Agent State in Agentic AI Systems

A Practitioner's Framework for Working Memory Design, Logging Strategy, and Multi-Agent Coordination

---

Author: Clarence "Faheem" Downs
Affiliation: Professor Bone Lab
Program: Johns Hopkins University | Agentic AI Certificate Program
Date: December 2025

Abstract

This paper develops a practitioner-oriented framework for agent state management in large language model (LLM)-based agentic systems. Building upon the Cognitive Architectures for Language Agents (CoALA) framework (Sumers et al., 2024), which established the theoretical foundations for language agent memory and decision-making, this work extends that foundation with actionable engineering guidance for state schema design, logging strategy, and production deployment. The paper's primary contributions are: (1) an operational interpretation of CoALA's working memory concept for software engineering contexts; (2) a novel taxonomy of four state update classes with corresponding logging policies; and (3) practical design patterns for single-agent and multi-agent state coordination. This work is positioned as a bridge between academic theory and engineering practice, intended for practitioners building production agent systems who seek grounded, systematic approaches to state management.

Keywords: agent state, working memory, agentic AI, CoALA, multi-agent systems, LangGraph, observability, state management

---

1. Introduction

The emergence of large language models (LLMs) as reasoning engines for autonomous agents represents a fundamental shift in AI system architecture. Unlike traditional software systems where state management follows well-established patterns, LLM-based agents present unique challenges: the reasoning core is stateless by design, yet sophisticated task execution requires maintaining context, tracking progress, and coordinating across multiple steps or agents (Wang et al., 2024; Guo et al., 2024).

The academic community has begun to address these challenges. Most notably, the Cognitive Architectures for Language Agents (CoALA) framework proposed by Sumers, Yao, Narasimhan, and Griffiths (2024) provides a comprehensive theoretical foundation, drawing parallels between classical cognitive architectures

(SOAR, ACT-R) and modern language agents. CoALA establishes key concepts including working memory, long-term memory taxonomies, action spaces, and decision-making cycles.

However, a gap remains between CoALA's theoretical elegance and the practical decisions facing engineers building production systems. Questions arise that theory alone does not answer:

- How should state schemas be designed before implementation begins?
- What information should be logged, and what should remain ephemeral?
- How should state be validated across multi-step workflows?
- What patterns enable reliable multi-agent coordination?

This paper addresses that gap by developing an engineering framework grounded in CoALA's theoretical foundations but oriented toward actionable implementation guidance. The contributions are:

1. **Operational definitions:** Translating CoALA's working memory concept into software engineering terms with explicit schema design principles
2. **A logging taxonomy:** A novel four-category classification of state updates with corresponding observability strategies (Section 5)
3. **Design patterns:** Practical patterns for state validation, multi-agent handoffs, and production deployment
4. **Integration with tooling:** Explicit connections to frameworks (LangGraph) and observability standards (OpenTelemetry)

The paper proceeds as follows: Section 2 reviews theoretical foundations, centering on CoALA while connecting to classical cognitive architectures and contemporary frameworks. Section 3 develops an operational definition of agent state. Section 4 addresses the relationship between state and observability. Section 5 presents the logging taxonomy. Section 6 examines multi-agent coordination. Section 7 provides production guidelines. Section 8 discusses limitations and future directions.

1.1 Scope and Positioning

This work is explicitly positioned as **applied research** bridging theory and practice. It does not claim to advance fundamental theory; rather, it interprets and extends existing theory (primarily CoALA) for engineering contexts. Novel contributions—particularly the logging taxonomy—are clearly marked as such and should be understood as practitioner frameworks requiring empirical validation rather than established scientific results.

---

2. Theoretical Foundations

2.1 Classical Cognitive Architectures

The challenge of managing state in intelligent systems predates LLMs by decades. Cognitive architectures developed in artificial intelligence and cognitive science provide foundational insights that remain relevant to contemporary agent design.

SOAR (State, Operator, And Result), developed by Laird, Newell, and Rosenbloom (1987) and refined over subsequent decades (Laird, 2012; Laird, 2022), models cognition through a production system operating over working memory. SOAR distinguishes between:

- **Working memory:** The active substrate maintaining situational awareness, including perceptual input, intermediate reasoning results, active goals, and buffers for interacting with long-term memory systems
- **Long-term memories:** Semantic (facts), episodic (experiences), and procedural (skills) knowledge that persists across tasks

Critically, in SOAR, working memory is not merely a buffer—it is the space where reasoning occurs. As Laird (2022) explains, all interactions with the environment occur through structures attached to the top state, and state changes are the primary mechanism through which cognition advances.

ACT-R (Adaptive Control of Thought—Rational), developed by Anderson and colleagues (Anderson, 1996; Anderson & Lebiere, 1998), presents a complementary modular architecture. ACT-R's production system operates over the contents of buffers that mediate between specialized modules. This buffer-mediated approach offers a key insight for LLM agent design: state should be structured, explicitly defined, and serve as the sole interface through which reasoning proceeds.

A systematic review of 84 cognitive architectures over 40 years (Kotseruba & Tsotsos, 2020) reveals that despite their diversity, successful architectures share common principles: state must be explicit rather than implicit, structured rather than amorphous, and designed to support both current task execution and meta-level reasoning about the task itself.

2.2 The CoALA Framework

The Cognitive Architectures for Language Agents (CoALA) framework (Sumers et al., 2024) represents the most comprehensive effort to date to systematize language agent design by drawing on cognitive architecture principles. Published in *Transactions on Machine Learning Research*, CoALA provides the theoretical foundation upon which this paper builds.

CoALA proposes that a language agent can be understood through three primary components:

Memory System: - **Working memory:** Short-term storage for information relevant to the current decision cycle, analogous to a "scratchpad" that persists across LLM calls - **Long-term memory:** Persistent storage including semantic memory (facts and knowledge), episodic memory (past experiences), and procedural memory (skills and procedures, potentially including the agent's own code)

Action Space: - **External actions:** Interactions with the environment (tool use, API calls, physical actions) - **Internal actions:** Operations on the agent's own memory (reasoning, retrieval, learning)

Decision-Making Cycle: - A structured loop alternating between planning (proposing and evaluating actions) and execution (carrying out selected actions)

CoALA's insight is that LLMs function analogously to production systems in classical architectures—they define distributions over changes to text, just as productions define possible string modifications. This suggests that control structures developed for production systems may be applicable to LLM-based agents.

This paper's relationship to CoALA: We adopt CoALA's conceptual framework, particularly its working memory concept, and develop operational guidance for implementing these concepts in software systems. Where CoALA asks "what components should a language agent have?", this paper asks "how should engineers implement those components in production code?"

2.3 The ReAct Paradigm

The ReAct (Reasoning and Acting) paradigm (Yao et al., 2023) demonstrates the practical value of interleaving reasoning and action in LLM agents. Presented at the International Conference on Learning Representations (ICLR 2023), ReAct showed that agents generating both reasoning traces and task-specific actions outperform those doing either in isolation.

In ReAct's formulation, reasoning traces help the model "induce, track, and update action plans as well as handle exceptions, while actions allow it to interface with external sources, such as knowledge bases or environments, to gather additional information" (Yao et al., 2023).

What ReAct demonstrates implicitly—and what CoALA theorizes explicitly—is the necessity of state representation to carry forward results across thought-action-observation cycles. The agent must track what has been tried, what information has been gathered, and what remains to be done. ReAct's success is evidence that well-managed working memory enables more effective agent behavior.

2.4 LangGraph and Contemporary Orchestration

LangGraph, developed by LangChain, operationalizes these theoretical insights through a graph-based orchestration framework treating state as a first-class architectural component (LangChain, 2024). Applications are represented as directed graphs where nodes are processing steps and edges define transitions, with explicit state flowing through the graph.

LangGraph introduces several concepts directly relevant to state management:

Concept	Description
StateGraph	Container modeling the application as nodes and edges
TypedDict State	Strongly-typed state schemas defining information flow
Checkpointers	Mechanisms persisting state for recovery and human-in-the-loop
Reducers	Functions defining how concurrent state updates merge

LangGraph's approach aligns with CoALA's framework: the StateGraph implements the decision-making cycle, TypedDict schemas structure working memory, and checkpointers enable persistence analogous to long-term memory formation.

2.5 Memory in Generative Agents

The "Generative Agents" work by Park et al. (2023) demonstrated sophisticated memory management in simulated social agents. Their architecture includes:

- **Memory stream:** Continuous log of experiences in natural language
- **Retrieval:** Selecting relevant memories based on recency, importance, and relevance
- **Reflection:** Periodic synthesis of memories into higher-level insights

This work illustrates the distinction between *logging everything* (the memory stream) and *using information effectively* (retrieval and reflection). The memory stream is comprehensive but not directly usable; it requires retrieval mechanisms to become actionable working memory for current decisions.

This distinction—between comprehensive recording and selective activation—informs the logging taxonomy developed in Section 5.

---

3. Defining Agent State: An Operational Interpretation

3.1 From Theory to Implementation

CoALA defines working memory as information "relevant to the current decision cycle" that "persists across LLM calls" and functions as "a scratch pad that can be used for reasoning" (Sumers et al., 2024). This paper operationalizes that definition for software engineering contexts:

> **Agent state** is a structured, typed data object representing the information an agent currently has access to for completing its task. It is explicitly defined by a schema, mutable across reasoning steps, and serves as the sole interface through which the LLM reasons about the task.

This operational definition emphasizes properties relevant to implementation:

Property	Theoretical Basis (CoALA)	Implementation Implication
Structured	Working memory as organized scratchpad	Use TypedDict, dataclass, or Pydantic schemas
Typed	Distinct memory components	Explicit type annotations for all fields
Mutable	State changes across decision cycles	Design for update patterns, not just initialization
Bounded	Finite working memory capacity	Include only task-relevant information
Observable	Enables reasoning and debugging	State must be serializable and inspectable

3.2 What Agent State Contains

Drawing from CoALA's working memory concept, agent state typically includes:

Task Context: - Goal or objective for the current task - Constraints and requirements - User preferences relevant to task execution

Progress Tracking: - Steps completed and their outcomes - Current phase or stage of execution - Remaining steps or open questions

Retrieved Information: - Documents, data, or facts gathered during execution - Tool outputs and API responses - Relevant context from long-term memory

Decision History: - Key decisions made and their rationale - Branch points taken in conditional logic - Errors encountered and recovery actions

Intermediate Results: - Partial outputs being constructed - Drafts, calculations, or analyses in progress - Hypotheses under consideration

3.3 What Agent State Is Not

Clarity requires distinguishing agent state from related concepts:

Agent state is not conversation history. While conversation may inform state, raw message logs are not structured for reasoning. State extracts and organizes relevant information from conversation.

Agent state is not the schema definition. The schema (TypedDict class) is the *blueprint*; state is the *runtime instance* with actual values. This parallels the class/instance distinction in object-oriented programming.

Agent state is not LLM internal representation. LLMs have no persistent internal state across API calls. What appears as "memory" in LLM conversations is actually context window content—information passed explicitly in each request.

Agent state is not long-term memory. Following CoALA's distinction, long-term memory persists across tasks and sessions; working memory (state) is task-scoped and potentially ephemeral.

Agent state is not logging. Logs observe state; they do not define it. This distinction is developed in Section 4.

3.4 The Primacy of State in Agent Reasoning

A key insight from cognitive architectures, formalized in CoALA, is that working memory defines the boundary of what an agent can reason about. We articulate this as a design principle:

> **Design Principle (State Primacy):** Information not represented in agent state is not available for agent reasoning. The state schema defines the agent's "cognitive horizon" for the current task.

Note: This principle is the author's formulation, synthesizing insights from CoALA and classical cognitive architectures. It is proposed as a useful design heuristic rather than an established theoretical result.

This principle has practical implications: - If retrieved documents are not stored in state, the agent cannot reference them - If previous decisions are not tracked, the agent cannot explain its reasoning - If error history is not maintained, the agent cannot learn from failures within a task - If progress is not marked, the agent cannot know where it is in execution

State schema design is therefore not an implementation detail—it is an architectural decision determining what the agent can and cannot do.

---

4. State and Observability

4.1 The Logging Question

A common question in agent development: "If state updates at every step, should we log every update?"

This question reveals a conceptual confusion between two distinct concerns:

- **State management:** Enabling the agent to reason effectively
- **Observability:** Enabling humans to understand, debug, and evaluate agent behavior

These concerns have different optimization targets:

Concern	Optimized For	Update Frequency	Persistence
State	Agent reasoning speed and relevance	Every step	Task-scoped
Observability	Debugging, evaluation, compliance	Selective	Long-term

4.2 Modern Observability Standards

The emerging consensus in AI observability, reflected in standards like OpenTelemetry's GenAI semantic conventions (OpenTelemetry, 2025) and commercial platforms (Datadog, Langfuse, Arize), emphasizes:

- **Comprehensive tracing:** Capture complete execution flows including LLM calls, tool invocations, and decision points
- **Structured telemetry:** Use standardized schemas for metrics, events, logs, and traces (MELT)
- **Sampling and filtering:** Manage volume through intelligent sampling rather than exclusion at source
- **Cost and performance tracking:** Monitor token usage, latency, and resource consumption

This represents a shift from the older pattern of selective logging toward comprehensive instrumentation with smart analysis.

4.3 Reconciling State and Observability

The relationship between state and observability can be understood as follows:

State is the agent's working memory—what it needs to reason. State should be lean, containing only task-relevant information.

Observability is the human operator's window into agent behavior. Observability infrastructure should capture comprehensive telemetry, then filter and analyze as needed.

The mistake is conflating these: trying to make state serve observability needs (leading to bloated state) or limiting observability to state contents (missing important behavioral signals).

4.4 Practical Recommendation

For production systems, we recommend:

1. **Design state for reasoning:** Include only information the agent needs for current decisions
2. **Instrument comprehensively:** Use OpenTelemetry or equivalent to capture full traces
3. **Sample intelligently:** Apply sampling strategies to manage storage and cost
4. **Analyze selectively:** Build dashboards and alerts focused on decision-relevant events

This approach keeps state lean while providing full observability when needed.

---

5. A Taxonomy of State Updates

This section presents a novel taxonomy classifying state updates by their observability requirements. This framework is the author's contribution, synthesizing insights from CoALA's memory taxonomy, production experience, and observability best practices. It should be understood as a proposed practitioner framework rather than an empirically validated classification.

5.1 The Four Categories

State updates can be classified into four categories based on their persistence requirements and observability value:

Category 1: Ephemeral Reasoning **Definition:** Internal, high-frequency state changes that exist only to reach the next reasoning step.

Examples: - Intermediate calculations during planning - Draft thoughts before final formulation
- Temporary variables in multi-step reasoning - Token-level generation states

Observability Policy: Do not persist to logs. These updates have high volume and low long-term value. They may be captured in development/debug modes but should not appear in production telemetry.

Rationale: Following CoALA's distinction between working memory and long-term memory, ephemeral reasoning represents the "scratchpad" activity that enables reasoning but need not be preserved.

Category 2: Decision-Relevant Updates **Definition:** State changes that affect which action the agent takes next.

Examples: - Plan modifications or revisions - Branch selection in conditional logic - Retry decisions after failures - Confidence threshold crossings - Tool selection decisions

Observability Policy: Log as structured events capturing the decision, its rationale, and the resulting action.

Example Event Structure:

```
{
  "event_type": "decision",
  "decision_id": "d-12345",
  "category": "plan_revision",
  "reason": "Initial search returned no results",
  "action_selected": "broaden_query",
  "timestamp": "2025-12-23T10:15:30Z"
}
```

Rationale: These events form the "decision trace"—the audit trail explaining why the agent behaved as it did. They are essential for debugging, evaluation, and compliance.

Category 3: External Interaction Updates **Definition:** State changes resulting from interaction with systems outside the agent.

Examples: - Tool calls and their responses - API requests and results - Database queries and returned data - User inputs and agent outputs - File system operations

Observability Policy: Always log with full request/response details (or hashes/summaries for large payloads).

Rationale: External interactions are the agent's interface with the world. They must be logged for: - **Debugging:** Reproduce failures by replaying interactions - **Cost tracking:** Monitor API usage and associated expenses - **Compliance:** Demonstrate what external systems were accessed - **Reproducibility:** Verify that inputs produce expected outputs

Category 4: Memory-Qualifying Updates **Definition:** State changes that should persist beyond the current task—candidates for long-term memory.

Examples: - User preferences discovered during interaction - Successful strategies worth remembering - Error patterns suggesting systematic issues - Corrections or feedback from users - Domain knowledge extracted during execution

Observability Policy: Log the event AND write to long-term memory storage.

Example Event Structure:

```
{
  "event_type": "memory_formation",
  "memory_type": "semantic",
  "content": "User prefers concise responses without extensive caveats",
  "confidence": 0.85,
  "source_task": "task-67890",
  "timestamp": "2025-12-23T10:20:00Z"
}
```

Rationale: Following CoALA's learning framework, these updates represent the agent's ability to improve through experience. They bridge working memory and long-term memory.

5.2 Summary Table

Category	Log?	Persist to Memory?	Frequency	Example
Ephemeral Reasoning	No	No	High	Intermediate calculation
Decision-Relevant	Yes (as event)	No	Medium	Plan change
External Interaction	Yes (full detail)	No	Medium	API call
Memory-Qualifying	Yes (as event)	Yes	Low	User preference

5.3 Applying the Taxonomy

When implementing state updates, ask:

1. **Does this change affect what action comes next?** → Decision-Relevant
2. **Does this involve an external system?** → External Interaction
3. **Should this be remembered for future tasks?** → Memory-Qualifying
4. **None of the above?** → Ephemeral Reasoning

This classification determines both logging behavior and persistence strategy.

5.4 Relationship to OpenTelemetry Standards

This taxonomy aligns with emerging OpenTelemetry GenAI semantic conventions:

- **Ephemeral Reasoning** → May be captured in detailed spans during debugging but typically excluded from production traces

- **Decision-Relevant** → Maps to agent decision spans with semantic attributes
- **External Interaction** → Maps to tool/function call spans with input/output attributes
- **Memory-Qualifying** → Represents a learning event that may trigger separate memory system instrumentation

The taxonomy provides conceptual clarity; OpenTelemetry provides implementation standards.

---

6. Multi-Agent State Coordination

6.1 The Coordination Challenge

When multiple agents collaborate, state management complexity increases significantly. The fundamental question: **How do agents share information while maintaining coherent individual reasoning?**

The multi-agent systems literature (Guo et al., 2024; Wooldridge, 2009) identifies several coordination models, each with implications for state architecture:

Model	Description	State Implications
Centralized	Single coordinator dispatches to workers	Coordinator maintains global state; workers have local state
Decentralized	Agents communicate peer-to-peer	Each agent maintains own state; explicit message passing
Hierarchical	Teams with local supervisors	Layered state with team-level and global-level views
Blackboard	Shared workspace all agents access	Single shared state object with concurrent access patterns

6.2 Shared State Architecture

LangGraph and similar frameworks typically implement a **shared state** model where all agents read from and write to a common state object. This approach, aligned with CoALA's working memory concept extended to multiple agents, requires:

State Schema Design:

```
class MultiAgentState(TypedDict):
    # Global coordination
    overall_goal: str
    current_phase: str
    agents_completed: list[str]
```

```python
    # Agent-specific sections
    researcher_output: Optional[ResearchResult]
    writer_output: Optional[str]
    reviewer_feedback: Optional[ReviewResult]

    # Shared artifacts
    final_deliverable: Optional[str]
    revision_count: int
```

Advantages: - All agents have consistent view of task progress - Simple handoff mechanism (write to state, next agent reads) - Natural support for supervisor/worker patterns

Challenges: - Schema grows with number of agents - Requires coordination to prevent conflicts - Single point of failure if state becomes corrupted

6.3 State Contracts

For reliable multi-agent coordination, we propose (as a novel contribution) the concept of **state contracts**—explicit specifications of what each agent reads and writes:

Contract Structure:

```
Agent: [Name]
Reads: [List of state fields consumed]
Writes: [List of state fields produced]
Preconditions: [What must be true before agent runs]
Postconditions: [What will be true after agent completes]
```

Example Contract:

```
Agent: Writer
Reads: research_findings, outline, style_preferences
Writes: draft_content, writing_status
Preconditions:
  - research_findings is not empty
  - outline contains at least 3 sections
Postconditions:
  - draft_content is non-empty string
  - writing_status in ["complete", "needs_revision"]
```

State contracts enable: - **Static validation:** Check that agent compositions are valid before runtime - **Clear debugging:** When handoffs fail, contracts identify which preconditions were violated - **Documentation:** Contracts serve as executable specifications - **Testing:** Agents can be tested in isolation against their contracts

6.4 Coordination Patterns

Several patterns have emerged for multi-agent state coordination:

Supervisor Pattern: A central supervisor agent maintains global state and dispatches tasks to worker agents. Workers return results to the supervisor, which updates shared state and determines next steps.

```
Supervisor    Worker A    Supervisor    Worker B    Supervisor

        updates state          updates state
```

Pipeline Pattern: Agents execute in sequence, each consuming predecessor output. State flows linearly with clear handoff points.

```
Agent A    Agent B    Agent C    Final Output

      state      state
```

Collaborative Pattern: Multiple agents operate on shared workspace simultaneously. Requires conflict resolution strategies: - **Last-write-wins:** Simple but may lose information - **Merge functions:** Custom logic combining concurrent updates - **Locking:** Exclusive access to state sections (reduces parallelism)

6.5 Validation at Handoffs

Multi-agent systems require validation at agent boundaries:

```python
def validate_handoff(state: MultiAgentState, from_agent: str, to_agent: str) -> bool:
    """Validate state before agent handoff."""

    contract = get_contract(to_agent)

    # Check preconditions
    for condition in contract.preconditions:
        if not evaluate_condition(condition, state):
            raise HandoffError(
                f"Precondition failed for {to_agent}: {condition}"
            )

    # Check required fields exist
    for field in contract.reads:
        if field not in state or state[field] is None:
            raise HandoffError(
                f"Required field missing for {to_agent}: {field}"
            )

    return True
```

This validation prevents cascading failures where one agent's incomplete output corrupts downstream agents.

---

7. Production Deployment Guidelines

7.1 State Schema Design Process

We propose a systematic process for state schema design:

Step 1: Task Analysis - What is the agent trying to accomplish? - What information does it need at each step? - What decisions must be traceable?

Step 2: Information Inventory - Inputs from users or upstream systems - Retrieved or generated intermediate results - Decisions and their rationale - Final outputs and deliverables

Step 3: Schema Definition - Define TypedDict or Pydantic schema - Annotate all fields with types - Identify optional vs. required fields - Specify reducers for concurrent updates

Step 4: Validation Rules - Preconditions for each processing step - Postconditions confirming step completion - Invariants that must always hold

Step 5: Observability Integration - Map state fields to telemetry events - Define what triggers logging - Configure sampling and retention

7.2 Design Checklist

Before implementing an agent, verify:

Core Structure: - [ ] Goal/objective field defined - [ ] Progress tracking mechanism included - [ ] All inputs explicitly represented - [ ] Intermediate results captured - [ ] Final output field specified

Technical Requirements: - [ ] Schema uses TypedDict, dataclass, or Pydantic - [ ] All fields have type annotations - [ ] Optional fields properly marked - [ ] Collections typed (list[T], dict[K, V]) - [ ] State serializable to JSON

Observability: - [ ] Logging category assigned to each update type - [ ] External interactions always logged - [ ] Decision events captured with rationale - [ ] Memory-qualifying updates route to persistence

Validation: - [ ] Preconditions defined for each step - [ ] Postconditions verify step completion - [ ] Error handling preserves valid state

7.3 State Persistence Strategies

Production systems require durable state:

Checkpointing: Periodically save complete state snapshots. LangGraph provides built-in checkpointers for Postgres, Redis, SQLite, and custom backends. Enables: - Recovery from failures - Human-in-the-loop pause/resume - Time-travel debugging

Event Sourcing: Store sequence of state-changing events. Reconstruct state by replaying events. Provides: - Complete audit trail - Ability to rebuild state at any point - Natural fit for the logging taxonomy

Hybrid Approach: Checkpoint periodically for fast recovery; maintain event log for auditability. Balances performance with completeness.

7.4 From Prototype to Production

Aspect	Prototype	Production
State Storage	In-memory dict	Database with transactions
Validation	Optional, print statements	Required at every transition
Logging	print() or basic logging	Structured telemetry (OpenTelemetry)
Serialization	json.dumps()	Versioned schemas with migration
Error Handling	Exceptions propagate	Graceful degradation, state preservation
Concurrency	Single-threaded	Proper locking or event sourcing

7.5 Observability Integration

For production observability, we recommend:

Instrumentation: - Use OpenTelemetry SDK with GenAI semantic conventions - Instrument all LLM calls with model, tokens, latency - Instrument all tool calls with inputs and outputs - Create spans for agent decision points

Dashboards: - Success/failure rates by agent and task type - Latency distributions across execution steps - Token usage and cost attribution - Error categorization and frequency

Alerting: - Elevated error rates - Latency threshold breaches - Unusual token consumption - State validation failures

---

8. Discussion

8.1 Summary of Contributions

This paper has developed a practitioner-oriented framework for agent state management, building on the theoretical foundations established by CoALA (Sumers et al., 2024). The primary contributions are:

1. **Operational interpretation of working memory:** Translating CoALA's theoretical concepts into software engineering terms with explicit schema design guidance
2. **Logging taxonomy:** A novel four-category classification (ephemeral, decision-relevant, external interaction, memory-qualifying) providing clear guidance on observability strategy
3. **State contracts for multi-agent systems:** A proposed mechanism for specifying and validating agent interactions in multi-agent architectures
4. **Design patterns and checklists:** Practical tools for state schema design, validation, and production deployment

8.2 Limitations

This work has several important limitations:

Lack of empirical validation: The logging taxonomy and design recommendations are based on synthesis of existing work and practical experience, not systematic empirical study. Validation against diverse agent architectures would strengthen these contributions.

LLM-centric scope: The framework assumes LLM-based agents. Applicability to other agent architectures (reinforcement learning, classical planning) is not addressed.

Evolving ecosystem: The agentic AI tooling landscape changes rapidly. Specific recommendations regarding LangGraph, OpenTelemetry, or other tools may require updates as these projects evolve.

Scalability unaddressed: The paper does not analyze performance characteristics of different state management approaches at scale, such as the overhead of validation or the storage requirements of comprehensive logging.

Security and privacy: State may contain sensitive information. The paper does not address encryption, access control, or privacy-preserving state management.

8.3 Future Directions

Several directions warrant further investigation:

Empirical validation: Systematic studies comparing state management approaches across different agent types, tasks, and scales

Standardization: Development of industry-standard state schemas and contracts that enable interoperability between agent frameworks

Automated schema inference: Tools that analyze agent code and task specifications to suggest appropriate state schemas

Privacy-preserving state: Techniques for maintaining useful state while protecting sensitive information, potentially drawing on federated learning or differential privacy

Cognitive fidelity: Deeper investigation of how well software state representations capture the cognitive processes they model, and whether closer alignment improves agent performance

8.4 Conclusion

Agent state management is foundational to building effective LLM-based agents. The theoretical groundwork laid by CoALA and classical cognitive architectures provides essential conceptual clarity. This paper has attempted to bridge that theory to practice, offering frameworks and patterns that engineers can apply directly.

The core insight is simple but consequential: **state schema design is architecture, not implementation detail.** The choice of what information flows through an agent system determines what that system can reason about, how it can be debugged, and whether it can be deployed reliably.

As agentic AI systems grow in sophistication and deployment, systematic approaches to state management will become increasingly critical. We hope this framework contributes to that emerging discipline.

References

Anderson, J. R. (1996). ACT: A simple theory of complex cognition. *American Psychologist, 51*(4), 355-365.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought.* Lawrence Erlbaum Associates.

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., & Zhang, X. (2024). Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680.*

Kotseruba, I., & Tsotsos, J. K. (2020). 40 years of cognitive architectures: Core cognitive abilities and practical applications. *Artificial Intelligence Review, 53*(1), 17-94. https://doi.org/10.1007/s10462-018-9646-y

Laird, J. E. (2012). *The Soar cognitive architecture*. MIT Press.

Laird, J. E. (2022). Introduction to Soar. *arXiv preprint arXiv:2205.03854*.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33*(1), 1-64. https://doi.org/10.1016/0004-3702(87)90050-6

LangChain. (2024). LangGraph documentation. https://langchain-ai.github.io/langgraph/

OpenTelemetry. (2025, March 12). AI agent observability - Evolving standards and best practices. *OpenTelemetry Blog*. https://opentelemetry.io/blog/2025/ai-agent-observability/

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. https://doi.org/10.1145/3586183.3606763

Sumers, T. R., Yao, S., Narasimhan, K., & Griffiths, T. L. (2024). Cognitive architectures for language agents. *Transactions on Machine Learning Research*. https://openreview.net/forum?id=1i6ZCvflQJ

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., Zhao, W. X., Wei, Z., & Wen, J. R. (2024). A survey on large language model based autonomous agents. *Frontiers of Computer Science, 18*(6), 186345. https://doi.org/10.1007/s11704-024-40231-1

Wooldridge, M. (2009). *An introduction to multiagent systems* (2nd ed.). Wiley.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *Proceedings of the International Conference on Learning Representations (ICLR 2023)*. https://arxiv.org/abs/2210.03629

Appendix A: State Schema Examples

A.1 Research Agent State

```python
from typing import TypedDict, Optional, Annotated
from datetime import datetime
import operator

class Source(TypedDict):
    """A source document retrieved during research."""
    title: str
    url: str
    snippet: str
```

```
    relevance_score: float

class ResearchAgentState(TypedDict):
    """
    State schema for a research agent.

    Follows CoALA working memory principles:
    - Contains only task-relevant information
    - Structured for agent reasoning
    - Supports progress tracking and decision tracing
    """

    # Task context
    research_question: str
    constraints: list[str]

    # Progress tracking
    steps_completed: Annotated[list[str], operator.add]
    current_phase: str  # "planning" | "searching" | "analyzing" | "synthesizing"

    # Retrieved information
    search_queries_tried: list[str]
    sources_found: list[Source]
    sources_selected: list[Source]

    # Analysis
    key_findings: list[str]
    contradictions_noted: list[str]
    gaps_identified: list[str]

    # Output
    draft_answer: Optional[str]
    final_answer: Optional[str]
    confidence: Optional[float]

    # Error handling
    errors_encountered: list[dict]
    retry_count: int
```

A.2 Multi-Agent Collaborative State

```
from typing import TypedDict, Optional, Literal
from enum import Enum

class AgentStatus(str, Enum):
    PENDING = "pending"
```

```
    RUNNING = "running"
    COMPLETE = "complete"
    FAILED = "failed"
    BLOCKED = "blocked"

class CollaborativeState(TypedDict):
    """
    State schema for multi-agent collaboration.

    Implements shared state pattern with agent-specific sections
    and coordination metadata.
    """

    # Global coordination
    task_id: str
    overall_goal: str
    deadline: Optional[str]

    # Phase management
    current_phase: Literal["research", "drafting", "review", "finalization"]
    phases_completed: list[str]

    # Agent status tracking
    researcher_status: AgentStatus
    writer_status: AgentStatus
    reviewer_status: AgentStatus

    # Agent outputs (structured for handoff)
    research_output: Optional[dict]  # Conforms to ResearchOutput schema
    draft_output: Optional[str]
    review_output: Optional[dict]   # Conforms to ReviewOutput schema

    # Iteration tracking
    revision_count: int
    max_revisions: int
    revision_history: list[dict]

    # Final deliverable
    final_output: Optional[str]
    approval_status: Literal["pending", "approved", "rejected"]
```

Appendix B: Validation Patterns

B.1 Step Validation

```
from typing import Callable
from dataclasses import dataclass

@dataclass
class StepValidator:
    """Validates state before and after agent steps."""

    preconditions: dict[str, Callable[[dict], bool]]
    postconditions: dict[str, Callable[[dict], bool]]

    def validate_preconditions(self, state: dict, step: str) -> list[str]:
        """Check preconditions for a step. Returns list of violations."""
        violations = []

        if step not in self.preconditions:
            return violations

        for name, check in self.preconditions[step].items():
            if not check(state):
                violations.append(f"Precondition '{name}' failed for step '{step}'")

        return violations

    def validate_postconditions(self, state: dict, step: str) -> list[str]:
        """Check postconditions after a step. Returns list of violations."""
        violations = []

        if step not in self.postconditions:
            return violations

        for name, check in self.postconditions[step].items():
            if not check(state):
                violations.append(f"Postcondition '{name}' failed for step '{step}'")

        return violations

# Example usage
research_validator = StepValidator(
    preconditions={
        "search": {
            "has_question": lambda s: bool(s.get("research_question")),
        },
```

```
        "analyze": {
            "has_sources": lambda s: len(s.get("sources_selected", [])) > 0,
        },
        "synthesize": {
            "has_findings": lambda s: len(s.get("key_findings", [])) > 0,
        },
    },
    postconditions={
        "search": {
            "found_sources": lambda s: len(s.get("sources_found", [])) > 0,
        },
        "analyze": {
            "extracted_findings": lambda s: len(s.get("key_findings", [])) > 0,
        },
        "synthesize": {
            "produced_answer": lambda s: s.get("final_answer") is not None,
        },
    }
)
```

B.2 Contract-Based Handoff Validation

```
@dataclass
class AgentContract:
    """Specifies an agent's state interface."""

    agent_name: str
    reads: list[str]          # State fields consumed
    writes: list[str]         # State fields produced
    requires_agents: list[str] # Agents that must complete first

    def validate_can_start(self, state: dict, completed_agents: list[str]) -> bool:
        """Check if this agent can start given current state."""

        # Check required agents have completed
        for required in self.requires_agents:
            if required not in completed_agents:
                raise ContractViolation(
                    f"{self.agent_name} requires {required} to complete first"
                )

        # Check required fields exist and are non-empty
        for field in self.reads:
            if field not in state or state[field] is None:
                raise ContractViolation(
                    f"{self.agent_name} requires field '{field}' but it is missing"
```

```
                )

        return True

    def validate_completed(self, state: dict) -> bool:
        """Check if this agent has fulfilled its contract."""

        for field in self.writes:
            if field not in state or state[field] is None:
                raise ContractViolation(
                    f"{self.agent_name} should have produced '{field}' but didn't"
                )

        return True

# Example contracts
CONTRACTS = {
    "researcher": AgentContract(
        agent_name="researcher",
        reads=["research_question"],
        writes=["research_output"],
        requires_agents=[],
    ),
    "writer": AgentContract(
        agent_name="writer",
        reads=["research_output", "overall_goal"],
        writes=["draft_output"],
        requires_agents=["researcher"],
    ),
    "reviewer": AgentContract(
        agent_name="reviewer",
        reads=["draft_output", "research_output"],
        writes=["review_output"],
        requires_agents=["writer"],
    ),
}
```

Appendix C: Logging Implementation

C.1 Structured Event Logging

```
import json
from datetime import datetime
from enum import Enum
from typing import Any
```

```python
from dataclasses import dataclass, asdict

class UpdateCategory(str, Enum):
    EPHEMERAL = "ephemeral"           # Don't log
    DECISION = "decision"             # Log as event
    EXTERNAL = "external_interaction" # Always log with detail
    MEMORY = "memory_qualifying"      # Log and persist

@dataclass
class StateEvent:
    """Structured event for state changes."""

    event_id: str
    timestamp: str
    category: UpdateCategory
    event_type: str
    agent_name: str
    details: dict[str, Any]

    def to_json(self) -> str:
        return json.dumps(asdict(self), default=str)

class StateLogger:
    """Logger implementing the four-category taxonomy."""

    def __init__(self, logger, memory_store=None):
        self.logger = logger
        self.memory_store = memory_store

    def log_update(self, category: UpdateCategory, event_type: str,
                   agent_name: str, details: dict) -> None:
        """Log a state update according to its category."""

        if category == UpdateCategory.EPHEMERAL:
            # Don't log ephemeral updates in production
            return

        event = StateEvent(
            event_id=self._generate_id(),
            timestamp=datetime.utcnow().isoformat(),
            category=category,
            event_type=event_type,
            agent_name=agent_name,
            details=details,
        )
```

```
        # Log the event
        self.logger.info(event.to_json())

        # For memory-qualifying updates, also persist
        if category == UpdateCategory.MEMORY and self.memory_store:
            self.memory_store.save(event)

    def log_decision(self, agent_name: str, decision_type: str,
                     reason: str, action_selected: str) -> None:
        """Convenience method for decision events."""
        self.log_update(
            category=UpdateCategory.DECISION,
            event_type="decision",
            agent_name=agent_name,
            details={
                "decision_type": decision_type,
                "reason": reason,
                "action_selected": action_selected,
            }
        )

    def log_tool_call(self, agent_name: str, tool_name: str,
                      inputs: dict, outputs: dict, latency_ms: float) -> None:
        """Convenience method for external interaction events."""
        self.log_update(
            category=UpdateCategory.EXTERNAL,
            event_type="tool_call",
            agent_name=agent_name,
            details={
                "tool_name": tool_name,
                "inputs": inputs,
                "outputs": outputs,  # Consider hashing large outputs
                "latency_ms": latency_ms,
            }
        )
```

Acknowledgments: This paper builds substantially on the CoALA framework developed by Sumers, Yao, Narasimhan, and Griffiths. The author thanks the developers of LangGraph, OpenTelemetry, and the broader agentic AI community for the tools and standards that informed this work.