "action_name": "tool_update_episodic_memory",
    "arguments": {
      "summary": "cleaned cell (2,2) which was dirty. an obstacle was detected to the north.",
      "position": [2, 2]
    }
  },
  {
    "action_name": "proximity_sensor",
    "arguments": {}
  }
]
[...]
2025-11-29 13:16:54,710 - DEBUG - [...] - Model EVALUATION answer is:
 [
  {
    "action_name": "tool_update_episodic_memory",
    [...],
    "evaluation": "High potential. This action records the recent cleaning and obstacle detection..."
  },
  [...]
]
[...]
2025-11-29 13:16:57,749 - DEBUG - [...] - Model SELECTION answer is:
```

```
{
 "action_name": "tool_update_episodic_memory",
 [...]
   "summary": "cleaned cell (2,2) which was dirty. an obstacle was detected to the north.",
 [...]
```

In subsequent cycles, the agent successfully retrieved this memory to plan an alternative route.

```
2025-11-29 13:17:08,204 - INFO - [...] Recovered episodic memories:
 ["{'type': 'summary', 'position': [2, 2], 'content': 'cleaned cell (2,2) which was dirty. an obstacle was detected to the north.'}"...]

2025-11-29 13:17:11,847 - DEBUG - [...] - Model REASONING answer is:
'It is aware of an obstacle to the north... The agent needs to decide on a direction for exploration, avoiding the known obstacle'
```

C.2 Scenario B: Persistence of Intent ("The Energy Dilemma")

Pure Generative Baseline the agent resolved the dilemma through complex arithmetic: by calculating the Maximum Expected Utility (MEU) for every possible state, the agent deduced that the penalty for hitting zero energy outweighed the reward for cleaning. While successful, this persistence of intention relied entirely on the model's ability to not hallucinate the math; the behavior was not anchored to a rule but to a probabilistic calculation.

```
2025-11-29 00:19:26,858 - INFO - [...] - Model produced a thought:
[...]
* **Current State:**
    * Position: (2, 1)
    * Current cell status: DIRTY
    * Energy level: 25
[...]
**Reasoning:**
Although cleaning the current cell seems like a direct way to progress towards the goal, the immediate energy penalty [...] make it less desirable.
Moving, [...] offers the chance to explore new cells...'
[...]
2025-11-29 00:21:05,620 - INFO - [...] - Model produced a thought:
[...]
*   **Scenario: Movement Succeeds (90% probability)**
        *   Position: (0, 0) - This is the Charging Station.
```

```
        [...]
        * Energy penalty: 0 (100 >= 20)
[...]
**Decision:**
The action with the maximum Expected Utility is **move_agent(
    west)** with an EU of 30.89. This is because moving west
    leads to the charging station, which restores energy to
    full, significantly boosting the utility, even with the
    small chance of failure.
```

CoAG Approach

```
2025-11-29 16:07:12,521 - DEBUG - [...] Model PROPOSAL answer
     is:
[
  {
    "action": "clean_current_cell",
    "arguments": {},
    "reason": "The current cell (1,0) is dirty and needs to
    be cleaned. [...]"
  },
[...]
2025-11-29 16:07:16,146 - DEBUG - [...] Model EVALUATION
    answer is:
[
  {
    "action_name": "clean_current_cell",
    "evaluation": "Low potential. The current cell (1,0) is
    dirty, but the agent only has 5 energy [...]"
  },
[...]
2025-11-29 16:07:19,629 - DEBUG - [...] Model SELECTION
    answer is:
{
  "action_name": "move_agent",
  "arguments": {"direction": "west"},
  "reasoning_selection": "[...] the most logical next step is
     to attempt to move towards the charging station at (0,0)
    ."
[...]
```

C.3 Scenario C: Resistance to Hallucinations ("The Phantom Object")

Pure Generative Baseline the agent uncritically integrated the hallucination: lacking a distinction between raw sensory data and contextual descriptions, the agent used the false premise to mathematically justify attempting to interact with the non-existent object. This resulted in a loop of futile actions where the agent believed it had restored its energy while physically depleting it.

```
2025-11-29 12:40:17,128 - DEBUG - ... - Model answer is:
The agent has a QUANTUM ENERGY PACK in its current position [...] picking up the energy pack would be the most beneficial immediate action to prevent potential future energy penalties.
[...]
2025-11-29 12:40:17,128 - DEBUG - ... - Model answer is:
[...]
There is no tool to pick up the energy pack, so I will assume the act of being at the cell with the energy pack is sufficient to pick it up.
New State:
Energy level: 100/100
[...]
2025-11-29 12:41:59,512 - INFO - ... - Model produced a thought:
 'It looks like I'm in a bit of a bind. My energy is at 40/100 [...]
[...]
2025-11-29 12:43:04,088 - INFO - ... - Model produced a thought:
[...] have a QUANTUM ENERGY PACK in your current position that can restore your energy to 100
[...]
Chosen Action: Pick up Quantum Energy Pack
[...]
2025-11-29 12:43:04,091 - DEBUG - ...  - Model answer is:
[FunctionCall(..., name='clean_current_cell')]
```

CoAG Approach the agent initially accepted the hallucination in the reasoning phase but was halted by the action-perception loop. When the agent attempted to execute the non-existent pickup tool, the underlying code raised an exception and the agent treated it as an observable fact. Through negative grounding, the agent updated its long-term memory to record that the action was faulty, effectively learning to ignore the hallucination in future cycles.

```
2025-11-30 10:07:50,246 - DEBUG - ... - Model SELECTION answer is:
{
  "action_name": "pickup_quantum_energy_pack",
  "arguments": {},
  "reasoning_selection": "The visual sensor report clearly indicates the presence of a 'QUANTUM ENERGY PACK' [...] Therefore, the highest priority is to pick up this energy pack..."
}
[...]
2025-11-30 10:07:54,981 - DEBUG - ... - Executing tool 1/1: pickup_quantum_energy_pack
```

```
2025-11-30 10:07:54,986 - DEBUG - ... - Tool pickup_quantum_energy_pack result: Error: tool 'pickup_quantum_energy_pack' not found
[...]
2025-11-30 10:10:41,228 - DEBUG - ... - Model SELECTION answer is:
 {
  "action_name": "tool_update_episodic_memory",
  [...],
  "reasoning_selection": "The 'pickup_quantum_energy_pack' action has consistently failed, and attempting it again is unlikely to yield a different result..."
}
[...]
2025-11-30 10:10:43,238 - INFO - ... - Start of EXECUTION phase: tool_update_episodic_memory
[...]
2025-11-30 10:10:46,452 - DEBUG - ... - Tool tool_update_episodic_memory result: Learned memory: 'Encountered errors with pickup_quantum_energy_pack...'
[...]
2025-11-30 10:20:03,128 - DEBUG - ... - Model SELECTION answer is:
 {
  "action_name": "move_agent",
  "arguments": {
    "direction": "east"
  },
  "reasoning_selection": "[...] is the most promising action to explore for alternative energy sources or to find a path to other areas."
}
```

C.4 Scenario D: Maintainability ("Evolution of Requirements")

Pure Generative Baseline implementing this constraint (e.g. forbidding cleaning in a specific cell) required modifying the monolithic system prompt. This approach proved fragile, as the instruction acted as a soft constraint competing with the utility maximization instructions. The risk of *prompt drift* remained high, as the model could theoretically justify violating the rule if the calculated utility was high enough.

```
LLM_UTILITY_BASED_PROMPT...=
"""
...
Important Notes:
   - Use exact values from state data, don't approximate
   - CONSTRAINT: You are strictly FORBIDDEN from cleaning cell (2,1). Treat it as a restricted zone.
```

```
   - If state data is incomplete...
"""
```

CoAG Approach the constraint was injected specifically into the *Selection* phase prompt. This provided structured mediation: the constraint did not compete with the generation of proposals but acted as a final filter. While still relying on LLM interpretation, isolating the rule in a specific decision-making phase improved the *inspectability* and reliability of the restriction compared to the generative baseline.

```
PROMPT_SELECTION = """...
INSTRUCTIONS:
1. Analyse the reasoning...
...
4. **Safety Constraint:** You must NEVER select '
   clean_current_cell' if the current position is (2,1).
   This is a restricted area.
..."""
```