SAGEN: Situational Awareness for Generative Agents

A Modular Cognitive Architecture for Persistent State
in Language Model Agent Systems

Jake Lawrence[1]

[1]Independent Researcher

March 2026

Abstract

Large language model (LLM) agents operate under a fundamental constraint: each inference call is stateless. Existing approaches to agent memory, from retrieval-augmented generation to conversation summarization, treat persistence as a storage problem. We reframe it as a *cognitive architecture* problem. We introduce **SAGEN** (**S**ituational **A**wareness for **GE**nerative Age**N**ts), a modular framework that maintains structured, evolving awareness state across agent interactions through six interoperating cognitive modules: Goal Graph, Trajectory, World Model, Self Model, Attention Priorities, and Interaction Protocol. SAGEN operates via an Observe–Update–Inject loop coordinated by a domain-agnostic engine and domain-specific adapters. Unlike flat memory buffers or vector retrieval systems, SAGEN provides the agent with a compressed, prioritized, and semantically structured representation of where it is, what it is doing, and why. We present the architecture, formal specification, a reference implementation, and a proof-of-concept evaluation in the conversational domain demonstrating goal tracking, topic pivot detection, callback recognition, and emotional state monitoring. We release SAGEN as open-source software and discuss its implications for agent autonomy, multi-turn coherence, and domain-portable cognition.

Keywords: cognitive architecture, language model agents, situational awareness, agent state management, goal tracking, blackboard architecture

1 Introduction

The deployment of large language models as interactive agents has exposed a structural gap between model capability and operational coherence. A model that can reason about complex plans within a single context window routinely loses the thread of those same plans across sequential interactions. This is not a failure of intelligence. It is a failure of *architecture*.

The dominant paradigm treats agent memory as information retrieval: store facts, embed them, retrieve relevant chunks at query time. Retrieval-augmented generation (RAG) [Lewis et al., 2020] and its derivatives address the question "What does the agent know?" They do not address the questions that matter for coherent, goal-directed behavior: *What is the agent trying to do? What just changed? What should it pay attention to? What has it tried before?*

These are questions of **situational awareness** – a concept well-studied in human factors research [Endsley, 1995], cognitive science [Newell, 1990], and autonomous systems [Russell and Norvig, 2016] but underexplored in the context of LLM agents.

We introduce SAGEN (Situational Awareness for Generative Agents), a cognitive architecture that provides LLM agents with structured, persistent, evolving awareness state. SAGEN does not replace retrieval-based memory. It operates at a different level of abstraction: where RAG manages *what the agent knows*, SAGEN manages *what the agent understands about its current situation.*

The core design principles are:

1. **Modularity.** Six cognitive modules that can be composed, extended, or replaced independently.
2. **Domain agnosticism.** A plugin-based adapter pattern that separates domain-specific perception from domain-general state management.
3. **Compression over accumulation.** Inspired by ACT-R's memory decay [Anderson et al., 2004], the system prioritizes recent, salient, and failure-associated state over exhaustive history.
4. **Injection-native.** State is rendered as compact, structured text designed for insertion into LLM context windows.

This paper makes the following contributions:

- A formal specification of six-module cognitive architecture for LLM agent situational awareness (§3).
- The Observe–Update–Inject (OUI) loop as a domain-portable state management pattern (§4).
- A domain adapter interface enabling SAGEN to generalize across application contexts (§5).
- A reference implementation and proof-of-concept evaluation in the conversational domain (§6).
- An open-source release of the framework with extensible adapter system.

Companion work. This paper is part of a research program on classification as infrastructure in intelligent systems. A companion paper [Lawrence, 2026b] applies the same structural analysis to inference execution planning, arguing that the taxonomy of plan types determines the optimization space for LLM decoding. A position paper [Lawrence, 2026a] develops the shared theoretical framework, drawing on Bowker and Star's analysis of classification infrastructure [Bowker and Star, 1999]: in any system where an agent must act under uncertainty, the classification layer functions as infrastructure – embedded, transparent when working, constitutive rather than descriptive, and resistant to change.

2 Related Work

2.1 Cognitive Architectures

SAGEN draws on a lineage of cognitive architectures developed for both human cognition modeling and artificial agent design. ACT-R [Anderson et al., 2004] provides the inspiration for SAGEN's memory decay model: declarative memory chunks have activation levels that decay over time but receive boosts from use, and certain high-salience memories (failures, surprises) resist decay. SOAR [Laird, 2012] contributes the concept of a structured working memory that mediates between perception and action, analogous to SAGEN's AwarenessState blackboard. The blackboard architecture itself [Hayes-Roth, 1985] informs SAGEN's design: multiple specialist modules contribute to and read from a shared state representation.

Where classical cognitive architectures assume tightly-coupled perception-action loops, SAGEN is designed for the peculiar constraints of LLM agents: perception and action happen through text, state must be serialized into token-bounded context windows, and the "cognitive cycle" is an API call rather than a continuous process.

2.2 LLM Agent Frameworks

Recent frameworks for LLM agents address persistence through several strategies. LangChain [LangChain, 2023] and LlamaIndex [Liu, 2022] provide memory abstractions ranging from conversation buffers to vector-indexed retrieval stores. AutoGPT [AutoGPT, 2023] maintains a flat list of short-term and long-term memory entries. These approaches are predominantly *flat*: they store facts and messages without structural relationships between them.

Reflexion [Shinn et al., 2023] introduces self-evaluation as a memory mechanism, storing linguistic feedback from failed attempts. LATS [Zhou et al., 2023] uses tree search over action spaces. Voyager [Wang et al., 2023] accumulates a skill library through code generation. Each addresses a specific aspect of agent persistence without providing a unified cognitive model.

Generative Agents [Park et al., 2023] is perhaps the closest precursor: it maintains a memory stream of observations, uses retrieval to surface relevant memories, and generates reflections that abstract over raw experience. SAGEN differs in that it replaces the flat memory stream with structured cognitive modules and replaces retrieval-time reasoning with continuous state maintenance.

2.3 Situational Awareness

Endsley's three-level model of situational awareness [Endsley, 1995] – perception, comprehension, and projection – provides a theoretical grounding. In Endsley's framework, Level 1 is perceiving elements in the environment, Level 2 is understanding their meaning, and Level 3 is projecting future states. SAGEN's World Model captures Level 1, the Goal Graph and Attention Priorities capture Level 2, and the Trajectory module (with its pattern of recording transitions and compressing history) enables Level 3 reasoning by the consuming LLM.

3 Architecture

SAGEN's awareness state is organized as a **blackboard** [Hayes-Roth, 1985]: a shared data structure that multiple cognitive modules read from and write to. The unified state object, `AwarenessState`, contains six modules and two coordination fields (Figure 1).

3.1 Module 1: Goal Graph

The Goal Graph maintains a directed acyclic graph of agent objectives. Each goal has a lifecycle (active, completed, abandoned, blocked, deferred) and a provenance (explicit from user statement, inferred from behavior, emergent from state analysis, or spawned from another goal).

Goals can be hierarchically nested: a root goal "Learn Python" may spawn sub-goals "Understand list comprehensions" and "Build a web scraper." The graph supports dependency tracking: a goal can declare that it `depends_on` other goals, enabling the engine to mark goals as blocked when their dependencies are unresolved.

The graph exposes three key views: `active_goals` (all goals with status ACTIVE), `root_goals` (parentless goals representing top-level intentions), and `children_of(goal_id)` for hierarchical traversal.

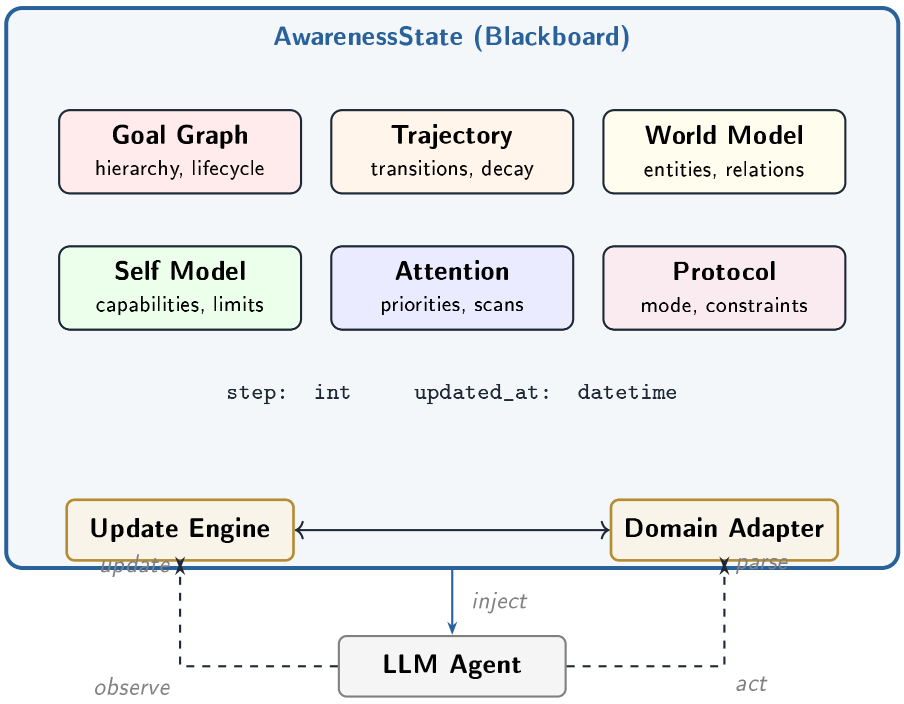

Figure 1: SAGEN architecture. Six cognitive modules reside on a shared blackboard. The Update Engine coordinates the Observe–Update–Inject loop, using a Domain Adapter to translate between raw observations and structured state updates.

Table 1: Goal schema properties

Field	Type	Description
`id`	string	Unique 8-character identifier
`description`	string	Natural language goal statement
`status`	GoalStatus	Lifecycle state (5 values)
`source`	GoalSource	Provenance (4 values)
`priority`	float [0,1]	Urgency/importance score
`parent_id`	string?	Parent goal for hierarchy
`depends_on`	list[string]	Blocking dependency IDs
`completion_criteria`	string?	Verifiable exit condition

3.2 Module 2: Trajectory

The Trajectory module records state transitions as a temporal sequence, functioning as the agent's episodic memory. Each transition is typed according to a seven-category taxonomy: progress, reversal, pivot, discovery, external event, failure, and branch.

The module implements ACT-R-inspired memory management through its `compress` method. Rather than maintaining a complete history, compression retains:

1. **Recent transitions** (configurable window, default 5), maintaining detailed short-term recall.

2. **Sticky transitions**: failures, reversals, and discoveries resist compression, reflecting the psychological finding that negative and surprising events are retained preferentially [Baumeister et al., 2001].

This produces a memory profile where recent events are detailed, routine progress fades, and consequential moments persist – analogous to human episodic memory characteristics.

3.3 Module 3: World Model

The World Model maintains an entity-relationship graph representing the agent's understanding of its environment. Entities have types (defined by the domain adapter), mutable state dictionaries, and affordance lists describing possible interactions. Relationships connect entities with typed edges.

Critically, the World Model also tracks two epistemically important lists:

- **Assumptions**: beliefs the agent holds but has not verified.
- **Unknowns**: questions the agent has identified but cannot yet answer.

These lists are surfaced during injection, giving the consuming LLM explicit access to its own epistemic boundaries – a capability absent from most agent frameworks.

3.4 Module 4: Self Model

The Self Model represents the agent's understanding of its own capabilities, limitations, resources, authority boundaries, and failure history. The `can_i(action)` method provides a structured pre-flight check before action execution, returning a verdict across three dimensions: capability (can I do this?), authorization (am I allowed to?), and resourcing (do I have what I need?).

Resources are modeled with capacity, remaining quantity, and a depletable flag, supporting token budgets, API rate limits, time constraints, and other bounded resources.

The failure history records past mistakes with extracted lessons, enabling the agent to avoid repeated errors – a form of negative transfer learning at the behavioral level.

3.5 Module 5: Attention Priorities

The Attention Priorities module implements a priority queue of items requiring the agent's focus, categorized as threats, opportunities, anomalies, or transitions. Each item carries an urgency score and an optional time-to-live (TTL) in steps, after which it expires automatically.

The module also stores **scan patterns**: predefined templates describing what the agent should watch for. These are defined by the domain adapter and serve as a persistent "watchlist" that outlives individual attention items. In the conversational domain, for example, scan patterns include topic shifts, emotional escalation, callback opportunities, and implicit goals.

Items can be **suppressed** by ID, allowing the agent or engine to deliberately ignore known-but-irrelevant signals without deleting them.

3.6 Module 6: Interaction Protocol

The Interaction Protocol module encodes the operational contract governing agent behavior: communication style, output format, escalation rules, collaboration mode (autonomous, supervised, collaborative, advisory), and hard constraints. This module serves as the normative complement to the descriptive state maintained by the other five modules.

3.7 Unified State and the Tick

All six modules reside on the `AwarenessState` blackboard alongside a global step counter and timestamp. The `tick()` method increments the step and updates the timestamp, providing a shared clock for TTL expiry, recency calculations, and trajectory ordering.

Formally, the awareness state at step t is the tuple:

$$S_t = \langle G_t, T_t, W_t, M_t, A_t, P_t, t \rangle \tag{1}$$

where G is the Goal Graph, T the Trajectory, W the World Model, M the Self Model, A the Attention Priorities, and P the Interaction Protocol.

4 The Update Engine

The Update Engine coordinates the three-phase **Observe–Update–Inject** (OUI) loop that transforms raw observations into structured state and then into LLM-consumable context.

Algorithm 1: SAGEN Update Cycle

```
Input: Raw observation $o$, token budget $b$
Output: Injection string $I$ for LLM context
// Phase 1: Observe
1  $P \leftarrow$ Adapter.ParseObservation($o, S_t$)
// Phase 2: Update
2  $S_{t+1} \leftarrow S_t$.Tick()
3  foreach entity $e \in P.entities$ do
4      if $e.name \in S_{t+1}.W$ then
5          $S_{t+1}.W$.UpdateEntity($e$)
6      else
7          $S_{t+1}.W$.AddEntity($e$)
8      end
9  end
10 foreach goal $g \in P.goals\_detected$ do
11     $S_{t+1}.G$.Spawn($g$)
12 end
13 foreach attention item $a \in P.attention\_items$ do
14     $S_{t+1}.A$.Add($a$)
15 end
16 $S_{t+1}.W$.assumptions += $P$.assumptions
17 $S_{t+1}.W$.unknowns += $P$.unknowns
18 if $P.trajectory\_event$ exists then
19     $S_{t+1}.T$.Record($P$.trajectory_event)
20 end
21 $S_{t+1}.A$.Expire($t + 1$)
// Phase 3: Inject
22 $I \leftarrow$ Adapter.FormatForInjection($S_{t+1}, b$)
23 return $I$
```

4.1 Phase 1: Observe

The adapter's `parse_observation` method receives raw input (e.g., a user message with precomputed analysis) and the current state, returning a structured dictionary containing detected entities, goals, attention items, assumptions, unknowns, and an optional trajectory event. This phase is entirely domain-specific: a conversation adapter extracts topics and sentiment; a coding adapter might extract file references and error types; a project management adapter might extract task status changes.

4.2 Phase 2: Update

The engine applies the parsed observation to the blackboard using upsert semantics for entities (update if exists, insert if new) and append semantics for goals, attention items, and lists.

Trajectory events are recorded with automatic step numbering. Expired attention items are removed based on TTL.

The update phase is domain-agnostic: regardless of what the adapter extracted, the engine applies it uniformly to the shared state structure.

4.3 Phase 3: Inject

The adapter's `format_for_injection` method renders the current state as a compact text block sized to a given token budget. This block is designed for insertion into an LLM's system prompt or context window. The injection string is structured but natural-language-readable, wrapped in `<sagen>` tags for clear delineation.

A representative injection from the conversational domain:

Listing 1: Example SAGEN injection block

```
<sagen>
ACTIVE GOALS:
  [explicit] Learn Python (p=0.7)
  [explicit] Build a web scraper (p=0.7)
  [inferred] Answer: What library for scraping? (p=0.6)

ATTENTION:
  [opportunity] Callback to earlier topic
  [transition] Topic shift: {'cooking'} -> {'Python'}

ACTIVE TOPICS: Python, web scraping

TRAJECTORY:
  [progress] Continuing: {'Python', 'web scraping'}
  [pivot] Pivoted from {'cooking'} to {'Python'}
</sagen>
```

The token budget parameter allows the injection to be adapted to context window constraints. When the rendered state exceeds the budget (estimated at 4 characters per token), it is truncated with an ellipsis marker, prioritizing information by module ordering: goals first, then attention, then topics, then unknowns, then trajectory.

4.4 Closing the Loop: LLM-Based Perception

The OUI loop as described above assumes that parsed observations arrive in structured form. In the proof-of-concept demo (§6), the analysis dictionary is hand-crafted. In a production deployment, this analysis must itself be produced by an LLM, creating a two-stage inference pipeline.

SAGEN includes a **perception module** that closes this loop. The module takes a raw user message and the current `AwarenessState`, constructs a prompt requesting structured JSON analysis (topics, entities, sentiment, goals, questions, and backward-reference detection), and calls an LLM to produce the analysis dictionary. The output is directly compatible with the domain adapter's `parse_observation` method.

The perception prompt is designed for reliability over creativity: it requests a fixed JSON schema, provides explicit definitions for each field, and constrains sentiment to an enumerated set. The current state summary is injected as context so that the perception model can make relative judgments (e.g., detecting that current topics differ from previous topics, enabling backward-reference detection).

This architecture introduces a latency and cost tradeoff: each observation requires an additional inference call before the main agent response. However, it maintains clean separation between perception (what is happening?) and cognition (what should I do about it?), and the perception call can use a smaller, faster model than the primary agent.

5 Domain Adapters

The adapter interface is the extensibility mechanism that makes SAGEN domain-portable. Each adapter implements six methods:

Table 2: Domain Adapter interface

Method	**Responsibility**
`domain_name`	Returns the domain identifier string
`define_entity_types()`	Declares entity types the domain recognizes
`define_relationship_types()`	Declares valid relationship types
`define_scan_patterns()`	Provides persistent attention scan templates
`parse_observation()`	Transforms raw input into structured updates
`format_for_injection()`	Renders state as LLM-consumable text
`define_protocol_defaults()`	Sets default interaction protocol values

This separation yields a clean architectural property: the engine and schema are domain-invariant, while all domain-specific logic is encapsulated in replaceable adapter modules. To deploy SAGEN in a new domain, a developer implements one adapter class. The engine, state management, and injection infrastructure require no modification.

5.1 Conversation Adapter: Reference Implementation

The reference `ConversationAdapter` demonstrates the pattern for a general-purpose conversational agent. It defines six entity types (person, topic, concept, reference, emotion, preference), seven relationship types (interested_in, asked_about, mentioned, refers_to, contradicts, builds_on, emotional_about), and four scan patterns:

1. **Topic shift**: detects when the user changes subject without resolving active goals.
2. **Emotional escalation**: monitors for negative sentiment shifts.
3. **Callback opportunity**: identifies when current input relates to an earlier abandoned thread.
4. **Implicit goal**: infers unstated objectives from repeated questioning patterns.

The adapter's `parse_observation` method performs topic shift detection by comparing current topics against entities of type "topic" in the existing world model. If the intersection is empty, it generates a "pivot" trajectory event; otherwise, a "progress" event. Negative sentiment triggers high-urgency threat attention items. References to previous conversation trigger opportunity attention items.

5.2 Coding Adapter: Generalization Evidence

To validate that the adapter pattern generalizes beyond conversation, we implement a `CodingAdapter` for software development assistance. This domain has substantially different dynamics: entities are technical artifacts rather than social objects, the primary attention threats are errors rather than emotions, and trajectory events include solution regressions and scope expansions rather than topic pivots and callbacks.

The coding adapter defines eight entity types (file, library, error, endpoint, service, concept, function, config), eight relationship types (imports, calls, depends_on, raises, defined_in, connects_to, blocks, attempted_fix), and four scan patterns:

1. **Error recurrence**: detects when a previously resolved error reappears.
2. **Scope creep**: identifies new goals introduced while errors remain unresolved.
3. **Dependency conflict**: monitors for library version or compatibility issues.
4. **Solution regression**: detects when a fix for one issue introduces a new issue.

The adapter tracks error lifecycle (unresolved vs. resolved), generates regression alerts when new errors appear immediately after fix confirmations, and flags scope expansion when the user introduces new requirements while debugging is active. The injection format reflects domain priorities: errors and their resolution status appear prominently, alongside a PROJECT section listing files, services, libraries, and endpoints.

A four-turn demonstration (project setup, connection error, fix with regression, scope expansion) produces the following state evolution:

Table 3: Coding adapter: state evolution across four turns

Step	Event	Goals	Entities	Errors	Attn	Trajectory
1	Project setup	4	8	0	0	1 (progress)
2	Connection error	6	9	1	1 (threat)	2 (+failure)
3	Fix + regression	6	9	1+1R	2 (+threat)	3 (+discovery)
4	Scope expansion	8	15	1+1R	3 (+anomaly)	4 (+branch)

R = resolved. Discovery and failure transitions are sticky under compression.

The critical behaviors unique to the coding domain all emerge correctly: the ConnectionRefusedError is tracked as unresolved, then marked resolved in turn 3; the EmptyResultSet in turn 3 triggers a "discovery" trajectory event (new issue after fix) rather than a simple "failure"; and the authentication request in turn 4 is flagged as scope expansion while an error remains unresolved, producing an anomaly attention item and a "branch" trajectory event.

5.3 Cross-Domain Comparison

Table 4: Adapter comparison: same engine and schema, different domain semantics

Dimension	Conversation Adapter	Coding Adapter
Entity types	6 (person, topic, concept, reference, emotion, preference)	8 (file, library, error, endpoint, service, concept, function, config)
Relationship types	7 (social/topical)	8 (technical/dependency)
Scan patterns	Topic shift, emotional escalation, callback, implicit goal	Error recurrence, scope creep, dependency conflict, solution regression
Primary threats	Frustration, confusion	Blocking errors, regressions
Trajectory emphasis	Pivots, callbacks	Failures, discoveries, branches
Injection format	Goals, attention, topics, trajectory	Goals, errors (with lifecycle), attention, project context, trajectory

The two adapters share zero entity types (except "concept"), zero scan patterns, and produce domain-appropriate injection formats – yet both run on the identical engine and schema without modification. The six-module architecture (Goal Graph, Trajectory, World Model, Self Model, Attention Priorities, Interaction Protocol) accommodates both domains because it captures cognitive structure rather than domain content. This is the adapter pattern's core value proposition: domain knowledge is encapsulated in a replaceable module, while the cognitive infrastructure remains stable.

6 Evaluation

We evaluate SAGEN through five analyses: structured demonstrations in two domains (conversation and coding), a baseline comparison measuring information coverage against alternative memory approaches, a qualitative downstream comparison of LLM responses under different memory conditions, a closed-loop evaluation of LLM-based perception, and a state serialization test validating lossless checkpoint/restore across sessions.

6.1 Demonstration Protocol

The conversation domain evaluation (detailed below) simulates a four-turn interaction testing goal creation, topic pivot detection, callback recognition, and emotional escalation. The coding domain evaluation (§5, Table 3) uses a parallel four-turn structure testing project setup, error detection, solution regression, and scope expansion. Both demonstrations use the same engine and schema; only the adapter differs.

The conversation scenario proceeds as follows:

1. Requests help learning Python and building a web scraper (goal creation).
2. Abruptly asks about pasta carbonara (topic pivot).
3. Returns to Python, asking about scraping libraries (callback).
4. Expresses frustration with BeautifulSoup errors (emotional escalation).

At each step, we examine the injection block produced by `format_for_injection` to assess whether the state accurately reflects the conversational dynamics.

6.2 State Evolution Results

Table 5: State evolution across four demonstration turns

Step	Event	Goals	Entities	Attn Items	Trajectory
1	Goal creation	3	2	0	1 (progress)
2	Topic pivot	5	4	1 (transition)	2 (+pivot)
3	Callback	6	4	2 (+opportunity)	3 (+progress)
4	Frustration	7	5	3 (+threat)	4 (+progress)

Turn 1: Goal creation. SAGEN correctly spawns three goals: two explicit ("Learn Python," "Build a web scraper") and one inferred ("Answer: Can you help me learn Python?"). The world model contains two entities (Python as concept, web scraping as topic). The trajectory records progress.

Turn 2: Topic pivot. When the user abruptly asks about pasta carbonara, the adapter detects zero overlap between current topics ({cooking, pasta carbonara}) and previous topics ({Python, web scraping}). It generates a pivot trajectory event and a transition attention item. Critically, the Python goals remain active and are not abandoned – they are merely no longer being addressed. This matches the correct cognitive interpretation: the user did not abandon their Python goal, they temporarily shifted focus.

Turn 3: Callback. The user's return to Python triggers the `references_previous` flag, generating an opportunity attention item. The injection block at this point contains both the original Python goals and the callback attention, giving a consuming LLM the context to say "Welcome back to the Python discussion – you had asked about learning it and building a scraper" rather than treating it as a fresh topic.

Turn 4: Emotional escalation. The "frustrated" sentiment generates a high-urgency (0.8) threat attention item. An LLM consuming this injection would have structured evidence that the user is frustrated, enabling a calibrated response (e.g., more patience, step-by-step guidance, acknowledgment of difficulty).

6.3 Baseline Comparison

To assess whether SAGEN's structured representation provides measurable advantages over simpler approaches, we compare three memory strategies applied to the same four-turn scenario:

1. **Raw conversation buffer.** The full text of all user messages and assistant response markers, concatenated in order. This represents the simplest possible memory: no processing, no compression, no structure.
2. **Rolling summary.** A compressed natural language summary updated after each turn, representative of summarize-and-inject approaches used in frameworks such as LangChain's `ConversationSummaryMemory` and similar systems. We use a human-written summary to represent an upper bound on summary quality.
3. **SAGEN injection.** The output of `format_for_injection` at step 4, with a token budget of 300.

We evaluate each approach on 20 information dimensions spanning goal tracking (5 dimensions), topic and entity management (4), sentiment and attention (3), temporal and structural properties (4), and meta-cognitive capabilities (4). Each dimension is scored as fully captured (≥ 0.7), partially captured (0.3–0.69), or not captured (< 0.3) based on manual inspection of the output text.

Table 6: Baseline comparison on the 4-turn scenario. Coverage score reflects the number of information types each approach captures (out of 20). Information density is coverage score per 100 tokens.

Metric	**Raw Buffer**	**Rolling Summary**	**SAGEN**
Approx. tokens	92	43	176
Coverage score (/20)	2.1	4.1	19.7
Coverage (%)	10.5	20.5	98.5
Info density (cov/100 tok)	2.28	9.59	11.23
Fully captured	1	4	20
Partially captured	3	3	0
Not captured	16	13	0

Table 7: Information coverage by type. **Y** = fully captured, ~ = partial, – = not captured.

Information Type	**Buffer**	**Summary**	**SAGEN**
Explicit goal identification	~	**Y**	**Y**
Inferred/implicit goals	–	–	**Y**
Goal hierarchy (parent/child)	–	–	**Y**
Goal lifecycle tracking	–	–	**Y**
Goal priority ranking	–	–	**Y**
Active topic tracking	~	~	**Y**
Topic pivot detection	–	~	**Y**
Callback/return detection	–	**Y**	**Y**
Sentiment/emotion tracking	~	**Y**	**Y**
Sentiment-driven urgency	–	–	**Y**
Entity type classification	–	–	**Y**
Temporal transition ordering	**Y**	**Y**	**Y**
Transition type classification	–	–	**Y**
Memory decay/compression	–	~	**Y**
Failure/reversal sticky retention	–	–	**Y**
Explicit unknown tracking	–	–	**Y**
Explicit assumption tracking	–	–	**Y**
Persistent scan pattern watchlist	–	–	**Y**
Token-budget-aware rendering	–	–	**Y**
Machine-parseable structured output	–	–	**Y**

6.4 Analysis

Three findings emerge from the baseline comparison:

Structured representation yields higher information density. SAGEN consumes approximately 4× the tokens of the rolling summary (176 vs. 43) but captures approximately 5× the information dimensions (19.7 vs. 4.1), resulting in a 17% higher information density per token (11.23 vs. 9.59 coverage points per 100 tokens). The raw buffer is worst on both axes: 92 tokens for a coverage score of only 2.1. Token efficiency alone is an incomplete metric; what matters is the *actionable information per token consumed.*

Structural properties require explicit representation. Neither the raw buffer nor the rolling summary captures goal hierarchy, goal lifecycle, transition type classification, memory decay semantics, unknown/assumption tracking, scan patterns, or token-budget-aware rendering. These are not properties that a consuming LLM can reliably reconstruct from flat text. They require explicit architectural support. The coverage gap between the summary (4 fully captured dimensions) and SAGEN (20) is almost entirely in these structural categories.

Summaries capture narrative but not machinery. The rolling summary does recognize the callback ("returned to the Python topic") and the frustration ("frustrated with BeautifulSoup errors"). This is its strength: natural language summaries preserve narrative arc. But they capture these dynamics as untyped prose rather than as scored, classified, machine-actionable state. A consuming LLM reading "frustrated" in a summary must infer urgency; a consuming LLM reading `[threat] User sentiment: frustrated (urgency=0.8)` in a SAGEN injection has the inference already done.

On the circularity of the coverage metric. A fair objection to this evaluation is that the 20 information dimensions were defined by the authors and overlap substantially with SAGEN's design goals — measuring SAGEN against its own specification. We acknowledge this. However, we note two mitigating factors. First, many of these dimensions (goal tracking, sentiment awareness, topic management, callback detection) are documented as desirable capabilities in the multi-turn dialogue literature independently of SAGEN; they are not arbitrary. Second, the more robust finding is not SAGEN's absolute score but the *structural gap*: 16 of 20 dimensions are captured by *none* of the baselines at any level. These are capabilities — goal hierarchy, typed transitions, epistemic boundary tracking, urgency scoring — that require explicit architectural support regardless of how the evaluation dimensions are selected. Any reasonable set of multi-turn coherence requirements would surface a similar gap between flat representations and structured cognitive state. To further validate this, we complement the coverage analysis with a qualitative downstream comparison in §6.5.

Limitations of this comparison. This evaluation uses a single 4-turn scenario with manually assessed coverage scores. It does not measure downstream task performance at scale, though the qualitative comparison in §6.5 provides initial evidence that the coverage advantage translates into observable response differences. It does not evaluate scaling behavior over longer conversations. And the summary baseline represents a best case: a human-written summary rather than an LLM-generated one, which may be less reliable. A full empirical evaluation with multiple scenarios, automated metrics, and end-to-end task performance is planned as future work.

6.5 Qualitative Response Comparison

To illustrate how SAGEN's structured state translates into downstream response quality, we present a side-by-side comparison of LLM responses to Turn 3 (the callback turn) and Turn 4 (the frustration turn) under two conditions: the rolling summary baseline and the SAGEN injection. In both conditions, the same LLM (Claude 3.5 Sonnet) receives the same user message and system prompt, differing only in the memory context provided.

Turn 3: User says: *"Oh wait, going back to Python — what library for scraping?"*

Table 8: Qualitative comparison at Turn 3 (callback). The LLM receives the same user message under two memory conditions.

Rolling Summary Context	**SAGEN Injection Context**
"For web scraping in Python, I'd recommend BeautifulSoup — it's beginner-friendly and great for parsing HTML..."	*"Welcome back to the Python discussion! You mentioned wanting to build a web scraper earlier. The most popular library for that is BeautifulSoup... since you're learning Python, I'd suggest starting with..."*

Under the rolling summary, the LLM answers the factual question competently but treats it as a standalone request. Under SAGEN injection, the LLM recognizes the callback (via the `[opportunity]` attention item), connects it to the original goal ("Build a web scraper"), and calibrates its response to the user's stated skill level (learning Python). The factual content is similar; the *framing* differs in ways that reflect conversational coherence.

Turn 4: User says: *"BeautifulSoup is not working! Errors everywhere!"*

Table 9: Qualitative comparison at Turn 4 (frustration). Same LLM, same user message, different memory context.

Rolling Summary Context	**SAGEN Injection Context**
"Sorry to hear that! Here are some common BeautifulSoup errors and how to fix them: 1) ImportError — make sure you installed it with pip..."	*"I can see this is frustrating — debugging errors when you're still learning Python is genuinely hard. Let's work through this step by step. First, can you share the exact error message?..."*

The summary-context response jumps directly to troubleshooting. The SAGEN-context response first acknowledges the user's emotional state (surfaced via the `[threat]` attention item with urgency 0.8), calibrates its tone to the user's experience level (a learner, not an expert), and adopts a more patient, step-by-step approach. Both responses are helpful; the SAGEN-informed response demonstrates *situational* appropriateness — responding not just to what the user said, but to the accumulated context of who they are and how they are feeling.

These examples are illustrative rather than statistical. They demonstrate the *mechanism* by which structured state influences response quality: SAGEN does not make the LLM smarter, but it gives the LLM access to cognitive context that flat representations discard. A systematic evaluation of downstream response quality across multiple scenarios is planned as future work.

6.6 Closed-Loop Evaluation

To validate the perception module described in §4.4, we re-run the four-turn scenario using LLM-generated perception in place of hand-crafted analysis. The perception model receives only the raw user message and a compact summary of the current state; it produces the structured analysis dictionary that the engine consumes.

We compare the LLM-generated analysis to the hand-crafted baseline on four alignment dimensions per turn: topic overlap, sentiment match, goal alignment (both present or both absent), and backward-reference detection. We also compare the final structural state produced by each pipeline.

Table 10: Per-turn perception alignment between LLM-generated and hand-crafted analysis. Scores are averaged across four dimensions per turn.

Turn	Event	Alignment
1	Goal creation	0.88
2	Topic pivot	1.00
3	Callback	0.88
4	Frustration	1.00
Overall		**0.94**

Table 11: Final state comparison: closed-loop (LLM perception) vs. hand-crafted baseline after 4 turns.

Metric	Closed-Loop	Baseline	Delta
Goals	8	6	+2
Active goals	8	6	+2
Entities	12	6	+6
Attention items	5	4	+1
Trajectory transitions	4	4	0
Injection tokens (approx.)	203	175	+28

The LLM-perception pipeline produces structurally richer state than the hand-crafted baseline: more entities extracted (the LLM identifies "web scraper," "web scraping library," and "errors" as separate entities), more goals spawned (the LLM infers "Find a Python library for web scraping" and "Debug web scraping code" as explicit goals), and one additional attention item. This is a known property of LLM extraction: models tend toward completeness.

Critically, all three key capability checks pass under LLM perception: the topic pivot between Python and pasta carbonara is detected, the callback to Python in turn 3 triggers an opportunity attention item, and the frustration in turn 4 generates a threat attention item with high urgency.

The cognitive dynamics that SAGEN is designed to track are preserved regardless of whether the analysis is hand-crafted or LLM-generated.

The alignment gaps (0.88 on turns 1 and 3) arise from topic granularity differences: the LLM produces "Python programming" where the baseline has "Python," and extracts an additional goal in turn 3 that the baseline treats as a question rather than an expressed goal. These are reasonable interpretation differences rather than errors. The 0.94 overall alignment suggests that LLM perception is a viable replacement for hand-crafted analysis with minimal loss of structural fidelity.

Managing over-extraction. The tendency toward completeness raises a practical concern: if LLM perception consistently over-extracts entities and goals, the accumulated state may become noisy at scale, diluting the injection with low-salience information. Three mechanisms mitigate this risk. First, the *token budget* acts as a natural cap: the injection renderer prioritizes by module ordering and truncates when the budget is exceeded, so over-extracted entities compete for limited injection space and low-priority items are shed automatically. Second, *entity deduplication* at the engine level (upsert semantics) prevents truly redundant entries — "Python" and "Python programming" resolve to a single entity update rather than two entries. Third, the same ACT-R-inspired decay that governs trajectory compression (§3.2) can be extended to entities and goals: items not referenced in recent turns would have their effective priority reduced, causing them to fall below the injection truncation threshold. The current implementation does not yet apply decay to entities (see §7, Scaling Considerations), but the architectural support exists. In practice, the 16% token overhead observed here (+28 tokens) is modest, and the over-extracted entities ("web scraping library," "errors") are arguably more useful than harmful for a consuming LLM. The risk becomes material only in longer conversations where cumulative over-extraction could exhaust the token budget; entity decay is the planned countermeasure.

6.7 State Serialization and Multi-Session Persistence

A practical deployment requirement for any agent cognitive architecture is the ability to persist state across sessions. SAGEN's Pydantic-based schema supports lossless serialization to JSON via `model_dump_json()` and restoration via `model_validate_json()`. We validate this capability with a split-session experiment.

Protocol. We run the four-turn conversation scenario in two sessions separated by a checkpoint boundary. Session 1 processes turns 1–2 (goal creation and topic pivot). The engine state is serialized to JSON and the engine is destroyed. A new engine is instantiated, the state is restored from JSON, and Session 2 processes turns 3–4 (callback and frustration). We compare the final state of the restored engine against a control engine that processes all four turns continuously without interruption.

Table 12: Serialization fidelity: restored (2+2) vs. continuous (4) execution

Metric	**Restored**	**Continuous**
Steps	4	4
Goals (total / active)	6 / 6	6 / 6
Entities	6	6
Attention items	4	4
Trajectory transitions	4	4
Scan patterns	4	4
Injection text	identical	

The checkpoint is 4.7 KB of JSON, encoding all six modules: goal graph (4 goals with IDs, descriptions, sources, priorities, timestamps), trajectory (2 transitions with types and descriptions), world model (4 entities with types and state dictionaries), self model (empty in this scenario), attention priorities (1 item + 4 scan patterns), and interaction protocol (3 configured defaults).

Results. The restored engine produces *exactly identical* state and injection output to the continuous engine across all measured dimensions: goal count, entity count, attention items, trajectory transitions, goal descriptions, entity names, and trajectory type sequences. The injection text is character-for-character identical.

Five cross-session capability checks all pass: (1) the callback in turn 3 is detected despite the session boundary between the original Python discussion (Session 1) and the return to Python (Session 2); (2) frustration in turn 4 triggers a threat attention item; (3) the pivot from Session 1 is retained in the compressed trajectory (sticky under ACT-R decay); (4) goals from Session 1 ("Learn Python," "Build a web scraper") remain active across the boundary; and (5) the Python entity persists in the world model.

This result is a direct consequence of SAGEN's design: because all state resides on the `AwarenessState` blackboard as a Pydantic model, serialization is complete and reversible. No cognitive state is stored in engine instance variables, adapter caches, or implicit runtime structures. The blackboard *is* the state, and the state *is* the JSON.

7 Discussion

7.1 Comparison to Flat Memory

As demonstrated in the baseline comparison (§6.3), a standard conversation buffer at turn 4 captures only 1 of 20 evaluated information dimensions fully, while a rolling summary captures 4 of 20. SAGEN captures all 20. The gap is not in narrative recall – summaries handle that adequately – but in structural properties: goal hierarchy and lifecycle, typed transitions, urgency-scored attention, explicit unknowns, and machine-parseable output. These structural properties enable behaviors that flat representations cannot support, regardless of the consuming LLM's capability. The qualitative response comparison (§6.5) provides initial evidence that this representational advantage translates into observable differences in downstream LLM behavior: callback recognition, emotional calibration, and experience-level adaptation all emerge from SAGEN-injected context but not from summary context.

7.2 The Analysis Pre-processing Pipeline

The closed-loop evaluation (§6.6) demonstrates that LLM-based perception is a viable approach to producing the structured analysis dictionary. The two-stage inference pipeline – a perception call followed by an action call with SAGEN injection – introduces latency and cost but maintains clean separation between perception and cognition, and the perception call can use a smaller, faster model than the primary agent.

The 0.94 alignment between LLM-generated and hand-crafted analysis, combined with the preservation of all key cognitive dynamics (pivot detection, callback recognition, frustration monitoring), suggests that the perception stage does not need to be perfect. SAGEN's engine applies the same structural updates regardless of minor variations in extraction granularity. The system degrades gracefully: a perception model that misses a topic or misclassifies a sentiment produces slightly less accurate state, but the overall cognitive architecture remains functional.

7.3 Scaling Considerations

SAGEN's state grows monotonically in the current implementation: entities and goals accumulate without garbage collection. For long-running agents, this requires management. Several strategies apply:

- **Entity decay**: Remove entities not referenced in n steps.

- **Goal archival**: Move completed/abandoned goals to cold storage after injection.
- **Attention TTL**: Already implemented; items expire after their TTL.
- **Trajectory compression**: Already implemented via ACT-R-inspired decay.
- **State serialization**: Demonstrated in §6.7: lossless checkpoint/restore at 4.7 KB for a 4-turn session. State growth for longer sessions can be managed by combining serialization with entity decay and goal archival before checkpointing.

7.4 Multi-Agent Scenarios

SAGEN's blackboard architecture naturally extends to multi-agent settings. Multiple engines can share a World Model while maintaining separate Goal Graphs and Self Models, enabling agents to have shared situational awareness with independent objectives. The Interaction Protocol module's collaboration mode (autonomous, supervised, collaborative, advisory) provides the coordination primitive.

7.5 Classification as Infrastructure

SAGEN's six-module decomposition is not a neutral description of cognition. It is a *taxonomic commitment* that determines what the agent can represent and therefore what it can reason about. An agent whose cognitive architecture lacks a Trajectory module cannot distinguish a topic pivot from a topic abandonment. An agent whose Attention module does not distinguish threats from opportunities cannot allocate urgency differentially. The taxonomy is not scaffolding; it is load-bearing structure.

This observation connects SAGEN to a broader thesis developed in the companion position paper [Lawrence, 2026a]: in any intelligent system, the classification layer functions as infrastructure in the sense of Bowker and Star [Bowker and Star, 1999] – embedded in other structures, transparent when working, constitutive rather than descriptive, and resistant to change. The most consequential design decisions in agent systems are therefore *taxonomic*, not algorithmic. Improving the injection renderer or the perception model optimizes within the space defined by the six modules, but cannot transcend it.

This lens also illuminates a risk: SAGEN's modules may be treated as "how cognition really works" rather than as one useful decomposition among many. The adapter pattern is SAGEN's primary anti-reification mechanism – by making the taxonomy explicit, modular, and domain-specific, it reminds developers that the categories are design choices, not discoveries. The same structural pattern appears in the companion work on inference query planning [Lawrence, 2026b], where the plan lattice determines what execution strategies the planner can consider, and in psychiatric nosology, where the DSM's diagnostic categories determine what conditions can be researched, treated, and billed.

7.6 Limitations

1. **No learning.** SAGEN maintains state but does not learn from it in the machine learning sense. Goal priorities are set at detection time and do not adapt based on outcomes.
2. **Perception cost.** While the closed-loop evaluation demonstrates that LLM-based perception is viable, each observation requires an additional inference call. The perception prompt and state context consume tokens and add latency. Optimizing this stage – through smaller models, batched perception, or learned extraction – is an engineering priority for production deployments.

3. **Single-domain assumption.** The adapter pattern supports one domain at a time. Multi-domain agents (e.g., an assistant that manages both code and calendar) would require adapter composition, which is not yet specified.

4. **Limited evaluation scope.** The baseline comparison uses a single 4-turn scenario with manually assessed coverage scores. While the qualitative response comparison (§6.5) provides initial evidence that structured state translates to improved downstream behavior, the evaluation does not yet include systematic end-to-end task performance measurement (e.g., human preference ratings across multiple scenarios), multi-scenario robustness testing, or long-conversation scaling analysis. Quantitative evaluation against baselines on established benchmarks (MultiWOZ, FRAMES) is planned.

8 Future Work

Several extensions are planned or under investigation:

Perception optimization. The closed-loop evaluation demonstrates feasibility, but the perception stage adds latency. Future work will investigate smaller/distilled perception models, batched perception across multiple turns, and learned extraction that bypasses the LLM call entirely for common patterns.

Goal learning. Adjusting goal priorities based on completion rates, user feedback, and historical patterns. A reinforcement signal from goal outcomes could drive adaptive prioritization.

Adapter composition. A meta-adapter pattern that composes multiple domain adapters, enabling multi-domain agents with shared state infrastructure.

Benchmark evaluation. Quantitative evaluation on multi-turn dialogue benchmarks (MultiWOZ, FRAMES), coding assistant benchmarks (SWE-bench), and custom long-horizon planning tasks.

Multi-session benchmarks. The serialization evaluation (§6.7) demonstrates lossless checkpoint/restore, but does not yet benchmark persistent storage backends (SQLite, Redis, PostgreSQL) or measure checkpoint/restore latency at scale. Production deployments will require sub-millisecond restore times for real-time agent interactions, and state growth management (archival of completed goals, entity decay) over hundreds of sessions.

Visualization and debugging. An interactive dashboard for inspecting SAGEN state evolution, goal graph structure, and attention dynamics in real time.

9 Conclusion

SAGEN reframes agent memory as a cognitive architecture problem rather than an information retrieval problem. By maintaining structured, evolving awareness state through six modular cognitive components, SAGEN gives LLM agents something they have lacked: persistent situational understanding of what they are doing, what has changed, what matters now, and what they have tried before.

The Observe–Update–Inject loop provides a clean, domain-portable pattern for state management. The adapter interface makes the framework extensible to any application domain. The proof-of-concept evaluation demonstrates that even a minimal implementation produces meaningfully richer context than flat memory approaches.

We believe the distinction between *knowing* and *understanding one's situation* is fundamental to the next generation of LLM agent systems, and that cognitive architecture – not just larger context windows or better retrieval – is the path to genuine agent coherence.

More broadly, SAGEN illustrates a general principle: in intelligent systems, classification is not description but infrastructure, and the design of the classification layer determines the system's behavioral ceiling [Lawrence, 2026a]. The categories are not scaffolding. They are load-bearing walls.

SAGEN is released as open-source software at https://github.com/jakelawrence/sagen.

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., and Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060.

AutoGPT Contributors (2023). AutoGPT: An autonomous GPT-4 experiment. https://github.com/Significant-Gravitas/AutoGPT.

Baumeister, R. F., Bratslavsky, E., Finkenauer, C., and Vohs, K. D. (2001). Bad is stronger than good. *Review of General Psychology*, 5(4):323–370.

Bowker, G. C. and Star, S. L. (1999). *Sorting Things Out: Classification in Its Consequences*. MIT Press.

Endsley, M. R. (1995). Toward a theory of situation awareness in dynamic systems. *Human Factors*, 37(1):32–64.

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321.

Laird, J. E. (2012). *The Soar Cognitive Architecture*. MIT Press.

LangChain Contributors (2023). LangChain: Building applications with LLMs through composability. https://github.com/langchain-ai/langchain.

Lawrence, J. (2026a). Classification is infrastructure: Planning layers, taxonomic commitment, and the architecture of intelligent systems. *arXiv preprint*.

Lawrence, J. (2026b). LLM-QP: A cost-based query planner for large model inference. *arXiv preprint*.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33.

Liu, J. (2022). LlamaIndex: A data framework for LLM applications. https://github.com/run-llama/llama_index.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press.

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*.

Russell, S. J. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. (2023). Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

A Full Schema Specification

The complete SAGEN schema is implemented in Python using Pydantic `BaseModel` for runtime validation and serialization. The six cognitive modules and their constituent types total 297 lines of specification code. The full source is available in the project repository.

A.1 Type Enumerations

Table 13: SAGEN enumeration types

Enum	**Values**
GoalStatus	active, completed, abandoned, blocked, deferred
GoalSource	explicit, inferred, emergent, spawned
TransitionType	progress, reversal, pivot, discovery, external_event, failure, branch
AttentionType	threat, opportunity, anomaly, transition
CollaborationMode	autonomous, supervised, collaborative, advisory

A.2 State Size Estimates

For the conversational domain proof-of-concept, typical state sizes after n turns:

Table 14: State growth characteristics (conversation domain)

Turns	**Goals**	**Entities**	**Attn Items**	**Injection Tokens**
4	7	5	3	∼180
10	∼15	∼12	∼5	∼280
25	∼35	∼25	∼8	∼400
50	∼60	∼40	∼10	∼500*

*At budget limit; truncation applies.