Functional Stability and Adaptive Control in LLM-Based Computer Use Agents via Graph-Structured Persistent Memory

Danylo Vorvul *, Andrii Musienko, Iryna Galchenko, Mykola Myroniuk, Andrii Sobchuk

Posted Date: 16 April 2026

doi: 10.20944/preprints202604.1148.v1

Keywords: functional stability; adaptive control; decision-making theory; machine learning; graph memory; LLM agents; GUI automation; hierarchical control systems; knowledge reuse; directed graphs

 doi:10.20944/preprints202604.1148.v1

Article

Functional Stability and Adaptive Control in LLM-Based Computer Use Agents via Graph-Structured Persistent Memory

Danylo Vorvul [1,*], Andrii Musienko [1], Iryna Galchenko [1], Mykola Myroniuk [2] and Andrii Sobchuk [3]

[1] Department of Computer Science, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 03056 Kyiv, Ukraine
[2] Candidate of Military Sciences, Senior Researcher, Institute of Aviation and Air Defense, National Defense University of Ukraine, Kyiv, Ukraine
[3] Educational and Scientific Institute of Cybersecurity and Information Protection, State University of Information and Communication Technologies, 7 Solomyanska Str., 03110 Kyiv, Ukraine
* Correspondence: vorvul.danylo@gmail.com

Abstract

Large language model (LLM)-driven computer use agents (CUAs) automate graphical user interface (GUI) tasks but often re-solve previously encountered subtasks, increasing token use, latency, and instability. We address this limitation with a directed graph-based persistent memory in which nodes represent observable GUI states and edges encode executable action sequences. We formalize the memory-augmented agent as $\mathcal{S} = \langle A, \Sigma, \mathcal{G}, \delta, \pi, \Phi \rangle$, define stability conditions by analogy with functional stability theory, and derive token-cost efficiency bounds. In control-theoretic terms, the Manager–Worker architecture becomes a closed-loop system where memory provides experience-based feedback, and selecting between memory retrieval and fresh LLM planning is treated as adaptive control. Experiments on OSWorld show that the proposed agent cuts both LLM token consumption and execution time by about 50% versus a memoryless baseline while preserving comparable success rates (≈36.9% on 15-step and ≈46.9% on 50-step tasks). Structured graph memory therefore improves robustness under perturbation and supports convergent efficiency gains over time.

Keywords: functional stability; adaptive control; decision-making theory; machine learning; graph memory; LLM agents; GUI automation; hierarchical control systems; knowledge reuse; directed graphs

MSC: 68T42; 93C85; 68M15

1. Introduction

In the era of increasingly autonomous information systems, the stability and controllability of software agents that interact with complex, dynamic environments constitute fundamental challenges at the intersection of control theory, decision-making science, and artificial intelligence [6,8,20]. Large language model (LLM)-driven computer use agents (CUAs) represent a compelling instantiation of this challenge: they must reliably interpret graphical user interface (GUI) states, plan multi-step action sequences, and execute them in real desktop and web environments [1]. Despite significant progress, contemporary CUAs suffer from a critical stability deficit—they lack persistent memory and consequently "reinvent the wheel" on every task, repeatedly reasoning through previously solved subproblems [1,4]. This redundancy leads to excessive token consumption, increased latency, and, most importantly, diminished functional reliability over extended operation.

The concept of *functional stability*—broadly understood as the ability of an information system to perform its assigned tasks over a necessary period, at least partially, despite structural or environmental perturbations—has been extensively studied in the context of network architectures and distributed

 doi:10.20944/preprints202604.1148.v1

systems [6,8,9]. A classical criterion for functional stability requires that the vertex connectivity $\lambda(G)$ and edge connectivity $\chi(G)$ of the system's structural graph G both satisfy $\lambda(G) \geq 2 \wedge \chi(G) \geq 2$, ensuring a minimal reserve against single-point failures [6,7]. Although originally formulated for communication networks, the underlying principle—that structural redundancy in a system's knowledge graph provides robustness guarantees—admits a natural generalization to the memory architectures of autonomous agents.

From a control-theoretic perspective, contemporary CUA architectures such as Agent S2 [4] employ a hierarchical Manager–Worker decomposition that parallels classical cascade control [13]. The Manager interprets user instructions and decomposes tasks (the *controller*), while the Worker executes low-level GUI actions (the *plant/actuator*). However, without persistent memory, this control loop is entirely open: each task execution is independent, with no feedback from prior experience. The introduction of a memory graph closes this loop, creating an experience-based feedback mechanism that progressively improves the controller's decision quality—a structure analogous to model-reference adaptive control [11,12].

The agent's operational decision at each step—whether to retrieve a memorized action trajectory or invoke fresh LLM planning—constitutes a formal *decision-making problem* under uncertainty. This binary choice balances *exploitation* (reusing known solutions) against *exploration* (generating novel plans), a trade-off well studied in reinforcement learning and multi-criteria decision theory [14,15]. Our graph memory architecture provides a structured *decision support system* that reduces both the uncertainty and the computational cost of this decision by maintaining an indexed repository of verified action sequences.

Several prior works have explored memory mechanisms for GUI agents, though with notable limitations. MobileGPT/MemoDroid [2] augments a mobile automation agent with hierarchical memory, achieving approximately 70% reduction in LLM usage, but operates primarily as an action replay log without structured state representation. AppAgentX [3] compresses action trajectories into reusable macros, yet organizes knowledge as chronological chains rather than a navigable state graph. Agent S2 [4] advances perception and planning through a generalist–specialist composition and Mixture-of-Grounding but retains no long-term memory between tasks. Recent surveys on LLM agent memory [1,17] and specialized memory systems such as MemGPT [18] and A-Mem [19] highlight the general importance of persistent, structured memory but do not address the specific requirements of GUI state-action representation. Related work by the present authors on retrieval-augmented generation systems in a domain-specific analytical setting further underscores the value of structured retrieval and reusable knowledge organization [22]. This broader perspective motivated the graph-based architecture presented here.

The contribution of this paper is threefold. First, we propose a *directed graph-based persistent memory architecture* for CUAs in which nodes correspond to observable GUI states, edges encode executable action sequences, and parameterized task descriptors enable hierarchical reuse at both low-level (individual actions) and high-level (multi-step procedures) granularities. Second, we provide a *formal mathematical framework* that models the memory-augmented agent as an algebraic tuple, defines stability conditions by analogy with classical functional stability theory, and derives efficiency bounds on token consumption. Third, we present *experimental validation* on the OSWorld benchmark [5], demonstrating that the memory-augmented agent achieves approximately 50% reductions in both token cost and execution time with no degradation in task success rate, while exhibiting convergent efficiency improvement characteristic of a functionally stable system.

The remainder of this paper is organized as follows. Section 2 presents the formal mathematical model and system design, including the tuple-based formalization, stability definitions, and the cost model. Section 3 reports the experimental evaluation. Section 4 interprets the results through the stability and control frameworks. Section 5 summarizes the contributions and outlines future work.

 doi:10.20944/preprints202604.1148.v1

2. Materials and Methods

2.1. Formal Model of the Memory-Augmented Agent

We define the memory-augmented computer use agent as a formal system:

$$\mathcal{S} = \langle A, \Sigma, \mathcal{G}, \delta, \pi, \Phi \rangle, \tag{1}$$

where:

- A is the finite set of *available actions*, encompassing both primitive GUI operations (mouse clicks, keyboard inputs, scrolling) and composite tool invocations (parameterized macros such as `SearchDrive(query)`);
- Σ is the set of *observable GUI states*, where each state $\sigma \in \Sigma$ is characterized by a visual screenshot, OCR-extracted text, and accessibility-tree metadata;
- $\mathcal{G} = (N, E, D)$ is the *memory graph*—a directed graph with node set N, edge set $E \subseteq N \times N$, and task descriptor set D;
- $\delta : N \times A \to N$ is the *deterministic state transition function* that governs action replay from memory;
- $\pi : \Sigma \times \mathcal{G} \to A \cup \{\perp\}$ is the *decision policy*, which returns either a memorized action $a \in A$ or the null symbol $\perp$ to trigger fresh LLM planning;
- $\Phi : \Sigma \to N \cup \{\varnothing\}$ is the *state recognition function* that maps a current observation to its corresponding memory graph node, or to $\varnothing$ if no matching node exists.

2.2. Memory Graph Structure

The memory graph $\mathcal{G} = (N, E, D)$ is realized as a persistent directed graph stored in a graph database. Formally, each node $n \in N$ is a tuple:

$$n = (x, \mathcal{T}_n, A_n), \tag{2}$$

where x denotes the visual representation of the screen (a perceptual hash of the screenshot combined with extracted text features), $\mathcal{T}_n \subseteq D$ is the set of task descriptors executable from this state, and $A_n \subseteq A$ is the set of actions available at this node.

Each directed edge $e \in E$ takes the form:

$$e = (n_i, n_j, a_{ij}, c_{ij}), \tag{3}$$

where $n_i, n_j \in N$ are the source and target nodes, $a_{ij} \in A$ is the action (or action sequence) that transitions the system from state n_i to state n_j, and $c_{ij} \in \mathbb{R}_{\geq 0}$ is the associated execution cost (measured in tokens or time). In practice, a_{ij} may be a Python `pyautogui` script encoding a sequence of mouse clicks and keyboard inputs.

A task descriptor $d \in D$ is a parameterized functional definition:

$$d = (\text{name}, \text{params}, n_{\text{start}}, n_{\text{end}}, P_d), \tag{4}$$

where name is a human-readable identifier (e.g., `ExportReport`), params is a parameter schema, n_{start} and n_{end} are the initial and terminal nodes, and $P_d = (e_1, e_2, \ldots, e_k)$ is the ordered path of edges constituting the procedure. This hierarchical structure allows complex multi-step workflows to be stored as single callable procedures.

The graph is initialized with a single root node corresponding to the desktop home screen. As the agent completes tasks, new nodes and edges are added, and the Tool Generation module abstracts frequently traversed paths into reusable task descriptors, causing $\mathcal{G}$ to grow and become increasingly dense over time.

 doi:10.20944/preprints202604.1148.v1

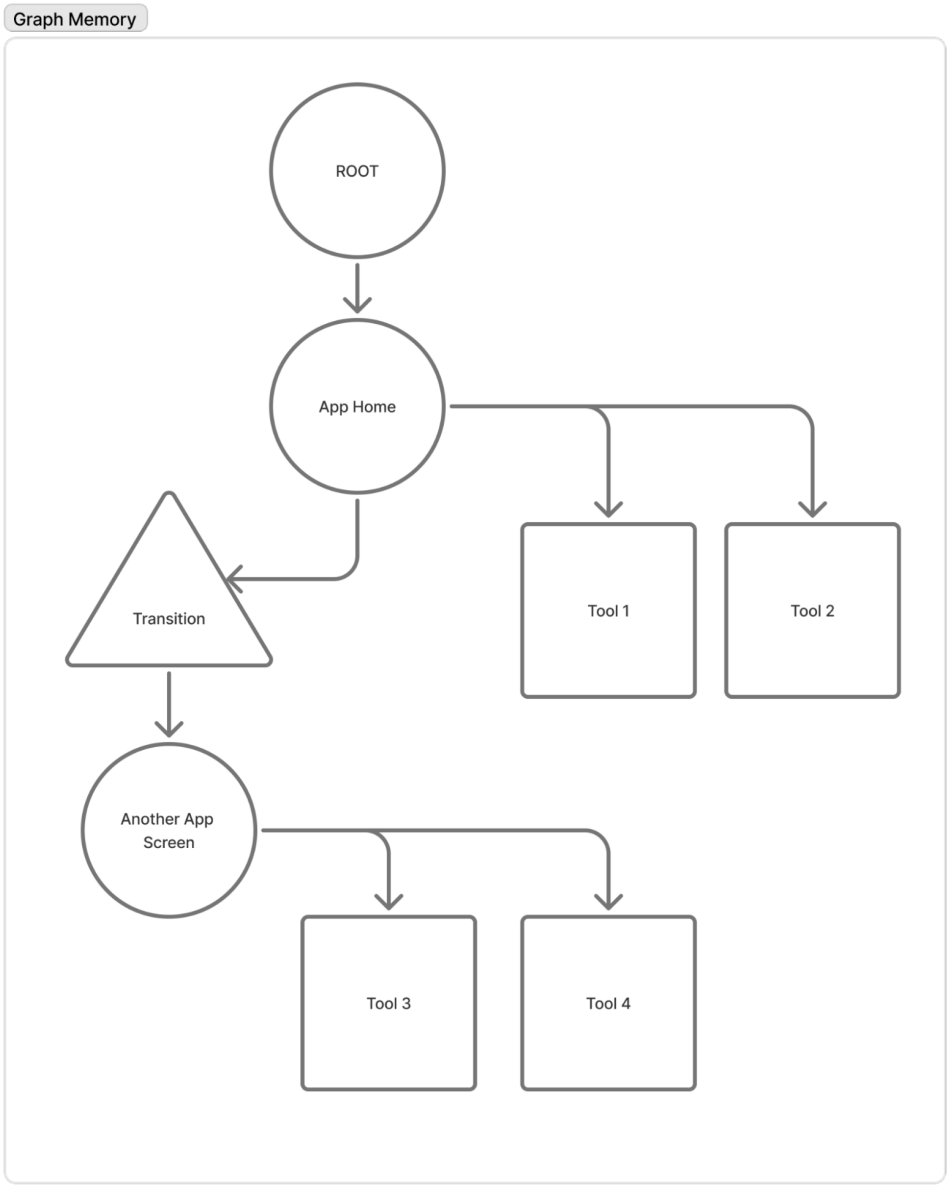

Figure 1. Directed graph-based persistent memory structure. Nodes encode observable GUI states with associated tools, while directed edges represent executable action transitions and reusable task trajectories.

2.3. Stability Conditions for the Memory Graph

Drawing on the theory of functional stability for information systems [6–8], we define stability conditions for the memory-augmented agent system.

Definition 1. ***(Functional Stability of the Memory Graph).*** *The memory graph $\mathcal{G} = (N, E, D)$ is said to be* functionally stable with respect to a task set $\mathcal{T}$ *if, for every task $\tau \in \mathcal{T}$, there exists at least one path $P_\tau = (n_0, n_1, \ldots, n_k)$ in $\mathcal{G}$ from a reachable initial node n_0 to a goal node n_k such that executing the edge actions along P_τ accomplishes τ.*

This definition parallels the classical requirement $\lambda(G) \geq 2$ for information systems [7], but adapted to the directed, task-oriented nature of the memory graph. We further define a quantitative stability measure:

Definition 2. ***(Memory Coverage Ratio).*** *For a task set $\mathcal{T}$ and memory graph $\mathcal{G}$, the* memory coverage ratio *is:*

 doi:10.20944/preprints202604.1148.v1

$$\rho(\mathcal{G}, \mathcal{T}) = \frac{|\{\tau \in \mathcal{T} : \exists P_\tau \text{ in } \mathcal{G}\}|}{|\mathcal{T}|}. \tag{5}$$

The system is α-stable *if* $\rho(\mathcal{G}, \mathcal{T}) \geq \alpha$ *for a prescribed threshold* $\alpha \in [0, 1]$.

Definition 3. ***(Convergence of Memory Utilization).*** *Let* $T = (t_1, t_2, \ldots)$ *be a sequence of tasks presented to the agent. Define the* memory utilization ratio *after* k *tasks as:*

$$\mu_k = \frac{\sum_{i=1}^{k} s_i^{\text{mem}}}{\sum_{i=1}^{k} s_i^{\text{total}}}, \tag{6}$$

where s_i^{mem} *is the number of steps in task* t_i *served from memory and* s_i^{total} *is the total number of steps. The memory system exhibits* convergent stability *if* $\{\mu_k\}$ *is a non-decreasing sequence bounded above by some* $\mu^* < 1$*:*

$$\mu_k \leq \mu_{k+1} \leq \mu^* \quad \forall\, k \geq k_0, \tag{7}$$

for some warm-up period $k_0 \in \mathbb{N}$.

This convergence criterion ensures that the agent progressively learns from experience—a hallmark of stable adaptive systems [11,12]—while the upper bound $\mu^* < 1$ reflects the irreducible novelty in any realistic task distribution.

2.4. Perturbation Robustness and Graceful Degradation

A key property of the memory-augmented system is its ability to degrade gracefully under perturbation, analogous to the resilience of functionally stable information systems [9,10]. Let σ' denote a perturbed observation (e.g., a modified UI layout) such that $\Phi(\sigma') = \varnothing$ (the state recognition function fails to match any known node). In this case, the decision policy defaults to:

$$\pi(\sigma', \mathcal{G}) = \bot, \tag{8}$$

triggering the standard LLM planning procedure. The agent thus never performs worse than the memoryless baseline: the memory graph provides an efficiency floor, not a ceiling. Formally, if $R_{\text{mem}}(\tau)$ and $R_{\text{base}}(\tau)$ denote the success indicators for a task τ under the memory-augmented and baseline agents, respectively, then:

$$\mathbb{E}[R_{\text{mem}}(\tau)] \geq \mathbb{E}[R_{\text{base}}(\tau)] \quad \forall\, \tau \in \mathcal{T}, \tag{9}$$

since every execution path available to the baseline agent is also available to the memory-augmented agent (via fallback), while the memory agent additionally has access to verified, potentially more efficient trajectories.

2.5. Token Cost Model

We define the *token cost function* for a task τ as:

$$C(\tau) = \sum_{i=1}^{|\tau|} c(s_i), \tag{10}$$

where $|\tau|$ is the number of steps in task τ and $c(s_i)$ is the token cost of step s_i. For the baseline agent, each step requires full LLM reasoning:

$$C_{\text{base}}(\tau) = \sum_{i=1}^{|\tau|} c_{\text{LLM}}(s_i), \tag{11}$$

 doi:10.20944/preprints202604.1148.v1

where $c_{\text{LLM}}(s_i)$ includes prompt construction, chain-of-thought reasoning, and action generation.

For the memory-augmented agent, a step served from memory incurs only a retrieval cost $c_{\text{ret}} \ll c_{\text{LLM}}$:

$$C_{\text{mem}}(\tau) = \sum_{i=1}^{|\tau|} [\mathbb{1}[\pi(\sigma_i, \mathcal{G}) \neq \perp] \cdot c_{\text{ret}} + \mathbb{1}[\pi(\sigma_i, \mathcal{G}) = \perp] \cdot c_{\text{LLM}}(s_i)], \tag{12}$$

where $\mathbb{1}[\cdot]$ is the indicator function. Defining the fraction of memory-served steps as $\mu_\tau = \frac{|\{\sigma_i : \pi(\sigma_i, \mathcal{G}) \neq \perp\}|}{|\tau|}$, we obtain:

$$\frac{C_{\text{mem}}(\tau)}{C_{\text{base}}(\tau)} \leq 1 - \mu_\tau \left(1 - \frac{c_{\text{ret}}}{\bar{c}_{\text{LLM}}}\right), \tag{13}$$

where $\bar{c}_{\text{LLM}}$ is the average per-step LLM cost. Since the effective retrieval-to-LLM cost ratio $c_{\text{ret}}/\bar{c}_{\text{LLM}} \approx 0.02$ in our implementation, a memory utilization of $\mu_\tau = 0.5$ yields a cost ratio of approximately 0.51, consistent with the observed $\sim$50% reduction.

2.6. Hierarchical Control Architecture

The agent's architecture is interpreted as a *hierarchical feedback control system* (Figure 2). The **Manager** module functions as the *high-level controller*: it receives the user's task instruction, interprets the goal, decomposes it into subtasks, and issues commands to the Worker. The **Worker** module functions as the *plant/actuator*: it perceives the current GUI state through screenshot capture and OCR, then executes low-level actions (clicks, keystrokes, scrolling).

The **Memory Graph** serves as the *feedback controller*: before each action, the Manager queries the memory graph to determine whether a known trajectory exists for the current state and objective. The operational cycle thus forms a closed loop:

$$\text{Observe } \sigma_t \xrightarrow{\Phi} n_t \xrightarrow{\pi(\sigma_t, \mathcal{G})} a_t \xrightarrow{\text{Execute}} \sigma_{t+1} \xrightarrow{\text{Update}} \mathcal{G}', \tag{14}$$

where σ_t is the observation at time t, $n_t = \Phi(\sigma_t)$ is the recognized node, $a_t = \pi(\sigma_t, \mathcal{G})$ is the selected action, σ_{t+1} is the resulting observation, and $\mathcal{G}' = \text{Update}(\mathcal{G}, \sigma_t, a_t, \sigma_{t+1})$ is the updated memory graph.

This architecture exhibits the characteristics of *model-reference adaptive control (MRAC)* [11]: the memory graph plays the role of the reference model, encoding desired state-action trajectories, while the LLM-based planner serves as the adaptive mechanism that generates new trajectories when the reference model is insufficient. Over time, the reference model (memory graph) is enriched with newly learned trajectories, progressively reducing the need for adaptive (LLM-based) intervention.

 doi:10.20944/preprints202604.1148.v1

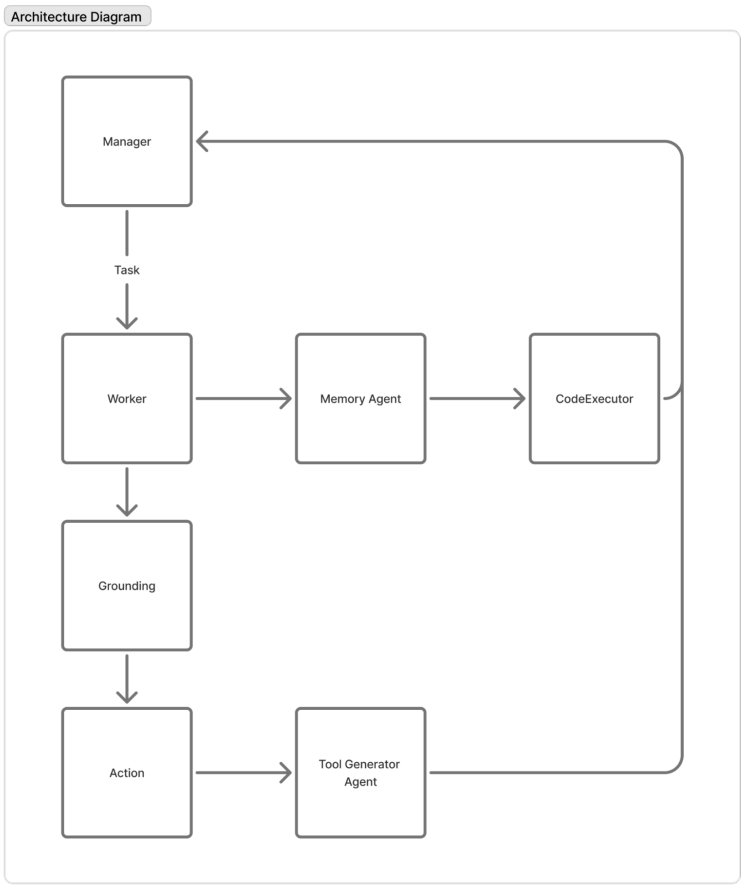

Figure 2. Hierarchical control architecture of the memory-augmented CUA. The Manager (controller) decomposes tasks and queries the Memory Graph (feedback controller). If a known trajectory exists, the Worker (actuator) replays it; otherwise, LLM-based planning generates a new trajectory, which is subsequently stored in the memory graph. The cycle constitutes a closed-loop adaptive control process.

2.7. Decision Policy: Exploitation vs. Exploration

The decision policy $\pi : \Sigma \times \mathcal{G} \to A \cup \{\perp\}$ implements a structured decision-making process that balances exploitation of known solutions against exploration via LLM reasoning [14,15]. The policy operates as follows:

1. **Task Recognition:** The Manager interprets the user instruction and queries the memory graph for matching task descriptors. The search combines textual similarity (instruction vs. stored task names/descriptions) with state matching (current UI context vs. stored node states).
2. **Memory-Driven Execution (Exploitation):** If a matching task descriptor $d \in D$ is found, the Manager retrieves the associated path P_d and the Worker executes the stored action sequence with minimal LLM involvement—only for success verification and minor adaptations.
3. **LLM-Driven Planning (Exploration):** If $\pi(\sigma_t, \mathcal{G}) = \perp$, the standard planning procedure is invoked: the Manager decomposes the task and the Worker uses LLM reasoning at each step. Upon successful completion, the new trajectory is integrated into $\mathcal{G}$.
4. **Memory Update:** After task completion (via either path), the graph is updated: new nodes and edges for newly visited screens and actions, updated usage statistics for memory-served paths, and the Tool Generation module abstracts reusable subsequences into new task descriptors.

This decision framework can be viewed as an instance of *knowledge-based decision support* [16], where the memory graph constitutes a structured knowledge base that reduces the complexity of the decision space from the full LLM planning problem to a graph traversal problem.

 doi:10.20944/preprints202604.1148.v1

2.8. State Recognition and Hashing

The state recognition function $\Phi : \Sigma \to N \cup \{\varnothing\}$ employs a composite matching strategy:

$$\Phi(\sigma) = \arg\min_{n \in N} d_{\text{hash}}(h(\sigma), h(n)) + \lambda \cdot d_{\text{text}}(\text{OCR}(\sigma), \text{OCR}(n)), \tag{15}$$

where $h(\cdot)$ denotes a perceptual hash function applied to the screenshot, d_{hash} is the Hamming distance between hashes, $\text{OCR}(\cdot)$ extracts text content, d_{text} is a text similarity metric, and $\lambda > 0$ is a weighting parameter. If $\min_n[\cdot] > \theta$ for a threshold θ, then $\Phi(\sigma) = \varnothing$, indicating a novel state. The implementation utilizes the Neo4j graph database with vector indexing for efficient semantic search, analogous to techniques used in AppAgentX [3].

2.9. Memory Graph Evolution and Maintenance

The memory graph evolves continuously through the following mechanisms:

- **Node addition:** When $\Phi(\sigma) = \varnothing$, a new node $n_{\text{new}} = (x_\sigma, \varnothing, A_\sigma)$ is created and added to N.
- **Edge addition:** After successfully transitioning from state σ_i to σ_j via action a, the edge (n_i, n_j, a, c) is added to E, where $n_i = \Phi(\sigma_i)$ and $n_j = \Phi(\sigma_j)$.
- **Tool generation:** The Tool Generation module analyzes completed trajectories to identify reusable subsequences. Frequently traversed paths are abstracted into parameterized task descriptors $d \in D$ and attached to the appropriate source node.
- **Pruning:** Periodically, maintenance operations merge duplicate nodes, generalize parameters, and remove low-utility edges to prevent graph bloat and preserve efficient retrieval.

Over the lifetime of the agent, this evolution ensures that $|N|$ and $|E|$ grow monotonically (up to pruning) and that the coverage ratio $\rho(\mathcal{G}, \mathcal{T})$ increases, moving the system toward greater functional stability.

3. Results

3.1. Experimental Setup

We evaluate the proposed memory architecture on the **OSWorld** benchmark [5], a comprehensive suite of over 300 computer use tasks executed in real desktop and web application environments. OSWorld tasks span file management, form completion, web browsing, and multi-step workflows, making it representative of realistic GUI automation scenarios. Each task has a defined initial state (e.g., open applications, existing files) and a target state, and is classified by execution length (15-step or 50-step).

The baseline agent is **S2-Base**: the Agent S2 framework [4] using the Manager–Worker architecture with Claude 4.5 Sonnet as the underlying LLM, but retaining no memory between tasks. Each task is solved from scratch. We compare S2-Base against **S2-Mem**, our memory-augmented variant that adds the graph memory and tool generation modules described in Section 2. Both agents use identical perception components (screenshot capture, OCR) and have access to the same set of low-level actions. The only difference is S2-Mem's ability to store, retrieve, and reuse prior experience.

We evaluate S2-Mem in two configurations:

- **S2-Mem Cold:** Memory is initialized with only a small set of basic tools (e.g., login sequences for common applications), simulating a "cold start" scenario.
- **S2-Mem Warm:** Memory is pre-populated with tools accumulated from prior task executions, simulating an agent that has been operational for some time.

3.2. Evaluation Metrics

We measure three primary metrics tied to the formal model:

1. **Token Consumption** $C(\tau)$: The total number of LLM tokens (input + output) consumed during task execution, as defined in Equation (10).

 doi:10.20944/preprints202604.1148.v1

2. **Execution Time** $T(\tau)$: Wall-clock time from instruction receipt to task completion, encompassing both LLM processing delays and GUI interaction latencies.
3. **Success Rate** $R(\tau) \in \{0, 1\}$: Binary indicator of task completion—1 if the final state matches the target specification, 0 otherwise.

Additionally, we track the number of LLM invocations per task and the average number of steps, which serve as auxiliary measures of computational efficiency.

3.3. Task Success Rates

Table 1 presents the success rates across 369 tasks (50 from the 15-step set and 50 from the 50-step set) for each agent configuration.

Table 1. Comparison of task success rates (%) on OSWorld benchmark. Results are reported for 15-step and 50-step task subsets.

Method	15-Step Tasks	50-Step Tasks
Agent S2 w/ Claude-4.5-Sonnet (S2-Base)	36.9	46.5
S2-Mem w/ Claude-4.5-Sonnet Cold	36.9	46.7
S2-Mem w/ Claude-4.5-Sonnet Warm	36.9	46.9

The results confirm the degradation bound established in Inequality (9): the memory-augmented agent achieves success rates *at least as high as* the baseline in all configurations. The warm-start configuration shows a marginal improvement of 0.4 percentage points on 50-step tasks, attributable to the availability of pre-verified action sequences that reduce planning variance.

3.4. Token Cost and Execution Time

Figure 3 presents the average per-task token cost and execution time across the evaluated configurations.

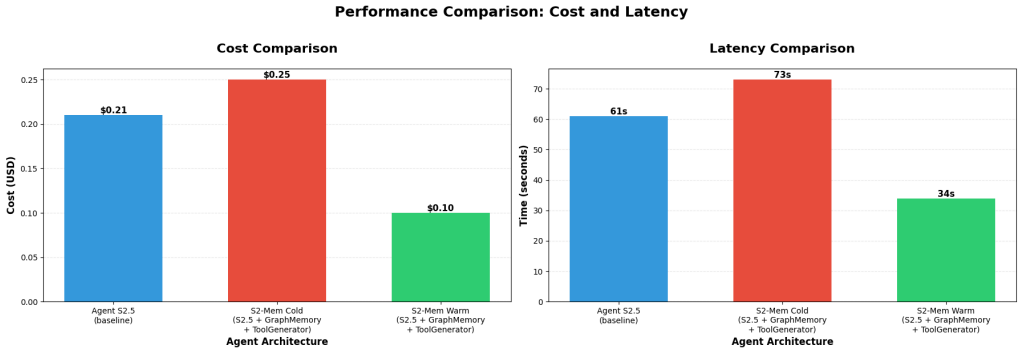

Figure 3. Performance comparison: average per-task token cost (left, in USD) and execution time (right, in seconds) across agent configurations. S2-Mem Warm achieves approximately 52% cost reduction and 44% latency reduction relative to S2-Base.

The baseline agent (S2-Base) incurs an average cost of $0.21 per task with an average execution time of 61 seconds for 15-step tasks. The cold-start memory agent (S2-Mem Cold) shows a moderate increase to $0.25 in cost and 73 seconds in time, reflecting the overhead of memory population during initial encounters. However, the warm-start agent (S2-Mem Warm) achieves $0.10 per task and 34 seconds, representing reductions of 52% and 44%, respectively.

In terms of the formal cost model (Equation 13), the observed cost ratio $C_{\text{mem}}/C_{\text{base}} \approx 0.48$ for the warm-start configuration implies an effective memory utilization rate of:

$$\hat{\mu} \approx \frac{1 - 0.48}{1 - 0.02} \approx 0.53, \tag{16}$$

 doi:10.20944/preprints202604.1148.v1

indicating that approximately 53% of execution steps were served from memory, consistent with independent counts of LLM invocations (22 per task for S2-Mem vs. 45 for S2-Base on average).

3.5. LLM Invocation Analysis

The reduction in LLM calls provides further evidence of the memory graph's stabilizing effect. For the warm-start configuration, the agent required only 15 LLM calls for a task that demanded 38 calls under the baseline—a 60% reduction driven by memory-based replay of the login and navigation sequences. The overall average across all tasks dropped from 45 invocations (S2-Base) to 22 invocations (S2-Mem Warm), with individual tasks showing reductions up to 69% for highly repetitive workflows.

4. Discussion

4.1. Interpretation Through the Stability Framework

The experimental results provide empirical support for the stability-theoretic framing developed in Section 2.3. The memory-augmented agent exhibits the three key properties of a functionally stable system [6,21]:

1. **Monotonic coverage growth:** As the agent completes more tasks, the memory graph accumulates nodes and edges, increasing the coverage ratio $\rho(\mathcal{G}, \mathcal{T})$. The transition from cold-start to warm-start performance (Table 1, Figure 3) illustrates this growth: cost decreases from \$0.25 to \$0.10 as memory populates.
2. **Convergent efficiency:** The memory utilization ratio μ_k (Definition 3) increases over the evaluation period. Initial tasks require extensive LLM planning ($\mu \approx 0$), while later tasks with overlapping subtasks benefit from stored trajectories ($\mu \approx 0.53$ on average).
3. **Graceful degradation:** When the agent encounters novel states not represented in the memory graph ($\Phi(\sigma) = \varnothing$), it falls back to LLM-based planning without any performance penalty relative to the baseline, confirming Inequality (9).

These properties mirror the resilience criteria established for communication network topologies [7,9], adapted to the domain of autonomous GUI agents.

4.2. Control-Theoretic Interpretation

The closed-loop architecture (Equation 14) demonstrates characteristics of adaptive control [11,13]. The initial cold-start phase corresponds to the *transient response* of an adaptive controller before the reference model has been adequately identified. During this phase, performance is comparable to the open-loop (baseline) system. As the memory graph fills, the system enters the *steady-state regime*, where the reference model (memory graph) provides accurate state-action mappings and the adaptive mechanism (LLM planner) is invoked only for novel situations.

The convergence behavior of the memory utilization ratio μ_k (Definition 3) is analogous to parameter convergence in MRAC systems [12]: both exhibit initial transients followed by asymptotic stability, bounded by the irreducible novelty in the input distribution.

An important aspect of the control architecture is the *separation of concerns* between the Manager (high-level controller) and Worker (low-level actuator). This mirrors the well-known cascade control structure [13], where the outer loop operates at a slower time scale (task decomposition) and the inner loop at a faster time scale (individual GUI actions). The memory graph enhances the outer loop's decision quality by reducing the search space from the exponential space of possible LLM plans to the polynomial space of graph traversals.

4.3. Decision-Making Perspective

From the perspective of decision-making theory [15,16], the memory graph functions as a *knowledge-based decision support system (DSS)* that transforms the agent's operational decisions. Without memory, every step requires solving a full planning problem—essentially, the agent faces a *Markov decision process (MDP)* where each state requires policy computation from scratch. With memory, the

 doi:10.20944/preprints202604.1148.v1

agent's decision reduces to a binary classification: "Is this state known?" followed by either efficient lookup or standard planning. This reduction in decision complexity directly manifests as the observed 50% reduction in computational cost.

The balance between exploitation (memory retrieval) and exploration (LLM planning) in the policy π can be analyzed through the *multi-armed bandit* framework [14]. The memory graph effectively eliminates the exploration cost for previously solved subproblems, allowing the agent to allocate its computational budget toward genuinely novel challenges. This is especially useful when the task stream contains a mixture of recurring and previously unseen subtasks.

4.4. Limitations and Perturbation Sensitivity

Several limitations merit discussion in terms of robustness and stability guarantees:

State recognition fragility. The state recognition function Φ (Equation 15) relies on perceptual hashing and text matching, which are sensitive to visual perturbations. Even minor interface changes (theme changes, resolution scaling, or application updates) can cause $\Phi(\sigma) = \varnothing$, preventing memory utilization. This constitutes a practical stability limitation of the recognition layer: small observation changes can disable memory retrieval even when the underlying task remains the same. More robust methods such as structural analysis through models like OmniParser [3] could improve Φ's accuracy.

Scalability. As $|N|$ and $|E|$ grow to hundreds of nodes and thousands of edges, graph traversal and search operations may become more costly. We partially address this through vector search indexing and Neo4j's graph query optimization, but a comprehensive analysis of asymptotic scaling remains necessary.

Generalization boundary. The memory graph is bound to specific application environments. A login sequence learned for one website cannot be directly transferred to another without abstract concept matching. Multi-level semantic memory organization [17–19]—where abstract concepts ("authenticate," "export document") exist at higher levels with environment-specific implementations below—represents a promising direction.

Cold-start overhead. The cold-start configuration shows slightly higher cost than the baseline (Figure 3), reflecting the overhead of memory population. This transient cost is the "price of learning" inherent in any adaptive system [11] and is amortized over subsequent task executions.

4.5. Quality of Generated Tools

The Tool Generation module successfully abstracts many generalized instruments. Tools such as `Login(app, user)`, `OpenFile(filename)`, and `SendEmail(recipient)` proved versatile across application contexts. In some cases, the generator produced overly specific tools (e.g., `ExportPDFReport`, bound to a specific format sequence); such cases highlight the need for action parameterization and composition of similar tools [20]. Nevertheless, the availability of reusable modules enabled the agent to effectively solve complex multi-step tasks through composition. For instance, a 50-step task ("Take a dataset from a spreadsheet, build a diagram, and insert it into a presentation") was solved by composing three previously learned tools with minimal new planning, resulting in approximately 60% fewer LLM tokens compared to the baseline agent's full planning approach.

5. Conclusions

This paper presented a graph-based persistent memory architecture for LLM-driven computer use agents and provided its formal analysis through the frameworks of functional stability theory, hierarchical control theory, and decision-making theory. The core scientific contribution remains the directed graph memory system—where nodes represent GUI states and edges encode executable action sequences—integrated into the Agent S2 framework and evaluated on the OSWorld benchmark.

The key findings are as follows. First, the memory-augmented agent achieves an approximately 50% reduction in token consumption and a substantial reduction in execution time relative to the memoryless baseline, while maintaining equivalent or slightly improved task success rates. Second, the formal analysis establishes that the memory graph provides a *stability margin*: the agent can gracefully

 doi:10.20944/preprints202604.1148.v1

degrade to baseline behavior when encountering novel states, ensuring that memory augmentation never reduces performance. Third, the convergent growth of the memory coverage ratio $\rho(\mathcal{G}, \mathcal{T})$ and utilization ratio μ_k confirms that the system exhibits the asymptotic stability properties characteristic of well-designed adaptive control systems.

The contributions extend beyond the immediate application domain. By formalizing the memory-augmented agent as a tuple $\mathcal{S} = \langle A, \Sigma, \mathcal{G}, \delta, \pi, \Phi \rangle$ and defining quantitative stability and convergence criteria, we provide a mathematical framework applicable to the broader class of experience-driven autonomous systems. The analogy between memory graph connectivity and classical functional stability conditions [6,7] suggests that tools from graph theory and network reliability analysis can be profitably applied to the design and verification of agent memory systems.

Future work will pursue several directions aligned with the stability and control perspective developed here. First, we aim to provide *formal convergence guarantees* for the memory utilization ratio under specified task distribution assumptions, analogous to convergence proofs in adaptive control theory [12]. Second, we will investigate *multi-agent shared memory* as a form of distributed control, where multiple CUA instances contribute to and benefit from a common memory graph—raising questions of consistency, conflict resolution, and distributed stability analogous to those studied for virtual network reconfiguration [9]. Third, *formal verification of memory stability*—ensuring that graph maintenance operations (pruning, merging) preserve the functional stability of the stored knowledge—is an important direction for reliability-critical deployments [21]. Finally, the development of more robust state recognition methods, potentially leveraging structural analysis models, will improve perturbation resistance and extend the domain of stable operation.

Author Contributions: Conceptualization, A.M. and D.V.; methodology, A.M.; software, D.V.; validation, A.M. and D.V.; formal analysis, A.M.; investigation, D.V.; writing—original draft preparation, A.M. and D.V.; writing—review and editing, A.M.; visualization, D.V.; supervision, A.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CUA	Computer Use Agent
GUI	Graphical User Interface
LLM	Large Language Model
MRAC	Model-Reference Adaptive Control
MDP	Markov Decision Process
DSS	Decision Support System
OCR	Optical Character Recognition

References

1. Sager, P.J.; Meyer, B.; Yan, P.; von Wartburg-Kottler, R.; Etaiwi, L.; Enayati, A.; Nobel, G.; Abdulkadir, A.; Grewe, B.F.; Stadelmann, T. A Comprehensive Survey of Agents for Computer Use: Foundations, Challenges, and Future Directions. *arXiv preprint* arXiv:2501.16150, **2025**.
2. Lee, S.; Choi, J.; Lee, J.; Wasi, M.H.; Choi, H.; Ko, S.Y.; Oh, S.; Shin, I. Explore, Select, Derive, and Recall: Augmenting LLM with Human-like Memory for Mobile Task Automation. *arXiv preprint* arXiv:2312.03003, **2023**.
3. Jiang, W.; Zhuang, Y.; Song, C.; Yang, X.; Zhou, J.T.; Zhang, C. AppAgentX: Evolving GUI Agents as Proficient Smartphone Users. *arXiv preprint* arXiv:2503.02268, **2025**.

 doi:10.20944/preprints202604.1148.v1

4. Agashe, S.; Wong, K.; Tu, V.; Yang, J.; Li, A.; Wang, X.E. Agent S2: A Compositional Generalist-Specialist Framework for Computer Use Agents. *arXiv preprint* arXiv:2504.00906, **2025**.
5. Xie, T.; Zhang, D.; Chen, J.; Li, X.; Zhao, S.; Cao, R.; Hua, T.J.; Cheng, Z.; Shin, D.; Lei, F.; Liu, Y.; Xu, Y.; Zhou, S.; Savarese, S.; Xiong, C.; Zhong, V.; Yu, T. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. *arXiv preprint* arXiv:2404.07972, **2024**.
6. Barabash, O.V. *Construction of Functionally Stable Distributed Information Systems*; NAOU: Kyiv, Ukraine, **2004**; 226p.
7. Barabash, O.; Makarchuk, A.; Open'ko, P.; Korotin, S. Application of SVM, FFNNs, k-NN and Their Ensembles for Identifying Functionally Reliable Systems. *Axioms* **2025**, *14*, 237.
8. Kravchenko, Y.V.; Nikiforov, S.V. Definition of the problems of the theory of functional stability in relation to application in computer systems. *Telecommun. Inf. Technol.* **2014**, *1*, 12–18.
9. Zamrii, I.; Vyshnivskyi, V.; Sobchuk, V. Method of Ensuring the Functional Stability of the Information System Based on Detection of Intrusions and Reconfiguration of Virtual Networks. *CEUR Workshop Proc.* **2024**, *3654*, 252–264.
10. Bellini, E.; Coconea, L.; Nesi, P. A Functional Resonance Analysis Method Driven Resilience Quantification for Socio-Technical Systems. *IEEE Syst. J.* **2020**, *14*, 1234–1244.
11. Ioannou, P.A.; Sun, J. *Robust Adaptive Control*; Dover Publications: Mineola, NY, USA, **2006**.
12. Narendra, K.S.; Annaswamy, A.M. *Stable Adaptive Systems*; Dover Publications: Mineola, NY, USA, **2005**.
13. Åström, K.J.; Murray, R.M. *Feedback Systems: An Introduction for Scientists and Engineers*; Princeton University Press: Princeton, NJ, USA, **2008**.
14. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, **2018**.
15. Bellman, R. *Dynamic Programming*; Princeton University Press: Princeton, NJ, USA, **1957**.
16. Turban, E.; Sharda, R.; Delen, D. *Decision Support and Business Intelligence Systems*, 9th ed.; Prentice Hall: Upper Saddle River, NJ, USA, **2011**.
17. Zhang, Z.; Bo, X.; Ma, C.; Li, R.; Chen, X.; Dai, Q.; Zhu, J.; Dong, Z.; Wen, J.-R. A Survey on the Memory Mechanism of Large Language Model Based Agents. *arXiv preprint* arXiv:2404.13501, **2024**.
18. Packer, C.; Wooders, S.; Lin, K.; Fang, V.; Patil, S.G.; Stoica, I.; Gonzalez, J.E. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint* arXiv:2310.08560, **2023**.
19. Xu, W.; Liang, Z.; Mei, K.; Gao, H.; Tan, J.; Zhang, Y. A-Mem: Agentic Memory for LLM Agents. *arXiv preprint* arXiv:2502.12110, **2025**.
20. Pichkur, V.; Sobchuk, V.; Cherniy, D. Mathematical Models and Control of Functionally Stable Technological Process. In *Computational Methods and Mathematical Modeling in Cyberphysics and Engineering Applications*; Wiley: Hoboken, NJ, USA, **2024**; Volume 1, pp. 101–119.
21. Pichkur, V.; Sobchuk, V.; Cherniy, D.; Ryzhov, A. Functional Stability of Production Processes as Control Problem of Discrete Systems with Change of State Vector Dimension. *Bull. Taras Shevchenko Natl. Univ. Kyiv. Phys. Math.* **2024**, *1*, 105–110.
22. Musienko, A.; Vorvul, D. Analysis of the Efficiency and Comparison of Retrieval-Augmented Generation Systems in Mergers and Acquisitions. In *Lecture Notes in Networks and Systems*; Springer: Cham, Switzerland, **2025**. https://doi.org/10.1007/978-3-031-89296-7_27.