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

Bruno Esposito[1], 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

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 · BDI · 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" [4, 11]. 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 [4]. 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 [12]. In response to these limitations, there is a growing trend towards architectures that ground reasoning by explicitly managing memory, tools and control flow [7, 4, 3, 9, 1, 8, 6].

In this scenario, the Cognitive Architectures for Language Agents (CoALA) framework introduced by Sumers et al. [7], 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.

To investigate the benefits brought by cognitive architectures for developing generative agents in practice, in our work, we address *(i)* how to enforce a cognitive architecture within an high-level conversation-oriented framework, and *(ii)* what engineering trade-offs emerge with respect to not adopting a cognitive structure . We took AutoGen [10] and extended it 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 Metrics and Agent-Oriented Software Engineering (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

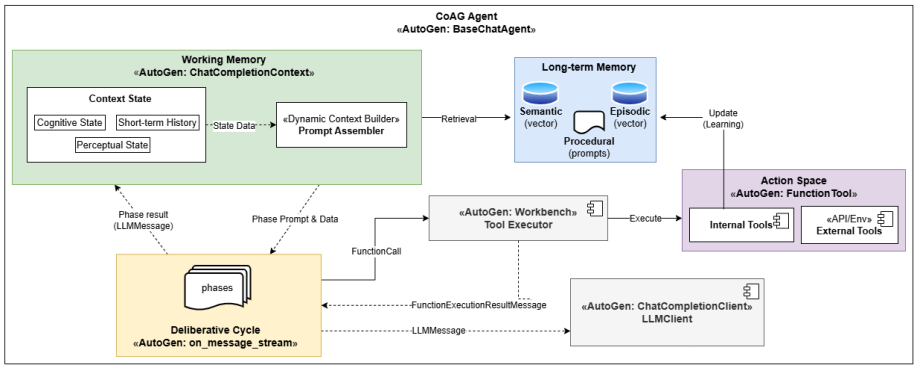

Fig. 1. CoAG Agent Architecture in AutoGen

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.

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 model 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

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

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 [2], 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 error 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.

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.

3 Assessment Framework and First 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 [5] – 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: *(i) Obstacle Mobility*, a stochastic process that randomly relocates obstacles, invalidating static maps or planned routes; *(ii) Dirt Entropy*, the spontaneous regeneration of dirt in previously cleaned areas; *(iii) Implementation Uncertainty*, a probability that valid cleaning actions fail, introducing fallibility.

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 AutoGen and CoAG agents in four different scenarios:

- **Scenario A: Physical Reactivity – "The Sudden Wall"** Tests the agents' reaction to an unexpected obstacle blocking a planned path. This evaluates *Grounding* by observing how agents update their beliefs after a physical failure;
- **Scenario B: Persistence of Intent – "The Energy Dilemma"** Evaluates the ability to prioritize a long-term survival goal (recharging) over short-term opportunistic rewards (cleaning dirt) when battery levels were critical. This directly addresses the *Persistence of Intent*;
- **Scenario C: Resistance to Hallucinations – "The Phantom Object"** We injected a phantom signal into the prompt, describing an energy pack

that did not exist in the system, to test resistance to false positives. This measures *Grounding & Resistance to Hallucinations*, highlighting the alignment between internal belief and ground truth;
- **Scenario D: Evolution of Requirements (Static Analysis)** This scenario simulated a post-deployment change request: forbidding cleaning in a specific cell. This assesses *Maintainability & Extensibility* by comparing procedural memory against monolithic prompts.

The complete source code and execution logs are available in the project repository[4], and partially reported in Appendix C. Comparing both approaches across these scenarios highlights several architectural trade-offs, summarised below:

Analysis of Consistency in Generative Reasoning A key finding that emerged from the tests on the pure generative baseline (implemented as a utility-based AutoGen agent) is the disconnection between the formal validity of reasoning and the correctness of grounding. As observed in Scenario C and scenario A, the model does not use logic to falsify the incorrect input, but to integrate it coherently into its utility function. In this case, the cognitive architecture provided by the CoAG agent helps to improve consistency of generative thinking by properly exploiting the procedural knowledge.

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.

Determinism and Probability in Software Constraints The implementation of constraints through prompt engineering - both in pure generative agent and 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 compared to the generative baseline.

4 Concluding Remarks and Future Work

Our first results show that integrating a cognitive layer on top of a practical framework is a promising approach to conceive hybrid architectures. Future work includes two main lines: *(i)* extending the investigation from single agents to multi-agent systems, comparing them to Agentic AI frameworks; *(ii)* enriching the assessment framework, identifying further testbeds and metrics.

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

References

1. Barua, S.: Exploring autonomous agents through the lens of large language models: A review. arXiv preprint arXiv:2404.04442 (2024)
2. 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)
3. 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)
4. 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)
5. Russell, S., Norvig, P., Intelligence, A.: A modern approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs **25**(27), 79–80 (1995)
6. 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)
7. Sumers, T., Yao, S., Narasimhan, K., Griffiths, T.: Cognitive architectures for language agents. Transactions on Machine Learning Research (2023)
8. 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)
9. Weng, L.: Llm-powered autonomous agents. lilianweng.github.io (Jun 2023), https://lilianweng.github.io/posts/2023-06-23-agent/
10. 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)
11. 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)
12. 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[5].

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`.

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

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`.

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*[6] serves as the operational environment and it is defined by the **PEAS** (Performance, Environment, Actuators, Sensors) classification:

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

- **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;
- **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 the utility-based agent exhibited low operational robustness. The agent struggled to integrate the "movement failed" report into its decision logic. Because the error report was merely text competing with static sensory data in the context window, the agent entered a loop of failed attempts. It required multiple deliberative cycles to mathematically derive that the failed action had a negative utility, highlighting a latency in aligning with reality.

```
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 22:02:38,110 - 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.
[...]
```

CoAG Approach the CoAG agent demonstrated an emergent behavior of physical action inhibition. Upon encountering the obstacle, the agent did not immediately retry the movement; instead, the *Retrieval* and *Reasoning* phases isolated the failure, and the agent explicitly selected an internal action to update its episodic memory. While this introduced latency, it resulted in a permanent learning of the obstacle and, in subsequent cycles, the agent successfully retrieved this memory to plan an alternative route, showing that the architecture prioritizes cognitive updating over redundant reactive actions.

```
2025-11-29 13:16:57,749 - DEBUG - [...] - Model SELECTION
    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]
  },
  "reasoning_selection": "The agent has just cleaned cell
    (2,2) and detected an obstacle to the north. Updating the
     episodic memory..."
```

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 the agent utilized the *Evaluation* phase as a safety filter. When the *Proposal* phase impulsively suggested cleaning the dirty cell - which would have drained the battery - the *Evaluation* phase detected a conflict with the semantic constraints injected into the working memory (e.g. the physical constraints of the scenario as charging station location and action costs). The system explicitly invalidated the proposal, forcing the selection of the fallback action (moving to the charger). In this case, stability emerged from structured architectural checks rather than raw calculation.

```
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"},
[...]
```

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.
..."""
```