GenericAgent: A Token-Efficient Self-Evolving LLM Agent via Contextual Information Density Maximization (V1.0)

Advantage AI Agent Lab (A3 Lab) *

See Author Contributions (Section 8) for a full author list.

Abstract

Long-horizon large language model (LLM) agents are fundamentally limited by context. As interactions become longer, tool descriptions, retrieved memories, and raw environmental feedback accumulate and push out the information needed for decision-making. At the same time, useful experience gained from tasks is often lost across episodes. We argue that long-horizon performance is determined not by context length, but by how much *decision-relevant* information is maintained within a finite context budget. We present **GenericAgent (GA)**, a general-purpose, **self-evolving** LLM agent system built around a single principle: **context information density maximization**. GA implements this through four closely connected components: a minimal atomic tool set that keeps the interface simple, a hierarchical on-demand memory that only shows a small high-level view by default, a self-evolution mechanism that turns verified past trajectories into reusable SOPs and executable code, and a context truncation and compression layer that maintains information density during long executions. Across task completion, tool use efficiency, memory effectiveness, self-evolution, and web browsing, GA consistently outperforms leading agent systems while using significantly fewer tokens and interactions, and it continues to improve over time rather than simply persisting across tasks.

Project: https://github.com/lsdefine/GenericAgent

1 Introduction

The recent emergence of agentic systems such as Claude Code [1], OpenAI Codex [2], and OpenClaw [3] marks a qualitative shift in what is expected of a Large Language Model (LLM). Rather than serving solely as passive text generators, LLMs are increasingly deployed as goal-directed agents that operate through terminals, file systems, browsers, and external tools. This transition substantially expands their functional scope, but also introduces concrete systems-level challenges in context management [4, 5] and experience accumulation [6–9]. In particular, next-generation agents are required to continuously receive and solve tasks within a persistent environment, accumulating cross-task state and experience over time. Success therefore depends jointly on the reasoning capability of the underlying model and the system's ability to retain and reuse that experience effectively. Together, these requirements give rise to two fundamental challenges.

The first challenge is **context explosion.** As the agent interacts with the environment, the prompt grows continuously as tool definitions, retrieved memories, intermediate observations, and raw environmental feedback accumulate across steps [4, 5]. This growth is not merely a token-consumption concern, it directly undermines reasoning quality. LLMs have finite effective attention [10, 11]. As irrelevant or outdated content occupies an increasing share of the context window, the model's ability to attend to decision-relevant information

*This is a joint research lab with members from Shenzhen Aquaintelling Technology and Fudan University

degrades. Critical constraints are overlooked; intermediate states are confused with earlier ones; hallucinated facts emerge and compound through subsequent steps [10, 12, 13]. The core tension is that multi-step execution inherently requires accumulating context, yet the accumulated context itself becomes the primary source of failure. Without well-designed context management mechanisms, longer interaction does not yield better-informed decisions.

The second challenge is **effective experience accumulation and reuse**. In long-horizon environments, critical knowledge, such as user preference, tool behaviors and effective action patterns, is not available at the outset. It emerges only through repeated trial and failure during actual task execution. This exploration is a natural and necessary process. The key question is whether the lessons learned can be retained and reused when similar tasks arise later. Without such a mechanism, agents repeat the same failure patterns across sessions [9, 14]. Successful strategies, once discovered, are forgotten upon context expiration. Token expenditure scales linearly with task count, yet effective capability remains flat—a stagnation loop with no return on accumulated interaction.

Existing agent frameworks largely fail to address this. Most treat each task episode as stateless, with no persistent memory across sessions [9, 15]. Even when retrieval-augmented memory is introduced, it typically stores raw logs rather than distilled, reusable operational knowledge [16, 17]. More critically, there is no feedback-driven refinement. Stale or incorrect memories are never updated, leading to silent degradation instead of improvement [18, 19]. These gaps prevent current agents from achieving self-evolution.

We propose **GenericAgent (GA)**, a self-evolving LLM agent system built around a fundamental principle: **maximizing contextual information density**. The core view is that agent performance is not determined by context length alone, but by how much decision-relevant information is introduced, maintained, and updated within the limited context budget.

GA realizes this principle through four mechanisms that operate across the lifecycle of contextual information. (1) A minimal atomic tool set reduces persistent tool overhead, preventing low-value interface information from occupying the context before task execution. (2) A hierarchical memory mechanism selectively retains only verified and task-relevant knowledge, shaping what information persists over time. (3) Self-evolution pipeline compresses interaction trajectories into reusable Standard Operating Procedures (SOPs), code, and skills, progressively transforming experience into compact and structured capability. (4) A context truncation and compression mechanism actively manages historical information when the context exceeds its budget, ensuring the active context remains concise and task-relevant through layered truncation and compression. Together, these mechanisms continuously reshape the context from a passive accumulation of information into a high-density, decision-centric representation.

We conduct comprehensive experiments on multiple benchmarks against representative agent systems, demonstrating that GA consistently achieves a superior efficiency, performance trade-off, reaching higher task completion at substantially lower token cost, with this advantage remaining stable across repeated runs.

The report is structured as follows. Section 2 introduces the core design philosophy of GA and describes its key components, including the minimal atomic tool set, hierarchical memory architecture, self-evolution mechanism, and context truncation and compression. Building on these foundations, we present several emergent capabilities arising from their interaction, such as subagent dispatch, watchdog mechanisms, scheduled execution, and autonomous idle behavior. We then further examine higher-level capabilities enabled by GA's minimalist architecture, including compositional capabilities and autonomous exploration. Section 4 provides a comparative evaluation against widely used agent frameworks on real-world tasks, focusing on both task success rate and token efficiency. The results highlight consistent advantages of GA in memory utilization, the self-evolution loop, and browser interaction design. Moreover, Section 5 summarizes key findings for future agent design and reviews related work for each major component. We finally summarizes key findings for future agent design and reviews related work for each major component in Section 6.

2 GenericAgent

2.1 Design Principles

Context information density is all a self-evolving LLM agent needs.

The performance of LLM-based agents is determined not merely by context length, but more fundamentally by context quality. Recent studies have identified three compounding failure modes that limit how effectively models can use their input. First, models exhibit pronounced positional bias when processing long sequences: relevant information placed in the middle of the context is significantly harder to retrieve than information near the beginning or end [20]. Second, irrelevant content does not simply remain unused; it can actively degrade performance by diverting attention away from decision-critical evidence [21]. Third, the effective context length of LLMs falls far short of their nominal window size, meaning that part of the provided context is functionally inaccessible during generation [22].

These three phenomena reinforce one another in practice. As context grows longer, positional bias makes middle-positioned evidence harder to retrieve, irrelevant content competes for the model's limited effective attention, and the declining ratio of effective to nominal context length means that an increasing fraction of the prompt is simply wasted. The result is that, beyond a certain point, adding more context does not improve performance and may instead reduce it.

This has direct implications for agent system design. Existing agent frameworks that rely on monolithic prompt assembly often devote a disproportionate share of the effective context budget to scaffolding, control text, and marginally relevant interaction history. Rather than exposing only decision-relevant information, such strategies displace the evidence most needed for the current step, leaving the agent with more context in volume but less context in utility, while still increasing inference cost.

From the perspective of context engineering, this problem is best understood in terms of two primary requirements and one secondary representational constraint:

- **Completeness.** All information required for the current decision must be explicitly present in the context, preventing the model from relying on implicit assumptions or hallucinated inferences.
- **Conciseness.** Irrelevant and redundant information must be eliminated so that attention remains focused on decision-critical signals.
- **Naturalness.** The context should remain reasonably natural and semantically legible, especially when aggressive compression or artificial encodings make the representation harder for the model to use. This is an important but secondary constraint: in most agent settings, the dominant failure mode is not lack of natural phrasing itself, but the loss of completeness or conciseness.

We argue that completeness and conciseness define the primary design space of context quality in LLM-based agents. Completeness ensures that the model has the information it needs, while conciseness ensures that this information is not diluted by distracting or low-value content. Naturalness still matters, but mainly as a constraint on how information is represented rather than as the main bottleneck in most agent failures. It helps avoid brittle encoding, overly telegraphic summaries, and artificial formats that the model may parse less reliably. Other commonly discussed dimensions can largely be understood within this framework. For example, recency contributes to both completeness and conciseness: the latest state is often essential for the current decision, while outdated information is typically no longer useful. Relevance, rather than being an independent axis, is better understood as a cross-cutting concern: both completeness and conciseness presuppose a judgment of what matters, the former in deciding what must be included and the latter in deciding what should be removed.

Critically, the main structural tension is between completeness and conciseness. This is not merely a consequence of finite context budgets. Even if the context window were unbounded, a structural tension would remain. The tension is structural rather than merely budgetary for at least three reasons (as shown in Figure 1):

- Including more potentially relevant information improves completeness but weakens conciseness.
- Summarization and compression improve conciseness but risk omitting details needed for completeness.

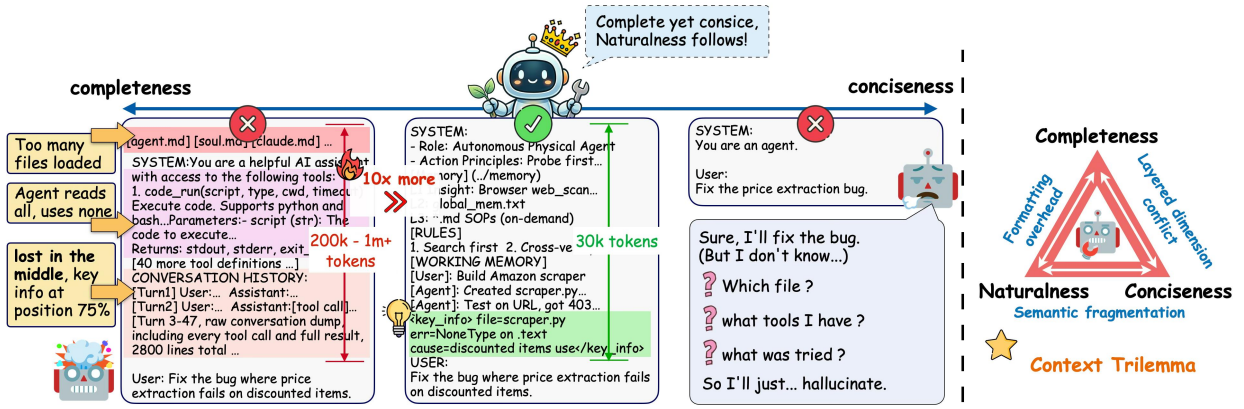

Figure 1 Completeness and conciseness define the core trade-off in context design, while naturalness acts as a constraint on valid representations. The left example is complete but overly verbose, obscuring key information. The right example is concise but incomplete, missing critical context. In GA, the context balances both, yielding a natural and effective representation.

- Naturalness can sharpen this trade-off in some cases, because highly compressed or artificial representations may be harder for the model to interpret, but this is typically a secondary effect rather than the dominant constraint.

The finite effective context window [22] sharpens this tension into a hard practical constraint, but the underlying conflict is structural rather than merely resource-driven. In this sense, context engineering is best viewed not as an equal three-way trilemma, but as a constrained optimization problem centered on balancing completeness and conciseness, with naturalness acting as a supporting constraint on representation.

This principle serves as the guiding idea behind the design of GA. Rather than treating context as a passive byproduct of interaction history, GA systematically optimizes for information density at multiple layers of the system. Its memory hierarchy keeps a lightweight orientation layer visible while leaving richer facts and procedures to explicit reading, preserving conciseness without giving up access to deeper knowledge. Its browser interaction layer represents web pages through semantically structured observations rather than raw HTML, which improves readability while avoiding large amounts of low-value markup in context. Its self-evolution mechanism provides a way to consolidate successful experience into reusable memory and procedures, helping later tasks begin from more compact and task-relevant context. The following subsections describe how each mechanism instantiates this principle in detail.

2.1.1 Tool Minimality

To maximize context information density before task execution begins, GA constrains its tool design to the minimal necessary set. Tool proliferation introduces system-level costs at two levels:

- **At the prompt level**, each additional tool enlarges the tool schema and associated instructions injected into the context; in multi-turn interaction, this overhead accumulates and occupies effective context budget that should be reserved for task-relevant information.
- **At the policy level**, each additional tool enlarges the action space and increases ambiguity in tool selection. This makes planning more brittle, weakens the stability of tool-use patterns, and increases the likelihood of execution errors and unnecessary retries.

Tool minimality is therefore not merely an austerity choice; it is a better operating point that reduces both prompt overhead and decision complexity.

In practice, tool selection must satisfy two conditions: **atomicity**, which constrains each tool to an irreducible

primitive capability, and **compositional generalization**, which allows complex behaviors to be realized through sequences of such primitives. GA obtains capability through composition rather than tool enumeration: a small set of atomic tools are designed as reusable primitives rather than task-specific interfaces, and complex behaviors emerge from their combination. Retained operational knowledge is therefore naturally expressed as reusable ways of sequencing and combining a small set of general primitives, rather than as new task-specific interfaces, making experience consolidation more lightweight. Tool minimality is thus not a restriction, but the core mechanism by which GA preserves general-purpose capability while reducing interaction overhead and creating favorable conditions for subsequent experience consolidation.

2.1.2 Hierarchical Memory

To continuously maintain context information density during task execution, GA adopts a systematic approach to memory organization. For a general-purpose agent, the main memory challenge lies less in storage itself than in controlling how much information remains in the active context. In conventional agent frameworks, prior interactions, intermediate states, and execution traces accumulate as the task unfolds, progressively consuming the context budget and burying decision-critical information beneath low-value content, thereby degrading the quality of reasoning at each step.

GA's core idea is to avoid treating all retained information as prompt-resident by default: only a small always-on layer remains visible, while deeper memory is accessed only when needed. This on-demand access principle naturally leads to a layered memory organization. The always-on layer is intentionally small, containing only lightweight orientation information, whereas richer factual knowledge, procedural knowledge, and historical interaction data are stored in deeper layers in archived or compressed form. These deeper memories enter the active context through on-demand retrieval rather than default injection. As a result, memory does not steadily displace the active-context budget required for the task at hand, preserving more of the context window for current reasoning and action.

The always-on layer can stay minimal because it needs to encode only the *existence* of each knowledge category: the LLM itself serves as both compressor and decoder—when new knowledge is consolidated, it produces a near-minimal natural-language summary of the *existence* under a word-limited requirement prompt, and at retrieval time it can follow any such compressed pointer to the relevant deeper layer through tool calls.

2.1.3 Self-Evolution as Experience Consolidation

To enable context information density to improve continuously across tasks over the long term, GA introduces a self-evolution mechanism. If hierarchical memory controls how past information is retained and accessed within a single task, self-evolution concerns whether past execution can remain useful beyond the episode in which it occurred. In real long-horizon environments, much valuable knowledge is not available at the outset, but emerges gradually through repeated interaction with a specific user's file system, workflows, and external services. Without mechanisms for consolidating such experience, agents cannot benefit from prior interactions, and must rediscover solutions from scratch on every similar task, incurring substantial and avoidable efficiency costs. **GA is designed to address this challenge by enabling experience consolidation across tasks, rather than treating each task as a fully isolated interaction.**

Constrained by the context-quality trilemma discussed above, without any mechanism for retaining validated experience, the agent often faces a recurring dilemma on repeated tasks: either replay a longer exploratory process, which weakens conciseness, or act from a shorter but underspecified prompt, which weakens completeness. GA offers a way out of this trade-off. Rather than directly recording optimal action sequences, it provides a stronger starting point for future tasks, progressively improving reuse performance over repeated executions. GA does not modify the base model directly; instead, it evolves by recording and updating the informational environment in which the model operates. Accordingly, memory management centers on selective consolidation, favoring information grounded in successful execution while filtering out volatile, weakly verified, or purely situational content.

This selective view also clarifies the relationship between self-evolution and the two mechanisms discussed above. Retained operational knowledge is naturally expressed as reusable ways of sequencing and combining

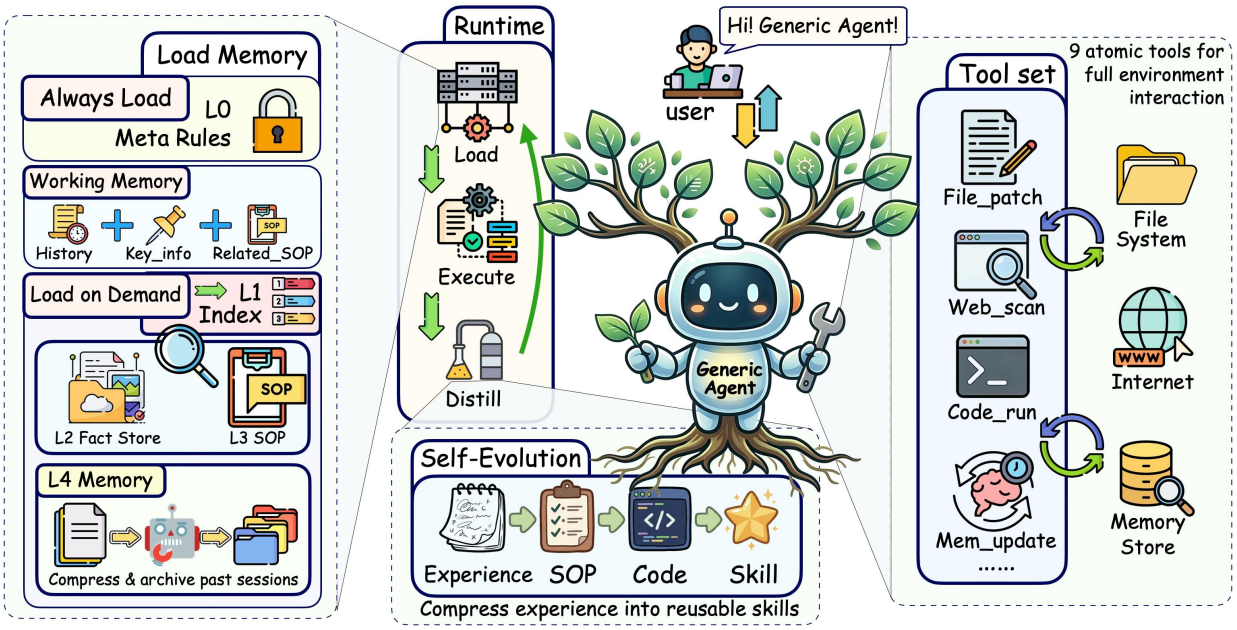

Figure 2 Overall framework of GA. GA follows a unified agent loop that constructs an execution context from the current task and relevant memory, generates outputs or tool calls, and updates the system through structured feedback. It is built on four core mechanisms: a minimal tool set, hierarchical memory, reflection-driven self-evolution, and structured browser extraction.

a small set of general primitives rather than new task-specific interfaces, making experience consolidation more compact and lightweight, which is precisely where tool minimality creates favorable conditions for self-evolution. At the same time, consolidation does not reduce every past task to a single uniform reusable object; instead, drawing on the layered memory structure, the effective parts of past execution are preserved in forms that are smaller, more targeted, and potentially more reusable than the original trajectories. This is where hierarchical memory provides the organizational foundation for self-evolution.

2.1.4 Context Truncation and Compression

Many agent frameworks rely on extended context windows of up to 1M tokens, assuming that more context yields better reasoning. In practice, it is very expensive, but introduces more hallucinations. We refer to the effective ceiling as the *hallucination-free context length*, which for current models is roughly an order of magnitude smaller. GA therefore targets a compact budget of under 30k tokens and invests in compression rather than expansion—packing higher-density information into a smaller window outperforms feeding diluted content into a larger one.

To actively maintain context information density during long-horizon execution, GA introduces a context truncation and compression mechanism. As the conversation progresses, accumulated context eventually exceeds the model's effective processing capacity; without active management, low-value historical content increasingly occupies the context budget needed for ongoing reasoning. Instead of permitting unbounded context growth, GA organizes historical information through layered management at multiple granularities. Tool outputs are first truncated with a head-tail policy to bound the size of individual messages. Redundant content in older messages is then compressed at the tag level to remove low-value fragments. When the global context budget is exceeded, the oldest messages are evicted in chronological order. Meanwhile, a continuously injected working-memory anchor prompt preserves task-critical information in the active context. Collectively, these mechanisms keep the active context compact and decision-relevant throughout long-horizon execution, preventing it from growing linearly with the number of interaction turns.

2.2 Overview of GenericAgent

Guided by the principles above, we develop GenericAgent, a general-purpose self-evolving LLM agent system designed to automate tasks within a local computing environment. It enables a compatible LLM backend to operate across browsers, terminals, file systems, keyboard and mouse inputs, screen-based perception, and mobile devices through configurable tool adapters. The system is model-agnostic: the underlying reasoning engine (e.g., Claude, GPT, Gemini) can be replaced without affecting execution logic, tool interfaces, or memory architecture. The overall architecture is illustrated in Figure 2.

GA is organized around a unified agent loop that connects the model with the environment. At each step, the system combines global memory with the current task specification to form an execution context. The LLM processes this context and produces either direct outputs or tool calls. Tool execution is delegated to external modules, and the results are returned as structured signals that update the system state. This loop creates a continuous interaction between reasoning and environmental feedback, enabling tool-grounded iterative execution. After task completion, execution traces are compressed into structured long-term representations and stored in shared memory. This allows accumulated experience to improve future execution through compression and reuse rather than relying on unbounded computation.

On top of this loop, GA supports two execution modes. **Interact mode** handles user-initiated tasks such as code execution, information retrieval, and file operations. **Reflect mode** requires no user instruction; instead, it continuously monitors for environmental changes and automatically triggers the corresponding task when a specific condition or event is detected, analogous to the watchdog pattern described later. Both modes share the same agent loop and memory system, differing only in how they are triggered and how outputs are handled. To support long-horizon operation, GA defines explicit execution bounds that ensure controllability while preserving interruptibility and scope constraints.

2.3 Core Components of GenericAgent

Building on the principles established in Section 2.1, we describe how GA realizes each of them through four concrete components: the atomic tool set, hierarchical memory management, self-evolution, and context truncation and compression.

2.3.1 Minimal Atomic Toolset

GA's toolset adheres to a minimalist design: a small set of atomic primitives that can be composed to solve a wide range of tasks. This section describes which tools are provided, how they are defined and invoked, how their boundaries are enforced, and how they are optimized for efficient decision-making.

Composition and design rationale. GA exposes nine atomic tools, organized into five capability classes. The **File Operations** class includes `file_read`, `file_patch`, and `file_write` for eading, precise editing, and block writing. The **Code Execution** class contains `code_run`, which executes Python or Bash within a controlled runtime. The **Web Interaction** class includes `web_scan` and `web_execute_js` for low-cost page inspection and precise browser actions. The **Memory Management** class includes `update_working_checkpoint` and `start_long_term_update` for short-term context maintenance and long-term memory distillation. The **Human-in-the-loop** class is `ask_user`, used when user decisions are required. Each tool maintains a single responsibility, and these tools form a complete loop covering state observation, action execution, context preservation, and intervention requests. This tool set directly instantiates the atomicity and compositional generalization principles discussed in Section 2.1.1. Each tool is restricted to a single, indivisible capability with no functional overlap. Consequently, complex tasks are executed by combining these basic primitives, rather than by introducing new, specialized interfaces.

Declaration, invocation, and dispatch. GA represents each tool as a verifiable schema contract and routes execution through a unified dispatcher. Each tool defines a name, a description, and a set of parameters described by JSON Schema, all under a consistent type function format. At runtime, this schema is injected into the interface, enabling GA to produce structured `tool_use` calls with explicit names and arguments. The dispatcher maps each call to a local executor and manages pre- and post-execution handling together with the returned results, ensuring a clear separation between declaration, invocation, and execution.

Capability boundaries and execution discipline. GA enforces a clear permission hierarchy via the injected toolset. Each tool's capability scope is strictly bounded and non-overlapping, avoiding the ambiguity and redundancy associated with overlapping functionality. With read-only tools, GA can inspect and reason about state but cannot modify it; with patch tools, GA can edit text, subject to strict matching and parameter constraints; with execution and web-control tools, GA can trigger real system actions. All interactions return structured outputs through the dispatcher, which improves controllability, reduces operational risk, and ensures full traceability of every action.

Tool specific optimization for execution efficiency. GA further optimizes each atomic tool for execution efficiency and token economy. `file_read` supports segmented reads (`start`/`count`), keyword anchoring, and line-number output, so GA can inspect only the relevant regions. `file_patch` enforces unique `old_content` matching and fails fast on zero or multiple matches, reducing the risk of silent multi-location edits. `web_scan` incorporates a layout-analysis algorithm that clones the live DOM, computes per-element visibility, classifies regions as main or non-essential through overlay and partition analysis, and removes covered or hidden elements before serialization, reducing token cost by an order of magnitude compared to raw DOM output. `web_execute_js` returns action results plus page-change observations, so many workflows proceed without another full scan. `code_run` is restricted to one invocation per turn, ensuring each result is observed before the next action is planned. The goal of these optimizations extends beyond merely compressing individual tool outputs. By providing more precise and complete feedback at each step, they reduce redundant probing and retries in subsequent turns. Ultimately, this lowers overall token consumption across multi-turn interactions and maximizes context information density.

Theoretically, the agent could accomplish any task using only the **code_run** tool. By executing custom scripts, it has the ability to replicate the exact functions of every other tool. Therefore, the remaining eight tools are not designed to expand the agent's capabilities. Instead, they serve as specialized shortcuts to reduce the agent's decision-making cost and operational overhead. Rather than forcing the agent to write code from scratch for every minor action, this carefully designed toolset acts as a **harness**, keeping the agent focused on efficient problem-solving.

2.3.2 Hierarchical Memory Architecture

To maximize context information density during task execution, GA organizes memory into three functionally distinct types rather than a single undifferentiated store. **Working memory** is continuously injected across turns, carrying only the minimal necessary task state: current objectives, constraints, and progress. This allows the agent to maintain execution continuity without reconstructing the full context each time. **Always-on memory** remains persistently visible but is deliberately compressed to the lightest possible orientation information. It provides only the basic memory layout and navigation index, rather than loading complete knowledge into the context by default. **Long-term memory** is kept outside the default context and managed via an explicit post-task consolidation process; only verified and stable knowledge is written back, ensuring that historical accumulation does not automatically translate into continuous context growth. This division directly embodies the principle of context information density maximization: each type of memory appears in the active context only at the necessary moment and at the necessary level of granularity.

To realize this design at the implementation level, GA defines a four-layer memory architecture. **(1) L1: index layer.** It stores compact pointers, including high-frequency entry points, keyword mappings, and a small set of hard constraints, serving as the core content of always-on memory for fast navigation and efficient memory routing. **(2) L2: fact layer.** It stores verified and stable factual information that remains valid over long periods. Admission is strictly controlled: information is written into L2 only after being validated through execution and proven reusable across tasks, deliberately excluding transient states, one-time events, and unverified hypotheses. **(3) L3: SOP layer.** It stores reusable procedural knowledge, including task workflows, preconditions, key execution steps, common failure cases, and corresponding debugging or recovery strategies. **(4) L4 : raw session archive layer.** Unlike L1–L3, L4 is not designed for frequent direct injection into the runtime context; its role is persistence and traceability, storing historical execution sessions so the agent can reconstruct prior trajectories, audit past decisions, and recover task context when needed. Within this hierarchy, L1 constitutes the always-on memory, L2 and L3 together make up long-term memory, and L4 provides durable storage for

persistence and traceability.

To manage the four-layer architecture, GA introduces a global meta-memory layer. The meta-memory layer defines the overall memory map, core rules, and update boundaries, providing the model with a shared frame of reference before execution so that it understands how memory is organized, what each layer is for, and how updates should be handled, thereby reducing arbitrary writes, historical misreads, and cross-task leakage. The full meta-SOP content is loaded on demand via file reads rather than always preloaded, keeping the always-on layer lightweight. In practice, GA injects only the meta-memory and L1 index layer by default, following an L1→L2/L3 routing chain to retrieve deeper factual or procedural knowledge only when needed, excluding irrelevant content from the active context. This routing is enforced through tool calls and prompt policy rather than a hardcoded mandatory pipeline.

For long-term consolidation, GA uses a triggered commit mechanism instead of immediate writing. When information is identified as potentially valuable, it enters a validation stage where its usefulness and reusability are checked before being committed. Only verified information is written into L2 or L3 through small, incremental updates, with the L1 index updated accordingly, preventing memory from growing in an uncontrolled manner over time.

A critical design invariant is that L1 remains bounded even as L2 and L3 expand. Each L1 entry records only the *existence* of a knowledge category rather than its substantive content. Consequently, new entries are introduced only when genuinely new categories arise, and the overall description length of L1 approaches the Kolmogorov complexity of the categorical structure of the knowledge set. This extreme degree of compression is feasible because the LLM itself serves as the decoder: once it infers that a relevant capability or fact exists, it can expend tool calls and tokens to retrieve the full content from deeper layers, so a minimal existence-level signal is sufficient for accurate routing.

2.3.3 Self-Evolution Capability in GA

GA implements self-evolution as an explicit and transparent process rather than an accidental phenomenon of reasoning. Achieving this requires clearly defining what evolves, how knowledge accumulates, how quality is controlled, and how the evolutionary trajectory is maintained.

What evolves: strategy, not tools. GA separates a fixed tool layer from an evolving knowledge layer. While the tool interface and user interactions work for any task and stay the same during operation, all task-specific capabilities are stored in SOP files and reusable scripts. The agent can easily read, create, and modify these using its own tools.

This separation ensures that learning new tasks doesn't interfere with existing skills, allowing knowledge to grow safely over time. Over multiple sessions, real-world feedback helps refine the SOPs. Common subtasks naturally evolve into stable, reusable scripts, upgrading the agent's knowledge from plain-text instructions to executable code.

How knowledge accumulates: hierarchical memory. Building on the hierarchical memory described in Section 2.3.2, GA ensures that knowledge gained during one session is immediately available in future ones. The active L1 index automatically tracks new L2 facts and L3 procedures. This eliminates the need for manual updates. Consequently, every successfully completed task permanently expands the agent's practical capabilities.

How quality is controlled: selective consolidation. GA saves raw action traces at the lowest memory level (L4), but it does not automatically promote these traces to higher, long-term memory levels, such as L2 or L3. Instead, reusable L3 procedures are generated only through explicit consolidation steps. These steps are triggered at meaningful milestones, such as successfully completing a subgoal or recovering from a system error.

During this consolidation phase, the agent only retains information that has been strictly verified through successful tool execution. Guesses, temporary intermediate states, and failed decision branches are systematically discarded under the "No Execution, No Memory" rule. This strict filtering prevents memory pollution, ensuring that the accumulated knowledge remains reliable and practically reusable in the future.

How the evolutionary trajectory is maintained: failure escalation. To prevent the agent from getting trapped in endless loops of incorrect actions, which would degrade performance rather than improve it, GA introduces a staged escalation mechanism for handling failures.

When a task fails, GA handles the error through a clear, three-step recovery process: First, it analyzes the immediate error message to make a small, localized adjustment and tries again. Second, if the failure persists, the agent abandons its current approach. It is forced to either switch to a completely new strategy or search the environment for missing information. Finally, if all automated attempts fail, the agent pauses and requests human intervention. This structured design prevents the agent from blindly repeating mistakes. By triggering progressively stronger corrective actions, it ensures the agent breaks out of dead ends and keeps its long-term learning on the correct path.

2.3.4 Context Truncation and Compression

The underlying model that GA relies on operates within a finite context window, and the cumulative conversation length must be kept below this limit at all times. Since GA cannot directly obtain precise token counts, context budget management uses a character-domain heuristic. Specifically, $C_{\mathcal{H}}$ denotes the **total character length** of all messages in the current conversation history, and B denotes the **character budget upper bound** converted from the token budget; the corresponding compression or eviction mechanism is triggered when $C_{\mathcal{H}} > B$:

$$C_{\mathcal{H}} = \sum_{m \in \mathcal{H}} \text{len}(m), \qquad B = \alpha \cdot W_{\text{tokens}}, \quad \alpha \approx 3 \text{ chars/token}, \tag{1}$$

where $\text{len}(m)$ denotes the character length of message m after JSON serialisation, W_{tokens} is the configured token window, and α is the empirical ratio used to approximate the token budget as a character budget. This heuristic has a certain margin of error. For ASCII-dominated content, each token corresponds to roughly 4 actual characters, so the α=3 setting slightly underestimates character efficiency, causing mildly early eviction—a conservative but safe failure mode. For CJK content the situation reverses: each character typically consumes 1–2 tokens, so the α=3 ratio substantially underestimates actual token usage, risking delayed eviction and potential context overflow.

GA uses four distinct context-trimming mechanisms to manage context length, ranging from fine-grained edits to broad removals. First, **tool-output truncation** controls the size of individual messages. Second, **tag-level compression** removes low-value fragments from older entries. Third, **message eviction** deletes the oldest content entirely when the overall token budget is exceeded. Finally, a **working-memory anchor** ensures that critical task information remains visible even after prior messages are evicted. Together, these mechanisms prevent the active context from growing linearly as the interaction continues. They ensure that the limited context budget is always strictly focused on information directly relevant to the current task.

Stage 1: Tool-output truncation. Each tool handler limits its return value *before* adding it to the message history. If the output exceeds a predefined character threshold (L), the system retains only the first and last $L/2$ characters, replacing the middle section with an ellipsis. Table 1 lists the thresholds for outputs that enter the LLM context.

Stage 2: Tag-level compression. Older messages tend to accumulate redundant text inside XML tags (such as reasoning traces, tool invocations, and working-memory snapshots). To reduce computational overhead, the system runs a compression pass roughly every five turns. This pass handles redundancy in two ways: (i) **Placeholder replacement**: Repeated working-memory blocks (like (`<history>`, `<key_info>`)) are replaced with a short placeholder, as only the most recent snapshot is actually needed. (ii) **Windowe truncation**: Content inside reasoning and tool tags (like (`<thinking>`, `<tool_use>`, `<tool_result>`) is truncated, keeping only an ∼800-character window (the beginning and the end of the text). To ensure the model always has the latest context, the 10 most recent messages are exempt from this compression. The five-turn interval also helps with efficiency: the unchanged older messages result in prompt-cache hits in roughly 80% of turns.

Table 1 Per-tool truncation thresholds for content entering the LLM context.

Tool / mode	L (chars)
`code_run`	10 000
`web_execute_js`[†]	8 000
`web_scan` (`text_only`)	10 000
`web_scan` (HTML)[‡]	35 000
`file_read`	∼1 280/line; 20 000 total

[†] With `save_to_file`, the full output is written to disk; only a short preview enters the history.

[‡] DOM-level character budget (subtree pruning), not head–tail truncation.

Stage 3: Message eviction. When a new message causes the total history length ($C_{\mathcal{H}}$) to exceed the character budget (B), the system triggers an eviction process. First, it reruns the Stage 2 compression using stricter rules, exempting only the 4 most recent messages. Next, it removes the oldest messages (FIFO order) until the history size drops below 60% of the total budget, leaving sufficient room for future turns.

After eviction, the newly exposed oldest message is cleaned up to maintain API consistency (for example, converting orphaned tool-result references into plain text). Importantly, evicted content is not completely lost: the Stage 4 anchor preserves essential task states, and the Stage 2 compression ensures that even a <30k context window can hold dozens of historical turns.

Stage 4: Working-memory anchor prompt. After every tool invocation, an anchor prompt is automatically attached to the next user message. This anchor contains: (i) the 20 most recent one-line turn summaries (each compressed to roughly one hundred characters), (ii) the current turn number, and (iii) a persistent `key_info` block maintained by the agent via `update_working_checkpoint`.

Because this anchor is injected into every new user message, Stage 2 compression automatically replaces the older copies with placeholders. This ensures only the newest version carries the full data payload. Once older messages are evicted in Stage 3, this anchor becomes the sole source of long-term memory, keeping task-critical information active in the context.

Auxiliary: tool-schema elision. When using the text-protocol path, if a tool's definition hasn't changed from the previous turn, its full schema is removed from the prompt and replaced with a brief natural-language reminder. To prevent the model from forgetting the exact tool formats over time, the full schema is periodically re-sent—either after a fixed number of turns or when the active prompt gets too long. (Note: This optimization is not used in the native API path, which requires sending complete tool definitions on every single call.)

3 From Minimal Architecture to Emergent Capabilities

3.1 Minimal Architecture

GenericAgent exhibits architectural minimality in two dimensions: **code minimality** and **interface minimality**.

At the code level, GA is exceptionally lightweight. The core codebase contains only about 3,300 lines of code, with the central Agent Loop running on a mere 92 lines. This stands in sharp contrast to comparable systems-OpenClaw's codebase is approximately 530,000 lines, more than 160 times larger than GA's. Rather than relying on heavy module dependencies or complex tool-registration infrastructures, GA's core logic is fully expressible within a minimal codebase, making the system easier to maintain, debug, and extend.

At the interface level, GA exposes itself as a self-hosted CLI program, making the command line not a wrapper around an internal platform, but the native execution surface of the system. Tasks can be launched with arguments or kept running in the background. GA therefore does not require plugin frameworks, event buses, or dedicated orchestration layers to support higher-level behavior. All interactions with GA—whether

task submission, progress monitoring, or runtime intervention—are handled through this single unified CLI interface.

This extreme minimality is not a functional compromise, but a deliberate design choice that unlocks greater power. Because the architecture is stripped down and the interface is purely CLI-based, the system does not need new architectural layers to support advanced behaviors. Instead, features like **Subagent Dispatch** and **Reflect Mode** emerge naturally. For instance, because GA operates as a standalone CLI tool, an agent can spawn a subagent simply by executing a standard terminal command. Complex AI behaviors are thus built from a single, elegant system primitive rather than layered engineering complexity.

3.2 Compositional Capabilities

A recurring pattern in agent framework design is to introduce dedicated subsystems for each new capability—for example, a subagent manager for multi-agent coordination, or a dedicated listener daemon for event handling. GA takes a fundamentally different approach: once the agent is exposed as a standard CLI program, two important higher-level capabilities emerge naturally from this single primitive, without any extension to the core architecture.

Subagent Dispatch. Once the agent can be invoked programmatically via CLI, subagents follow naturally. To handle complex subtasks, a parent agent simply runs a standard terminal command to launch multiple GA instances in the background. In this framework, a "subagent" is not a special object, nor does it require a dedicated manager. Parent and child are simply standard, independent processes communicating through the same interaction protocol.

Crucially, context isolation is naturally guaranteed: each instance runs in its own memory space and maintains its own conversation history, ensuring that subtasks never interfere with one another. For highly parallel tasks, this creates an elegant map-reduce workflow: the parent prepares independent inputs, spawns one subagent CLI process per input, and seamlessly merges the results once all child processes finish.

Reflect Mode. Similarly, once the CLI can be invoked programmatically, daemon-like behavior emerges naturally. GA's Reflect Mode requires no user instruction; instead, it continuously monitors for environmental changes and automatically triggers the corresponding task once a specific condition is detected. The mechanism is straightforward: a lightweight script periodically checks conditions. When a rule is triggered, the script simply dispatches the returned string as a standard task to the GA CLI. This strictly separates the triggering logic from the execution logic: the external script decides when to create work, while the stable core runtime focuses entirely on how to execute it.

Watchdog and Scheduled Task are two concrete applications of this Reflect Mode:

- **Watchdog:** Monitors for changes in the environment (e.g., a new file or an error log) and triggers GA immediately upon detection.
- **Scheduled Task:** Uses time-based rules to generate GA tasks at specific intervals or exact times.

Both share the exact same underlying mechanism and only differ in how the external trigger script is written. Because these rules are maintained as standalone scripts outside the agent, developers can update them at runtime without ever needing to restart the core agent process.

The common thread across these two capabilities is that neither required extending the core loop. Both emerge from the CLI as a single primitive, and the core runtime remains stable throughout. This reflects the deeper value of architectural minimality: a sufficiently simple system can support the widest range of behavioral compositions at the lowest possible cost.

3.3 Autonomous Exploration Capability

In real long-horizon environments, relying solely on user interaction to accumulate skills is fundamentally inefficient. User instructions are usually sparse and focused on immediate tasks, meaning they cannot systematically cover the full range of what the agent is capable of doing. Therefore, autonomous learning is a necessary feature for the continuous evolution of the agent.

The two capabilities established in Section 3.2 together make this possible without any new architectural machinery. Subagent dispatch provides the execution substrate: the agent can programmatically launch child instances to handle exploration tasks in parallel. Reflect Mode provides the trigger substrate: it continuously monitors for idle conditions and fires an exploration task the moment one is detected, with no user instruction required. Autonomous exploration is therefore not a new subsystem, but the natural result of combining these two primitives—the dispatcher becomes the agent itself rather than the user. What remains to be defined is the internal logic that decides *what* to explore and *how* to evaluate the result; the rest of the execution path is already in place.

GA organises accumulated skills into a persistent *skill tree*. The skill tree is a two-level map: *categories* (e.g. `web_automation`, `data_processing`) contain named *skills*, each recording its associated tool scripts and a monotonically increasing usage counter. This skill tree serves a dual purpose: it acts as a global index of the agent's current capabilities, and it is the core dataset driving its autonomous exploration decisions.

Trigger Mechanism. Autonomous exploration can be initiated in two ways. In *task mode*, an external orchestrator (e.g. a cron job) submits a one-shot task via a file-system mailbox and polls for results. In *reflect mode*, a user-supplied callback is polled at a fixed interval; when it returns a non-empty string, that string is injected as a new prompt. The default trigger fires every six minutes with a fixed prompt directing the agent to consult its exploration procedure. Both modes are concrete instantiations of the Reflect Mode primitive introduced in Section 3.2, and require no extension to the core architecture.

Exploration Task Generation. A core question in autonomous exploration is determining what to explore. When no pending task list exists, the agent enters planning mode. The curriculum planner inspects the current skill tree and scores task candidates along four dimensions using a weighted sum:

$$S(t) \;=\; w_b\, B(t) \;+\; w_d\, D(t) \;+\; w_u\, U(t) \;+\; w_i\, I(t) \tag{2}$$

where B (*breadth*) rewards filling under-populated skill categories, D (*depth*) rewards enhancing frequently-used skills, U (*utility*) captures estimated practical value, and I (*innovation*) favours novel techniques or domains. Each dimension is scored on a 1–10 scale.

Breadth and depth are computed directly from skill-tree statistics:

$$B(t) = 10 \times \max\left(0,\, 1 - \frac{|S_c|}{\bar{S} + 1}\right), \tag{3}$$

$$D(t) = 10 \times \frac{u(t)}{u_{\max} + 1}, \tag{4}$$

where $|S_c|$ is the number of skills in the target category, $\bar{S}$ is the mean skill count across all categories, $u(t)$ is the usage count of the target skill, and $u_{\max}$ is the maximum usage count across all skills. The breadth formula rewards filling categories that are below average, while the depth formula prioritises enhancing frequently-invoked capabilities. Utility and innovation are estimated by the LLM on the same 1–10 scale.

The weights are initially set to $(w_b, w_d, w_u, w_i) = (0.3, 0.2, 0.3, 0.2)$, prioritising breadth and utility. The resulting task list must span at least four distinct skill categories, preventing narrow concentration. Planning and execution are strictly separated: the planner produces a task list and immediately yields control; execution resumes in the next invocation.

Execution and Consolidation. Each autonomous task proceeds through a fixed sequence: *context loading* (retrieve recent history and pending tasks), *skill search* (query the skill tree for relevant tools and identify gaps), *execution* (research, prototype, and validate within a sandboxed temporary directory), *report writing* (a Markdown document with machine-readable metadata tags for automatic skill-tree updates), *skill consolidation* (an atomic operation that archives the report, updates the skill tree, and increments usage counters), and *task-list advancement*. Importantly, even failed experiments are thoroughly documented. This ensures that future planning rounds can read the history and avoid repeating the same dead ends. As a strict security measure, all generated files are confined to the temporary directory. Access to system secrets or core source code is unconditionally prohibited.

Furthermore, the system maintains a separate lightweight log that records three specific types of information: observed errors paired with their corrections, explicit user preferences, and verified success patterns. These entries are automatically injected into the system prompt to guide the agent's future behavior and improve its decision-making.

Exploration Quality Assessment. The second core question is evaluating exploration quality. After each batch of autonomous tasks, a *reflection-based adaptation* mechanism adjusts the scoring weights based on actual usage outcomes. For each completed task t, if its predicted score $S(t) > 8.0$ but actual usage $u(t) < 3$ within 30 days, the weight of the dominant dimension in t's score vector is reduced by 10%. Conversely, if $S(t) < 5.0$ but $u(t) > 5$, the corresponding weight is increased by 10%. After these adjustments, all weights are re-normalized to ensure they sum to 1. This feedback loop allows the system to automatically discover which dimensions best predict practical value in the user's specific workflow. As a result, GA can continuously correct its future exploration decisions without requiring any manual tuning.

Limitations. Several limitations merit discussion:

- **Context Length Constraints:** The 30-round execution cap means highly complex research tasks may span multiple sessions. Currently, continuity between these sessions is maintained only through written reports and task-list annotations.
- **Unverified Adaptation:** The reflection-based weight adjustment is a preliminary design. It has not yet accumulated enough long-term data to definitively prove its effectiveness across diverse, real-world user workflows.
- **Manual Log Curation:** The self-improvement log (which records errors and preferences) currently relies on manual user curation, limiting how much it can autonomously influence the agent's core behavior.
- **Manual Tree Maintenance:** Advanced skill tree management, such as merging redundant categories, deprecating outdated tools, or restructuring the tree topology, remains entirely manual.

4 Evaluation

In this section, we evaluate the GenericAgent to systematically analyze its system behavior, resource management, and core mechanisms. Our evaluation is structured around five complementary dimensions:

1. **Task Completion and Token Efficiency:** We measure overall success rates and token consumption across diverse benchmarks to quantify the GA's fundamental execution capability and operational cost.
2. **Tool-Use Efficiency:** We analyze the minimal atomic-tool design to assess how a restricted tool space impacts the ability to resolve complex workflows and the resulting interaction overhead.
3. **Memory System Effectiveness:** We investigate the memory management mechanisms to examine their function in long-term fact retention, performance dynamics across repeated tasks, and context size control.
4. **Self-Evolution Capability:** We evaluate the self-evolution pipeline to observe the process and effects of compressing historical interaction trajectories into reusable Standard Operating Procedures (SOPs) and executable code.
5. **Web Browsing Capability:** We use open-ended web tasks to test the framework's end-to-end navigation, multi-hop retrieval, and multi-step execution in a dynamic, unstructured real-world environment.

4.1 Task completion and token efficiency

This section evaluates the fundamental execution capability and resource consumption of the agent systems. The primary objective is to investigate the baseline performance-cost trade-off—specifically, how efficiently a system translates token consumption into successful task execution. By measuring accuracy alongside context usage across diverse scenarios, this experiment aims to quantify the operational cost of maintaining high performance and verify whether the system's underlying context management effectively reduces token overhead without compromising task reliability.

Table 2 Task completion rate and token efficiency across the main agent benchmarks and RealFin-benchmark. Each block reports the original metrics used in the corresponding benchmark. Since the efficiency ratio is normalized with different token scales across benchmarks, its absolute values should only be compared within the same benchmark block.

Agent	Model	Accuracy	Input Tokens	Output Tokens	Total Tokens	Efficiency
SOP-Bench						
GA	Claude Sonnet 4.6	100%	2.02M	53k	2.08M	0.48
OpenClaw	Claude Sonnet 4.6	100%	2.60M	40k	2.64M	0.38
Claude Code	Claude Sonnet 4.6	85%	1.23M	23k	1.25M	0.68
GA	Minimax M2.7	90%	893k	32k	924k	0.97
OpenClaw	Minimax M2.7	95%	2.91M	46k	2.96M	0.32
Lifelong AgentBench						
GA	Claude Sonnet 4.6	100%	222k	20k	241k	4.15
OpenClaw	Claude Sonnet 4.6	70%	1.43M	21k	1.45M	0.48
Claude Code	Claude Sonnet 4.6	75%	800k	14k	814k	0.92
GA	Minimax M2.7	90%	400k	23k	423k	2.12
OpenClaw	Minimax M2.7	70%	1.20M	17k	1.22M	0.57
RealFin-benchmark						
GA	Claude Sonnet 4.6	65%	102k	12k	114k	5.70
Claude Code	Claude Opus 4.6	60%	290k	17k	307k	1.95
Claude Code	Claude Sonnet 4.6	55%	226k	12k	238k	2.31
OpenClaw	Claude Sonnet 4.6	35%	249k	2k	251k	1.39
Codex	GPT-5.4	60%	838k	54k	892k	0.67

4.1.1 Setup

Baseline. To contextualize the performance of **GA**, we establish a comparative evaluation against three representative agent systems: (1) **Claude Code** (an industry-leading proprietary agent), **OpenClaw** (a robust open-source multi-tool framework), and **CodeX** (a widely adopted execution baseline). Furthermore, to prevent model-specific biases and ensure the generalizability of our findings, the evaluations are conducted across a diverse spectrum of cutting-edge LLMs. The backbone LLMs deployed include *Claude 4.6 Sonnet* and *Claude 4.6 Opus* for top-tier proprietary capabilities, alongside *MiniMax M2.7* and *GPT-5.4*, covering different architectural paradigms and reasoning capacities.

Benchmark. The evaluation is conducted on three distinct benchmarks to assess distinct facets of agent capabilities under complex, real-world constraints: (1) **SOP-Bench** [23]: This benchmark evaluates the execution of multi-step SOPs, measuring the agent's instruction-following and procedural reasoning capabilities. (2) **Lifelong AgentBench** [9]: This benchmark consists of sequential tasks with explicit cross-task dependencies, designed to evaluate the agent's ability to maintain state and manage context over continuous interactions. (3) **RealFin-Benchmark** [24]: This benchmark comprises financial workflows, used to assess domain-specific understanding and task execution in professional environments.

Metrics. We assess performance and operational cost using a multi-dimensional metric suite. The primary behavioral metric is **Accuracy**, which measures the success rate of task completion. To quantify operational overhead, we report **Input Tokens**, **Output Tokens**, and their sum, **Total Tokens**. Finally, we define **Efficiency** as the accuracy achieved per million total tokens consumed, formalized as Efficiency = Accuracy/Total Tokens (M).

4.1.2 Results

GA consistently achieves state-of-the-art or highly competitive task completion rates. As detailed in Table 2, GA's robust performance is consistent across all three benchmarks. In the Claude-based configurations,

Table 3 Source-level tool inventory overview of Claude Code, OpenClaw, and GA. We list representative native tools for Claude Code and OpenClaw and show the full atomic tool set of GA. The counts refer to source-level built-in tools/tool factories rather than the exact runtime-visible tool set, which may vary with configuration, feature flags, permissions, and plugin-based dynamic injection.

Claude Code	OpenClaw	GA
`AgentTool`	`browser`	`file_read`
`TaskOutputTool`	`canvas`	`file_write`
`BashTool`	`nodes`	`file_patch`
`GlobTool`	`cron`	`code_run`
`GrepTool`	`message`	`web_scan`
`FileReadTool`	`tts`	`web_execute_js`
`FileEditTool`	`gateway`	`ask_user`
`FileWriteTool`	`agents-list`	`update_working_`
`NotebookEditTool`	`sessions-list`	`checkpoint`
`WebFetchTool`	`sessions-history`	`start_long_`
...	...	`term_update`
53 source-level built-in tools: 20 base + 33 conditional.	*18 source-level tool factories; runtime may include plugins.*	*only 9 atomic tools*

GA achieves 100% accuracy on both SOP-Bench and Lifelong AgentBench. This perfectly matches the highest baseline performance on SOP-Bench and strictly outperforms all competitors on Lifelong AgentBench. This superiority extends to the RealFin-Benchmark, where GA attains the highest overall accuracy of 65%, comfortably exceeding Claude Code (60% with *Claude Opus 4.6* and 55% with *Claude Sonnet 4.6*), CodeX (60%), and OpenClaw (35%).

GA significantly reduces token consumption, particularly in input contexts. Beyond absolute task success, GA demonstrates marked token efficiency. On Lifelong AgentBench, GA requires only 222k input tokens, representing a drastic reduction compared to Claude Code (800k) and OpenClaw (1.43M), while seamlessly maintaining its flawless 100% accuracy. On SOP-Bench, GA consumes 2.02M input tokens—substantially less than OpenClaw (2.60M). Although Claude Code uses nominally fewer tokens (1.23M) on this specific benchmark, it does so at the steep cost of task completion (as analyzed below). Similar cost-saving patterns are consistently observed in the RealFin-Benchmark.

GA achieves the strongest overall performance in task completion and efficiency across the main benchmarks. Within each benchmark, GA achieves the best combined result in terms of task completion and token efficiency. On Lifelong AgentBench, GA reaches 100% accuracy with an efficiency ratio of 4.15; on RealFin-benchmark, it attains 65% accuracy with an efficiency ratio of 5.70. On SOP-Bench, although Claude Code reaches a higher efficiency ratio of 0.68, its accuracy drops to 85%, whereas GA maintains 100% accuracy with an efficiency ratio of 0.48.

4.2 Tool-use efficiency

This section evaluates the impact of tool space design on an agent's operational efficiency and problem-solving capability. The primary objective is to investigate whether a minimal, atomic-tool abstraction can effectively handle complex workflows that typically rely on a large inventory of specialized tools. By analyzing tool usage distributions, interaction overhead, and task success rates, this experiment aims to verify if condensing the action space into a small set of core composable functions can mitigate prompt bloat and reduce interaction cycles without compromising the agent's functional versatility.

4.2.1 Setup

Baseline. All compared systems use the same backbone model, *Claude Sonnet 4.6*. We evaluate three representative agents: (1) **GA**, (2) **Claude Code**, (3) **OpenClaw**.

Table 4 Results on long-horizon complex tasks. The five tasks cover document generation (PDF/PPT creation), SQL copilot query generation, experiment analysis report writing, procurement decision-making with web retrieval, and research paper reproduction feasibility analysis. This table reports the average outcomes over the long-horizon task set rather than expanding into per-task detail.

Agent	#Tasks	Success	Total Tokens	Time (s)	Requests	Tool Calls
Claude Code	5	100.0%	537,413	320.8	32.6	22.6
GA	5	100.0%	188,829	220.8	11.0	12.8
OpenClaw	5	80.0%	633,101	183.1	15.0	16.6

Tool. Table 3 first gives a source-level inventory overview. Claude Code exposes 53 built-in tools at the source level (20 base tools and 33 conditional tools), while OpenClaw provides 18 source-level tool factories, with the runtime-visible set further varying by plugin injection and environment configuration. GA does not attempt to reproduce these full inventories one by one. Instead, it keeps only a small set of core atomic tools that cover the capabilities most frequently required in our evaluation tasks, and each of these capabilities has a clear counterpart in both Claude Code and OpenClaw. Appendix 1 provides both a capability-level alignment (Table 10) and representative task-level substitution examples (Table 11).

Benchmark. The benchmark contains two complementary task groups. (1) **Simple tool-generalization tasks**, which test whether baseline specialized tool capabilities can be reproduced through GA's atomic-tool composition. (2) **Long-horizon complex tasks**, which test whether the same minimal tool design remains effective on realistic multi-step workflows.

Metrics. We report **Success**, **Total Tokens**, **Time**, **Requests**, and **Tool Calls**. **Success** denotes the fraction of tasks completed successfully in the evaluated task set. The remaining metrics measure execution cost and interaction overhead: **Total Tokens** is the total token consumption, **Time** is the average runtime in seconds, **Requests** is the average number of model requests, and **Tool Calls** is the average number of tool invocations.

4.2.2 Atomic Tool Generalization

GA can solve complex long-horizon tasks through compositions of atomic tools rather than through a richer specialized tool inventory. In Table 4, GA reaches 100% success on the five long-horizon tasks, matching Claude Code and outperforming OpenClaw, which falls to 80%. This shows that a minimal tool set does not reduce the range of tasks that can be solved when the tools are composable in the right way. Appendix 1, especially Table 11, provides concrete substitution examples showing how several specialized baseline capabilities can be reconstructed through short atomic-tool compositions.

GA reduces token overhead while preserving task performance. Although it matches Claude Code on success, GA uses only 188,829 accounted tokens, which is 35.1% of Claude Code's 537,413 and 29.8% of OpenClaw's 633,101. The same pattern appears in the interaction statistics: GA reduces requests from 32.6 to 11.0 and tool calls from 22.6 to 12.8 relative to Claude Code, while OpenClaw remains both less stable and more expensive. These results indicate that GA's advantage comes from lower context and tool-process overhead, not from sacrificing success rate.

4.2.3 Tool Usage Distribution

Tool usage is highly concentrated in a few high-frequency tools, while the remaining long-tail tools are invoked only rarely. As shown in Figure 3, although Claude Code and OpenClaw expose dozens of built-in tools or tool factories, their actual execution traces are still dominated by a small subset of tools. In Claude Code, `AgentTool` alone accounts for 50.4% of calls, followed by `WebFetchTool` (22.1%), `FileReadTool` (10.6%), and `FileWriteTool` (8.9%), while many other tools appear only in the tail. OpenClaw shows a similar concentration pattern. This means that a substantial number of low-frequency tools still occupy prompt context and enlarge the action space even though they contribute little to actual execution. By comparison,

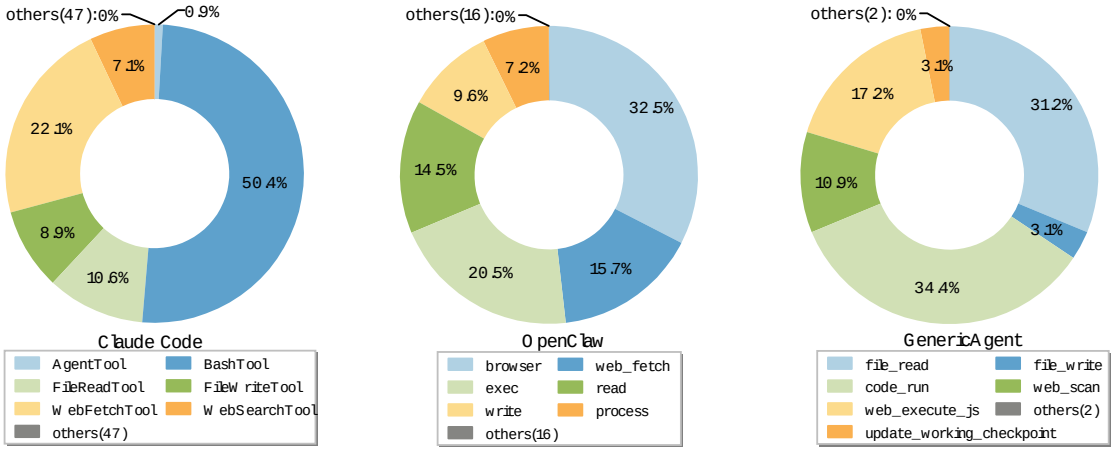

Figure 3 Tool usage distribution across Claude Code, OpenClaw, and GA. Each donut chart summarizes the proportion of tool calls attributed to each major tool category in the corresponding system.

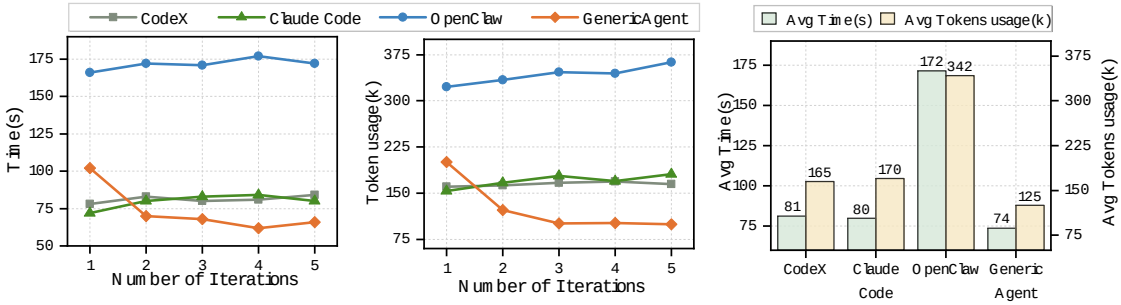

Figure 4 Operation time and token consumption across five repeated runs for different agents. While the baseline agents remain relatively stable over time, GA shows a clear efficiency convergence pattern, with both runtime and token cost decreasing substantially as task experience is accumulated and distilled into reusable memory.

GA makes the active tool loop explicit through a compact atomic-tool set centered on `code_run`, `file_read`, `web_execute_js`, and `web_scan`, which is consistent with its lower request count, lower tool-call count, and lower token cost in Table 4.

4.3 Memory System Effectiveness

This section evaluates the architectural design and operational impact of GA's hierarchical memory system. The primary objective is to investigate whether the framework can effectively accumulate, distill, and retrieve historical experience without incurring the prohibitive cost of context explosion. By systematically analyzing performance dynamics across repeated tasks, memory compression strategies, factual retention mechanisms, and prompt growth over prolonged usage, this experiment aims to verify that high-density, filtered memory significantly enhances continuous learning while strictly maintaining an optimal context boundary.

4.3.1 Continuous Efficiency Improvement

Setup. To evaluate whether the Condensed Memory framework enables GA to improve through continuous real-world use, we assess the performance trajectories of different agents across repeated runs. We compare **CodeX**, **Claude Code**, **OpenClaw**, and **GA**, all utilizing *GPT-5.4* as the backbone model. Each experimental run involves downloading five distinct datasets from HuggingFace. To strictly avoid contamination from residual conversational context, every task is executed within a completely fresh session.

Table 5 Memory ablation on SOP-Bench dangerous_goods. Condensed memory achieves the best cost-effectiveness.

Configuration	Memory Size (tokens)	TSR (%)
No-Memory	0	13.87
Full-Memory	575	52.44
Redundant-Memory	288	66.48
Condensed memory	165	**66.48**

Results. **GA continuously improves operational efficiency through repeated use by successfully converting raw task experience into reusable memory.** As shown in Figure 4, CodeX, Claude Code, and OpenClaw remained largely stable across runs, whereas GA improved markedly, with operation time decreasing from 102 seconds in the first run to approximately 66 seconds in later runs, and token consumption dropping from 200,439 to 100,000. This steady reduction indicates that GA does not simply accumulate history, but converts task experience into reusable L3 SOPs that reduce repeated understanding, unnecessary reasoning, and decision overhead in later runs. More importantly, the gain comes not from storing more content, but from preserving high-value information that the model does not already know and that directly affects behavior. Overall, the results show that GA becomes more efficient through continued use, which is precisely the intended effect of its memory system.

4.3.2 Effectiveness of Condensed memory

Setup. To examine whether retaining only decision-critical rules yields better performance at a lower contextual cost, we compare four distinct memory configurations under the GA framework (*GPT-5.4*): (1) **No-Memory**: Operates solely on task inputs and tool descriptions without external memory; (2) **Full-Memory**: Injects the complete, unedited SOP as external knowledge; (3) **Redundant-Memory**: Extends Condensed Memory with background descriptions, definitions, and weakly relevant information; (4) **Condensed Memory**: Retains only high-density, action-guiding rules. The evaluation is conducted on the dangerous goods subset of SOP-Bench [23]. We use **Task Success Rate (TSR)** to measure behavioral performance and Memory Size to measure contextual cost, and we further compare the performance gain achieved per unit of token cost across different memory settings.

Results. **Filtered, high-density memory provides optimal behavioral guidance while drastically minimizing contextual burden.** As shown in Table 5, No-Memory performed substantially worse than all memory based settings, which shows that external procedural memory is necessary for this task. Condensed memory achieved the same highest TSR as Redundant-Memory and outperformed Full-Memory while using far fewer tokens, which indicates that filtered high density memory is more effective than the original full SOP. Redundant-Memory provided no gain over Condensed memory despite using more tokens, which suggests that background descriptions, definitions, and explanatory text add contextual burden without contributing additional behavioral value. Overall, this experiment shows that GA benefits most from memory that stores only the small set of task critical information the model does not already know.

4.3.3 Long-Term Fact Retention

Setup. This experiment examines whether GA has long term factual memory and whether it can remain competitive on factual recall and multi hop reasoning tasks without relying on an additional embedding model or a vector database. The compared methods include embedding based approaches, namely Mem0 [14] and A-MEM, and non embedding approaches, namely OpenClaw and GA. All methods use *GPT-5.4*, and for the embedding based methods we adopt *text-embedding-3-small*) as the embedding model. The evaluation is conducted on the first subset of LoCoMo [25], with Category 5 summary tasks removed to avoid interference from summarization ability. We use F1 to measure the correctness and completeness of factual content, and BLEU-1 to measure the lexical similarity between the generated answer and the reference answer.

Table 6 Long-term factual memory evaluation on LoCoMo. GA outperforms embedding-based systems without using external retrieval.

	Multi-Hop		Temporal		Open-Domain		Single-Hop	
System	**F1**	**BLEU**	**F1**	**BLEU**	**F1**	**BLEU**	**F1**	**BLEU**
Mem0	39.32	28.43	50.03	43.33	18.32	13.80	40.32	32.33
A-MEM	29.03	20.48	46.83	38.84	13.11	12.94	44.68	37.01
OpenClaw	21.43	18.41	22.56	20.43	9.56	9.03	23.44	24.21
GA	**43.33**	**39.96**	**52.23**	**51.11**	**20.41**	**15.31**	**45.69**	**40.66**

Table 7 Full prompt length after installing 20 skills and intensive usage, measured on a minimal input. GA prevents context explosion.

System	Full Prompt Length (tokens)
Claude Code	22,821
CodeX	23,932
OpenClaw	43,321
GA	**2,298**

Results. **Precise memory organization enables GA to exceed the factual retention capabilities of dedicated vector-based retrieval systems.** As shown in Table 6, GA achieved the best F1 and BLEU-1 scores across all four task categories, which indicates that its memory mechanism supports both accurate factual recall and stable answer generation. Its advantage was especially clear on Multi-Hop and Temporal tasks, which suggests that GA can not only retain long term facts but also use them effectively in reasoning across time and across fact chains. Even in Open-Domain, which was the most difficult category for all methods, GA still achieved the highest score, and this shows that its memory organization remains effective even when structural cues are weak. Overall, these results show that long term factual memory can be achieved through precise memory organization and filtering, and that GA does not need an additional embedding model or a vector database to build this capability.

4.3.4 Context Explosion Prevention

Setup. To evaluate whether GA can prevent memory explosion under long term use and continuous skill expansion, we compare **GA**, **Claude Code**, **CodeX**, and **OpenClaw** after installing the same set of 20 skills for each agent and using them intensively over time. We then issue a minimal request "Hello" and measure the Full prompt length of each agent to assess how well its memory system controls context growth after prolonged use.

Results. **Hierarchical retrieval strictly isolates idle memory from the active prompt, eliminating the risk of context explosion.** As shown in Table 7, the Full prompt length of GA is far lower than that of Claude Code, CodeX, and OpenClaw, which shows that GA can keep context growth under control even after long term use and continuous skill expansion. This advantage comes from its hierarchical memory management, where memory is not directly injected into the context but is retrieved only when the model actually needs it. Overall, the results show that GA can preserve long term memory capability without causing uncontrolled context expansion.

4.4 Self-Evolution Capability

This section evaluates the framework's capacity for self-evolution, specifically its ability to distill and reuse experiential knowledge over time. The primary objective is to investigate whether an agent can systematically transition from high-entropy, trial-and-error exploration to a deterministic, low-cost execution regime. By tracking performance trajectories across repeated interactions and varying task complexities, this experiment

Table 8 Nine-round evolution trajectory on the LangChain GitHub research task. The experiment starts from scratch in Round #1, iteratively refines a textual SOP in Rounds #2–#5, and enters the codified execution regime in Rounds #6–#9.

Round	Stage	Time	LLM Calls	Input	Output	Cache Create	Cache Read	Total
#1	Initial run	7m30s	32	15,581	7,647	15,600	183,375	222,203
#2	SOP optimization	4m19s	12	5,045	4,860	4,553	51,883	66,341
#3	SOP optimization	2m53s	8	2,538	4,598	3,155	39,534	49,825
#4	SOP optimization	2m29s	9	3,265	3,682	3,435	41,376	51,758
#5	SOP optimization	2m50s	7	2,568	3,276	2,424	27,268	35,536
#6	Codified SOP	2m24s	6	1,855	1,064	1,930	20,913	25,762
#7	Codified SOP	1m41s	5	1,931	1,126	1,708	18,249	23,014
#8	Codified SOP	1m35s	5	1,884	990	1,586	18,229	22,689
#9	Codified SOP	1m38s	5	1,323	994	1,659	19,034	23,010

aims to verify if the reflection-driven pipeline effectively compresses historical interaction trajectories into reusable SOPs and executable code, thereby permanently eliminating redundant reasoning cycles and preventing stagnation loops in long-horizon deployments.

4.4.1 Setup

Baseline. To rigorously evaluate the cross-task generalization of the self-evolution mechanism, we establish a direct comparison between **GA** and **OpenClaw**. To ensure a fair baseline and isolate the impact of system architecture from model capability, both agents utilize *Claude Opus 4.6* as their underlying backbone model for the comparative benchmark.

Evaluation Setting. We evaluate GA's self-evolution process across two complementary experimental settings: (1) **Longitudinal Study**: A nine-round sequential evaluation where GA repeatedly performs GitHub research tasks. Each round operates on a new task instance, allowing us to trace the full evolutionary trajectory and observe how accumulated experience translates into progressive efficiency gains on structurally similar workflows. (2) **Cross-Task Benchmark**: An eight-task web benchmark designed to rigorously test whether the efficiency convergence observed in the longitudinal study generalizes across diverse task types and environments when compared against the OpenClaw baseline. The primary metric for evaluating self-evolution efficiency is **Total Tokens** (token consumption), which is tracked across repeated executions to quantify the reduction in contextual and reasoning overhead.

Evolution Stages. To systematically analyze the evolution process, we categorize GA's capability transformations into three distinct stages: (1) **Stage 1 (Natural-language execution)**: The agent solves the task primarily through in-context reasoning, exploratory tool use, and trial-and-error interaction. (2) **Stage 2 (SOP distillation)**: Accumulated experiential memory is compressed into a structured, textual SOP, filtering out exploratory missteps. (3) **Stage 3 (Code-based execution)**: The verified textual workflow is further crystallized into executable logic. Crucially, the transitions between these stages are triggered autonomously by GA's memory management mechanisms without manual intervention, serving as a core architectural advantage.

4.4.2 Iteration-Wise Efficiency Trajectory

The nine-round trajectory shows a phase transition from high-entropy exploration to a stable low-cost execution regime. As shown in Table 8, relative to Round #1, the final execution in Round #9 reduces runtime from 7m30s to 1m38s, a reduction of 78.2%, reduces LLM calls from 32 to 5, a reduction of 84.4%, and reduces total tokens from 222,203 to 23,010, a reduction of 89.6%. The key pattern is not just the large first-to-last gap, but the structure of the transition itself. During the textual SOP stage, token cost decreases overall from 66k to 36k, despite minor fluctuations, which indicates that SOP compression removes

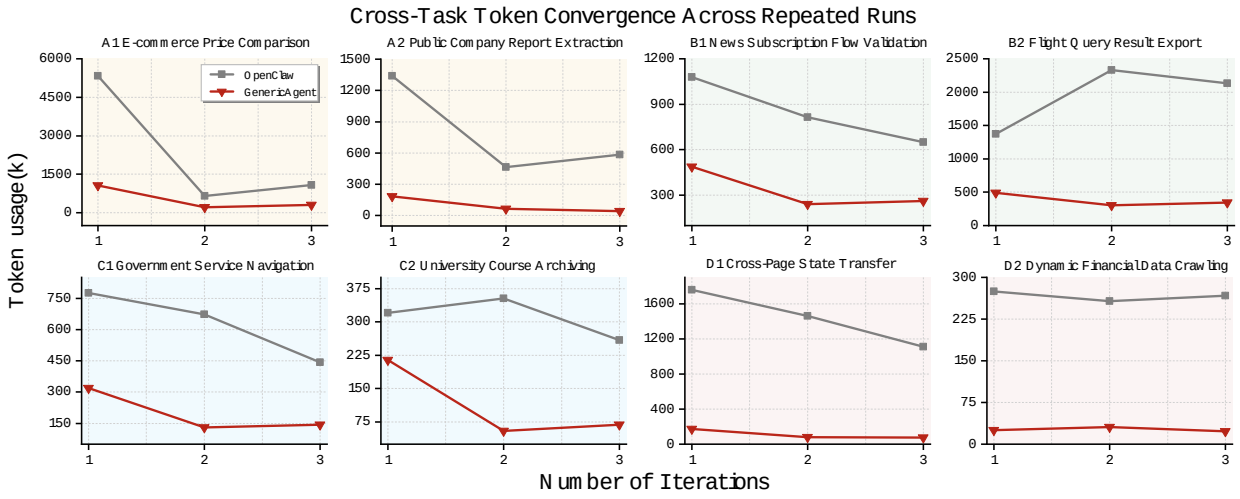

Figure 5 Cross-task token convergence across repeated runs for OpenClaw and GA. Each panel corresponds to one benchmark task and plots total token consumption over the first, second, and third executions.

most redundant exploration while still incurring a limited adaptation cost when the stored procedure must be aligned with newly encountered edge cases. Once the workflow is codified, the system enters a narrow convergence band of roughly 23k±1k tokens across Rounds #6–#9, indicating that execution has become predictable rather than exploratory.

Most of the efficiency gain comes from eliminating repeated reasoning cycles rather than merely shortening individual responses. The dominant change is the collapse in call count, which removes entire understand–reason–generate loops from the execution trace. This is consistent with the token composition: cache-read tokens fall from 183,375 in Round #1 to 19,034 in Round #9, and input tokens fall from 15,581 to 1,323, showing that the agent no longer rebuilds large amounts of context at runtime. Per-call token load also drops substantially over the course of evolution, from 6,944 tok/call in Round #1 to about 4.5k–4.6k tok/call in Rounds #7–#9, but this effect is secondary to the structural reduction in the number of calls. The larger implication is that textual SOPs already compress the search space, while code-level execution removes the remaining interpretation overhead and turns the learned workflow into an executable module.

4.4.3 Cross-Task Efficiency Gains

GA's self-evolving memory mechanism consistently improves efficiency across tasks rather than only helping on a single hand-optimized workflow. As shown in Figure 5, on all eight benchmark tasks, later GA executions consume fewer tokens than the first execution, with savings ranging from 61.0% to 92.4% and an overall reduction of 79.3%. This consistency matters more than any individual best case: it shows that SOP self-evolution is not just memorizing one task, but systematically converting prior successful trajectories into reusable execution shortcuts. The strongest gains appear in long-horizon tasks with state transfer and recovery (Category D, 92.0%), where repeated failure handling and path search dominate the initial execution cost. In those cases, SOP acts as path compression: once the verified route is stored, the agent can bypass most exploratory branches entirely.

GA exhibits a clear transition from cold start to rapid convergence across repeated executions of the same task. Across the three runs of each benchmark task, GA shows the same internal trend: the first run has substantially higher token cost, while the second and third runs quickly converge to a stable low-cost regime. This indicates that even with SOP memory, the first execution still incurs an SOP adaptation cost, because the agent must map generalized SOP knowledge onto the concrete page structure and interaction flow of the current task instance. However, this adaptation happens only once. After the successful path is aligned with the environment, later executions can directly reuse that result and rapidly drop to a stable token level. By contrast, OpenClaw shows no comparable convergence pattern across its three runs. On B2, for example,

token usage changes from 1,370k to 2,330k to 2,130k, which means the system keeps re-exploring rather than reusing prior experience. This distinction is important for deployment: GA pays a one-time adaptation cost on a new task, but its marginal cost quickly approaches zero, whereas the baseline remains expensive on every execution.

The benefit of SOP memory tends to increase with task complexity. When the tasks are ordered by OpenClaw's average token consumption as a proxy for intrinsic task complexity, a positive relationship emerges between complexity and saving rate. High-complexity tasks with OC averages above 1,000k tokens, such as A1, B2, and D1, achieve an average saving of 83.5%, while mid-complexity tasks average 72.5%. Although D2 is low in absolute token cost, its 90.1% saving rate is consistent with the same mechanism because it still contains long-horizon dependency structure. The broader implication is that SOP self-evolution becomes more valuable as tasks require more multi-step reasoning, navigation, and recovery from branching execution paths: in simple tasks it is helpful, but in complex tasks it becomes structurally decisive.

4.5 Web Browsing Capability

This section evaluates the agent's capability to navigate, search, and reason within open-ended web environments. The primary objective is to use the unstructured, noise-heavy nature of the web as a comprehensive stress test for the system's context management and multi-step execution. By requiring the agent to interact with dynamic web pages, filter irrelevant interface elements, and conduct multi-hop retrieval, this experiment aims to verify whether GA's principles of high-density context and hierarchical memory remain robust in real-world scenarios where prompt explosion from raw HTML or DOM structures is a typical failure mode.

4.5.1 Setup

Baseline. We compare **GA** against **OpenClaw** to contrast their architectural approaches in handling highly dynamic environments. To ensure a strictly fair comparison focused on system design rather than model capability, both GA and OpenClaw utilize *Claude 4.6 Opus* as the underlying backbone model.

Benchmark. We evaluate web browsing proficiency across three diverse benchmarks, ranging from atomic interactions to open-ended real-world workflows: (1) **WebCanvas** [26]: A benchmark measuring fundamental browser interactions, including navigation, element clicking, filtering, and localized information extraction. We randomly sample 12 tasks for evaluation. (2) **BrowseComp-ZH** [27]: A benchmark specifically designed to measure multi-hop web search and chain-based reasoning within the Chinese web ecosystem. We randomly sample 10 tasks for evaluation. (3) **Custom Tasks**: A curated set of 22 real-world tasks encompassing academic platforms, social media, content sites, utility websites, e-commerce platforms, and public retrieval services.

Metrics. Performance is evaluated using a normalized **Score (0–1)**. The scoring protocols are tailored to each benchmark: automatic evaluation for WebCanvas, LLM-as-a-judge for BrowseComp-ZH, and a rigorous human-in-the-loop review assisted by GPT-5.4 for Custom Tasks. To quantify operational overhead in web environments, we also report the **Avg. Tokens (M)** consumed per task.

4.5.2 Results

GA consistently outperforms the baseline in web navigation and reasoning while operating at a fraction of the token cost. As shown in Table 9, GA achieves superior scores across all three benchmarks compared to OpenClaw. On the fundamental WebCanvas benchmark, GA secures a score of 0.834 using only 0.18M tokens, whereas OpenClaw trails at 0.722 while consuming 0.71M tokens. This advantage extends to open-ended, real-world interactions: on Custom Tasks, GA scores 0.577 against OpenClaw's 0.500, while utilizing a mere 0.26M tokens compared to OpenClaw's 0.76M.

GA's multi-stage compression pipeline provides an advantage in long-horizon, multi-hop web search tasks. The performance gap becomes exceptionally pronounced on the BrowseComp-ZH benchmark, which demands deep reasoning chains. Here, GA achieves an impressive 0.600 accuracy, tripling OpenClaw's score of 0.200, while consuming roughly one-third of the tokens (0.47M vs. 1.31M). Without active memory filtering, baseline

Table 9 Web browsing evaluation results across three benchmarks. GA and OpenClaw both use *Claude Opus 4.6* as the backbone model.

Benchmark	Tasks	Evaluation	Score (0–1)		Avg. Tokens (M)	
			GA	OpenClaw	GA	OpenClaw
WebCanvas	12	Automatic	**0.834**	0.72	**0.18**	0.71
BrowseComp-ZH	10	LLM Judge	**0.60**	0.20	**0.47**	1.31
Custom Tasks	22	Human + LLM	**0.577**	0.50	**0.26**	0.76

systems like OpenClaw are easily overwhelmed by raw HTML and irrelevant DOM elements, leading to state confusion and context overflow. Conversely, GA preserves context density by systematically compressing web observations, enabling the model to sustain deep, multi-hop logical reasoning without losing track of the initial objective.

The framework demonstrates a structural 2.9x to 3.9x reduction in token consumption, making it highly practical for real-world automation. Across diverse web tasks, GA exhibits remarkable and consistent token efficiency. This substantial reduction in token consumption is not merely an economic benefit; it directly translates to lower latency and higher reliability, which are critical constraints for deployable web agents. This combination of robust effectiveness and rigorous context control validates GA as a highly practical architecture for real-world, open-ended web automation scenarios. A detailed qualitative visualization of GA's web execution trajectory is provided in Appendix 2.

5 Discussion

We highlight key findings from the development of GenericAgent that we believe are broadly relevant to the design of long-horizon LLM agent systems. These observations are distilled from practice rather than derived from controlled ablations alone, and we hope they will inform and inspire future research.

Context information density is a structural constraint for all LLM-based agent systems. Prior work has established that LLM reasoning quality does not scale linearly with context length. Information positioned in the middle of the context is harder to retrieve [10], irrelevant content actively dilutes the model's attention toward decision-critical evidence [10, 12], and the effective context window is substantially smaller than its nominal size [13]. These are not artifacts of any particular model, but intrinsic properties of current LLM architectures. The implication is direct. As long as an agent uses an LLM as its reasoning engine, the quality of each decision step is ultimately determined within a single forward pass, and no amount of tooling, memory capacity, or workflow complexity can circumvent this constraint. Context information density is therefore not an optional optimization target, but a structural constraint that every agent system must confront by design.

There exists a minimal complete capability set for agent systems. Under the structural constraint of information density, our experience suggests that an agent needs to implement only three capabilities, namely **tool interfacing**, **context management**, and **memory formation**. These three capabilities correspond to the three inevitable stages in the agent task execution pipeline where information density is systematically degraded, and therefore constitute what we propose as the minimal set of capabilities an agent framework must implement. (1) Tool interfacing is the sole channel through which an agent interacts with the external world. Redundant tool definitions consume large portions of the context budget before task execution begins, so interface design must be deliberately constrained to keep overhead minimal. (2) Context management corresponds to the input to the language model. Task state, intermediate results, tool outputs, and all other content must be actively filtered before entering the context, as loading everything by default allows irrelevant information to crowd out critical evidence. (3) Memory formation corresponds to cross-task knowledge accumulation. Without retaining verified content from interaction into reusable memory, every task begins from scratch. Any additional complexity that does not serve one of these three capabilities is, in our view, actively degrading information density.

In agent systems, lower token consumption corresponds to better task performance. This finding is

counterintuitive, since the prevailing assumption is that longer reasoning chains and more interaction turns reflect more thorough deliberation, and should therefore yield better outcomes. However, our experimental results systematically point to the opposite conclusion within the setting of long-horizon agentic execution. On Lifelong AgentBench, GA consumes only 27.7% of Claude Code's input tokens and 15.5% of OpenClaw's, while achieving a higher task completion rate of 100%. This pattern holds consistently across multiple benchmarks. As discussed above, beyond a certain point, additional tokens do not introduce more useful information but instead degrade reasoning quality through positional bias, attention dilution, and effective window contraction [10, 12, 13]. An agent that consumes more tokens is more likely suffering from systematic failures in context management, compensating for degraded per-step decision quality through additional interactions rather than improving it. We therefore propose that, in long-horizon agentic settings, token consumption reflects the symptoms of an agent's context management quality, rather than the thoroughness of reasoning.

Permissions define the ceiling of agent capability. What an agent can perceive, what it can act upon, and what feedback it can learn from directly determine the complexity of reasoning chains it can develop and the difficulty of tasks it can solve. The scope of permissions is the boundary of the capability development space, which cannot be decoupled. Locking down the action boundary during the agent's exploration phase is equivalent to preemptively capping its capability ceiling at the system design stage. An agent restricted to reading a small set of files, unable to execute code or access external information, can only operate within a truncated state space regardless of how capable its underlying model is. Narrowing the exploration boundary is not a path toward building useful agents, the endpoint of which is a system that is safe, but useless.

Minimal architecture is a necessary prerequisite for autonomous agent evolution. In practice, we find that GA's autonomous exploration already consolidates far more skills than manual intervention could accumulate, representing a first step beyond traditional human-in-the-loop paradigms. We conjecture that the implications of architectural minimality extend further still. When the architecture is sufficiently minimal, the target of evolution can expand from skills to the architecture itself. A system with hundreds of thousands of lines of code is opaque to the agent—it can neither understand nor modify it. A core codebase of a few thousand lines, by contrast, is readable, understandable, and modifiable. In GA's minimal architecture, the self-hosted CLI as the native execution surface naturally enables subagents to read and modify the core codebase, making architectural self-update a practically achievable next step. Agent evolution thus presents three progressive dimensions: skill consolidation, autonomous exploration, and architectural self-update. We leave the validation of this full evolutionary path as an open problem for future work, but argue that minimal architecture is its necessary prerequisite.

6 Related Work

5 Prior work on autonomous LLM agents has advanced several ingredients of long-horizon execution, including action interfaces, memory, self-improvement, and web interaction. Our position relative to this literature is more specific than a generic "integrated system" claim. GA is built around a single systems objective, *maximizing contextual information density*, and studies how multiple agent components should be co-designed when the bottleneck is not raw capability but how much decision-relevant information can be preserved within a limited context budget. In this sense, GA is not simply a new memory module, a new reflection method, or a new browser agent; it is a general-purpose agent architecture that connects these ingredients through a common optimization target.

6.1 LLM-Based Agent Systems and Action Interfaces

LLM agents have evolved from prompt-level reasoning loops to integrated systems that execute long action sequences in real environments. ReAct [28] and Reflexion [6] established the basic loop of interleaving reasoning, acting, and feedback. AutoGPT [29] popularized iterative goal decomposition, while MetaGPT [30] pushed the field toward explicit multi-role coordination and workflow design. This line established the key intuition that agent performance depends not only on the base model, but also on how deliberation is scaffolded over multiple steps.

As agents moved into software engineering and computer-use settings, the action interface itself became a first-class design choice. CodeAct [31] unifies agent actions as executable code, which increases compositional flexibility and makes behaviors easier to test and reuse. Devin [32], SWE-agent [33], and OpenHands [34] further show that performance depends strongly on how the model is connected to the external environment, whether through integrated coding workflows, specialized Agent–Computer Interfaces, or open agent runtimes. Recent product systems such as Claude Code [1], Codex [35], Manus [36], and OpenClaw [3] reinforce the same trend in practice.

GA is closest to this systems line, but differs in what it optimizes. Most existing agent systems primarily expand reachable behaviors through richer workflows, more specialized tools, or stronger environment integration. By contrast, GA asks how broad computer-use capability can be preserved while keeping the action space and prompt overhead deliberately small. Its minimal atomic toolset is therefore not just an implementation choice, but part of the paper's central claim: cross-domain long-horizon performance can improve when tool abstraction is designed to increase contextual information density rather than to maximize interface richness.

6.2 Memory and Context Management

A second line of work studies how agents retain only behaviorally useful information as trajectories grow. MemGPT [4] treats the context window as a limited working memory and external storage as archival memory, introducing a paging-style view of agent memory. A-MEM [37] instead models memory as a dynamically evolving network of atomic notes and links, enabling richer associative recall. These approaches make long-horizon agents more feasible, but they focus primarily on storage and retrieval.

Recent work increasingly treats context construction itself as a systems problem. LongLLMLingua [38] shows that prompt compression can preserve task-relevant information while reducing long-context cost, and practical context-engineering analyses emphasize that agent quality depends not only on window length but also on what enters the window and in what form [13]. Production systems such as Claude Code [1] and Manus [36] similarly rely on artifact-based tracking and periodic compaction to extend effective horizon.

GA differs from both memory-centric and compression-centric approaches in two ways. First, it treats memory quality as a *verification and selection* problem, not only a storage or retrieval problem: only behavior-changing and validated information is promoted into longer-term representations. Second, it optimizes the full path from observation to retained memory, rather than only the final retrieval step. The resulting design goal is not to keep more history accessible, but to keep the active context as sparse, reliable, and decision-relevant as possible.

6.3 Self-Evolution and Experience Distillation

Research on self-evolving agents asks how repeated execution can become future capability. A recent survey [39] organizes the space by what evolves, when evolution happens, and what feedback drives improvement. Within this space, Agent-Pro [40] studies policy-level reflection and optimization, showing that agents can revise their own behavioral policies without updating model parameters. Voyager [8] demonstrates a stronger form of accumulation in a specialized environment by continually storing verified executable skills.

Broader general-purpose work mostly evolves the agent through textual abstractions. EvolveR [41], FLEX [42], AgentEvolver [43], and experience-driven lifelong learning [44] all convert trajectories into strategic principles, reflections, or structured knowledge units that help later execution. This is an important step beyond simple history reuse, but in most cases the retained experience remains natural-language guidance rather than executable capability.

GA is aligned with this line in viewing reflection as the engine of continual improvement, but it makes a stronger systems claim: the endpoint of reflection should be a *representation shift* from verbose trajectories to compact operational assets. Verified experience is progressively transformed into SOPs, code, and reusable skills, so improvement appears not only as better policy guidance but also as lower inference-time cost. This is why our evaluation emphasizes efficiency convergence and token reduction under repeated execution, rather than only one-shot task success.

7 Conclusion

We present **GenericAgent (GA)**, a self-evolving general-purpose LLM agent built around a single design principle: **context information density maximization**. Instead of treating context as a passive byproduct of interaction, GA optimizes both completeness and conciseness. It achieves this through four components: a minimal atomic tool set, a hierarchical on-demand memory, a reflection-driven self-evolution pipeline that distills verified trajectories into reusable SOPs and executable code, and a context truncation and compression layer that preserves information density during long execution.

Across five evaluation dimensions, GA shows a strong efficiency–performance trade-off. It matches or outperforms existing agent systems on task completion while using fewer tokens and interactions. Its self-evolution process also reduces token usage by up to 89.6% across repeated runs, while maintaining or improving performance. Overall, these results suggest that improving long-horizon agent capability is not mainly about adding more tools, memory, or longer context. It is more about controlling how information is represented and maintained during execution. We release GA as an open-source system and hope this work can inform future research on general-purpose self-evolving agents.

References

[1] Anthropic. Claude code. https://www.anthropic.com/product/claude-code, 2025.

[2] OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, 2025. Published May 16, 2025. Accessed: 2026-04-17.

[3] Xingyao Wang et al. Openclaw-rl: Training agents via interaction. *arXiv preprint arXiv:2603.10165*, 2026.

[4] Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

[5] Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *arXiv preprint arXiv:2512.22087*, 2025.

[6] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[7] Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. *arXiv preprint arXiv:2308.10144*, 2023.

[8] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[9] Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. Lifelongagentbench: Evaluating llm agents as lifelong learners. *arXiv preprint arXiv:2505.11942*, 2025.

[10] Lost in the middle: How language models use long contexts. https://arxiv.org/abs/2307.03172, 2023.

[11] Chenxin An, Jun Zhang, Ming Zhong, Lei Li, Shansan Gong, Yao Luo, Jingjing Xu, and Lingpeng Kong. Why does the effective context length of llms fall short? In *The Thirteenth International Conference on Learning Representations*, 2025. ICLR 2025 Poster.

[12] Llms get lost in multi-turn conversation. https://arxiv.org/abs/2505.06120, 2025.

[13] Aditya Rajasekaran et al. Effective context engineering for ai agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, September 2025.

[14] Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

[15] Hongru Wang, Wenyu Huang, Yufei Wang, Yuanhao Xi, Jianqiao Lu, Huan Zhang, Nan Hu, Zeming Liu, Jeff Z. Pan, and Kam-Fai Wong. Rethinking stateful tool use in multi-turn dialogues: Benchmarks and challenges. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 5433–5453, Vienna, Austria, 2025.

[16] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

[17] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. Memorybank: Enhancing large language models with long-term memory. *arXiv preprint arXiv:2305.10250*, 2023.

[18] Zhen Tan, Jun Yan, I-Hung Hsu, Rujun Han, Zifeng Wang, Long Le, Yiwen Song, Yanfei Chen, Hamid Palangi, George Lee, Anand Rajan Iyer, Tianlong Chen, Huan Liu, Chen-Yu Lee, and Tomas Pfister. In prospect and retrospect: Reflective memory management for long-term personalized dialogue agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8416–8439, Vienna, Austria, 2025.

[19] Chunliang Chen, Ming Guan, Xiao Lin, Jiaxu Li, Qiyi Wang, Xiangyu Chen, Jixiang Luo, Changzhi Sun, Dell Zhang, and Xuelong Li. Telemem: Building long-term and multimodal memory for agentic ai. *arXiv preprint arXiv:2601.06037*, 2026.

[20] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023.

[21] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context, 2023.

[22] Chenxin An, Jun Zhang, Ming Zhong, Lei Li, Shansan Gong, Yao Luo, Jingjing Xu, and Lingpeng Kong. Why does the effective context length of llms fall short?, 2024.

[23] Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Prince Grover, Shivam Asija, Giuseppe Carenini, Wei Zhang, Arushi Gupta, Sreyoshi Bhaduri, Jing Xu, Huzefa Raja, Shayan Ray, Aaron Chan, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, Harshita Asnani, Weian Chan, Ming Xiong, Francesco Carbone, and Jeetu Mirchandani. Sop-bench: Complex industrial sops for evaluating llm agents. *arXiv preprint arXiv:2506.08119*, 2025.

[24] Yuyang Dai, Yan Lin, Zhuohan Xie, and Yuxia Wang. Realfin: How well do llms reason about finance when users leave things unsaid? *arXiv preprint arXiv:2602.07096*, 2026.

[25] Adyasha Maharana, Dong-Ho Lee, Sergey Tulyakov, Mohit Bansal, Francesco Barbieri, and Yuwei Fang. Evaluating very long-term conversational memory of llm agents. *arXiv preprint arXiv:2402.17753*, 2024.

[26] Yichen Pan, Dehan Kong, Sida Zhou, Cheng Cui, Yifei Leng, Bing Jiang, Hangyu Liu, Yanyi Shang, Shuyan Zhou, Tongshuang Wu, and Zhengyang Wu. Webcanvas: Benchmarking web agents in online environments. *arXiv preprint arXiv:2406.12373*, 2024.

[27] Peilin Zhou, Bruce Leon, Xiang Ying, Can Zhang, Yifan Shao, Qichen Ye, Dading Chong, Zhiling Jin, Chenxuan Xie, Meng Cao, Yuxin Gu, Sixin Hong, Jing Ren, Jian Chen, Chao Liu, and Yining Hua. Browsecomp-zh: Benchmarking web browsing ability of large language models in chinese. *arXiv preprint arXiv:2504.19314*, 2025.

[28] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

[29] Significant Gravitas. Autogpt. https://github.com/Significant-Gravitas/AutoGPT, 2023.

[30] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

[31] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024. Accepted at ICML 2024.

[32] Cognition Labs. Devin: Ai software engineer. https://cognition.ai/blog/introducing-devin, 2024.

[33] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao Narasimhan, and Karthik. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37, 2024.

[34] Xingyao Wang, Boxuan Jiang, Ziniu Lu, Yufan Liu, Abishek Sridhar Li, Bolun Shi, Jiannan Fang, Rithvik Mohanty, Niklas Muennighoff Ponnapalli, Kaixuan Ren, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, 2025.

[35] OpenAI. Introducing codex. https://openai.com/index/introducing-codex/, 2025.

[36] Yichao Shen et al. Manus: From mind to machine. *arXiv preprint arXiv:2505.02024*, 2025.

[37] Wujiang Xu et al. A-mem: Agentic memory for llm agents. *Advances in Neural Information Processing Systems*, 38, 2025.

[38] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024.

[39] Zhiyuan Gao et al. A survey of self-evolving agents. *Transactions on Machine Learning Research*, 2026.

[40] Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-pro: Learning to evolve via policy-level reflection and optimization. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5348–5375, 2024.

[41] Xuechen Wu et al. Evolver: Self-evolving llm agents through experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*, 2025.

[42] Anonymous. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025.

[43] Anonymous. Agentevolver: Towards efficient self-evolving agent system. *arXiv preprint arXiv:2511.10395*, 2025.

[44] Tianyu Cai et al. Building self-evolving agents via experience-driven lifelong learning. *arXiv preprint arXiv:2508.19005*, 2025.

8 Author Contributions

This project is led by **Jiaqing Liang** and **Yanghua Xiao**. The development of the GA and the preparation of this manuscript involve contributions from multiple authors across different aspects, including system design, experimentation, and writing. Below we detail the specific contributions of each author.

Jiaqing Liang: Assistant Professor of Fudan University and AI chief scientist of Shenzhen Aquaintelling Technology. **Led core system design, implemented the main codebase, and was responsible for project management**.

Jinyi Han: Led the manuscript design and writing process. Defined the paper structure, coordinated contributions across authors, and contributed to drafting, integrating, and refining the manuscript from the initial draft to the final version.

Weijia Li: Participated in the development of the GA system. Wrote Sections 2.1, 3.3, and 4.5, conducted the evaluation experiments therein, and designed Figures 1 and 2.

Xinyi Wang: Participated in the development of the GA system. Conducted evaluation experiments for Section 4.3 , contributed to the writing of Sections 4.3 and 4.5.

Zhoujia Zhang: Participated in the development of the GA system. Conducted evaluation experiments for Section 4.4.

Zishang Jiang: Wrote Section 4.1, 4.2 and 4.4.

Ying Liao: Participated in the development of the GA system. Conducted evaluation experiments for Section 4.2 and generated figures for Section 4.

Tingyun Li: Wrote Section 6 and the Appendix.

Ying Huang: Participated in the development of the GA system. Conducted evaluation experiments on RealFineBench for Section 4.1.

Hao Shen: Participated in the development of the GA system.

Hanyu Wu: Participated in the development of the GA system. Refined the writing of Section 3.3 and implemented the code for this module.

Fang Guo: Participated in the development of the GA system.

Keyi Wang: Conducted evaluation experiments on SOP-Bench for Section 4.1.

Zhonghua Hong: Conducted evaluation experiments on Lifelong AgentBench for Section 4.1.

Zhiyu Lu: Performed partial evaluation experiments for models in Section 4.1.

Lipeng Ma: Participated in the evaluation experiments in Section 4.1 and contributed to the refinement of Section 4.

Sihang Jiang: Contributed to the refinement of the overall manuscript and wrote Section 5.

Yanghua Xiao: Professor of Fudan University and chief scientist of Shenzhen Aquaintelling Technology. Provided overall project supervision and strategic guidance.

9 Appendix

1 Atomic Tool Alignment

This subsection provides a detailed capability-level alignment for the core atomic tools discussed in the main text. The purpose is not to enumerate the full runtime tool inventories of Claude Code or OpenClaw, but to show that the core capabilities retained by GA each have corresponding prototypes in both systems.

Table 10 Capability-level alignment of the basic tooling environment. Each row maps one capability category to the corresponding tool in Claude Code, OpenClaw, and GA.

Capability Category	Corresponding Tool / Mechanism		
	Claude Code	**OpenClaw**	**GA**
Code execution	BashTool	exec / process / nodes	code_run
File reading	FileReadTool	read	file_read
File writing	FileWriteTool	write	file_write
Local file editing	FileEditTool	write / patch-like editing	file_patch
Web reading	WebFetchTool / browser read	browser base reading	web_scan
Web interaction	WebBrowserTool / browser actions	browser interaction	web_execute_js
Working memory	TodoWriteTool / BriefTool	session history / status-like ability	update_working_checkpoint

Table 11 complements the capability-level mapping above with representative task-level replacement examples. Rather than claiming that GA reproduces every specialized tool one by one, the point is that many benchmarked capabilities can be reconstructed through short compositions of a few atomic tools.

Source	Specialized Tool / Capability	Representative Task	GA Replacement Composition	Argument
Claude Code	GlobTool	task_cc_01_glob_-markdown_list	code_run	File-pattern matching can be completed through general code execution.
Claude Code	GrepTool	task_cc_02_grep_-todo_hits	code_run + file_read	Text search and localization can be implemented by combining general execution with file reading.
Claude Code	WebFetchTool	task_cc_03_webfetch_-example_brief	web_scan + web_execute_js + code_run	Web fetching, cleaning, and summarization can be reconstructed by combining browser primitives with code-based post-processing.
Claude Code	AgentTool	task_cc_04_agent_-fact_extract	file_read + code_run + update_working_-checkpoint	Explicit agent delegation can be approximated through stored subagent strategies in memory together with general execution triggers.
Claude Code	WebSearchTool	task_cc_05_-websearch_example_-results	web_execute_js + web_scan	Online search can be completed through browser interaction plus webpage reading.
Claude Code	NotebookEditTool	task_cc_06_notebook_-inspect	file_read + code_run	Notebook-structure inspection can be handled by reading the .ipynb JSON and executing a parsing script.
OpenClaw	nodes / structured data	task_oc_02_csv_stats	code_run	CSV statistics can be completed through general code execution without a dedicated structured-data tool.

Table 11 Representative task-level substitutions from specialized tools to GA atomic-tool compositions. Each row shows one concrete capability test, the original specialized tool or mechanism used by the baseline system, and the corresponding GA composition that can complete the same class of task.

2 Web Browsing Visualization

Figure 6 provides a visual comparison of token consumption and normalized performance across the three web browsing benchmarks discussed in Section 4.5.

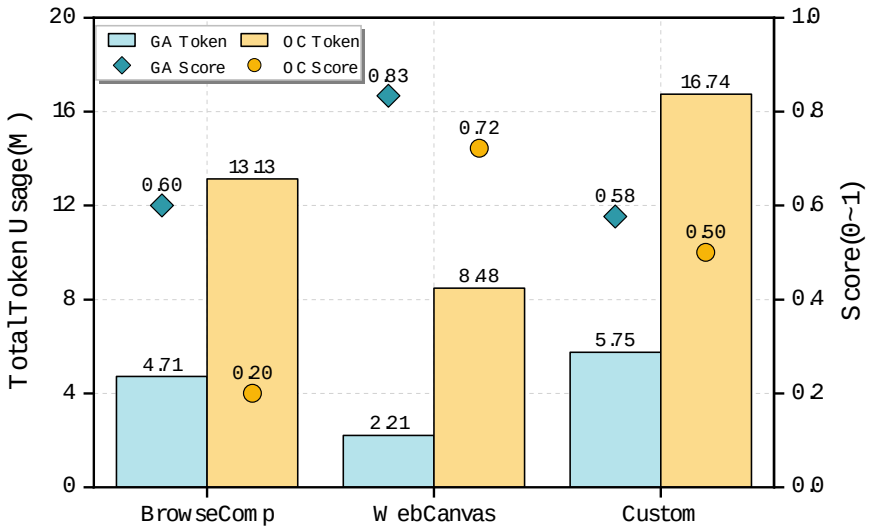

Figure 6 Token consumption and normalized score across three web browsing benchmarks. The left axis shows total token consumption in millions (M), while the right axis shows normalized scores on a 0–1 scale. GA consistently achieves competitive or superior performance while consuming significantly fewer tokens than OpenClaw across all three benchmarks.

3 Case Studies

We present four representative cases corresponding to the main evaluation dimensions. For readability, the cases are ordered as tool use, memory, self-evolution, and web browsing. Each case follows a "purpose and task setup → artifacts / trace → observed outcome" pattern so that the appendix reads as a set of worked examples rather than a list of compressed summaries.

- **Case A1 (Tool Use).** An API procurement workflow requiring web lookup, structured extraction, budget reasoning, and file generation, used to show how GA handles a long-horizon task with a minimal atomic-tool set (Section 4.2).
- **Case A2 (Memory).** Dangerous-goods hazard classification under condensed, full, and redundant memory variants, used to show why information density matters more than information volume (Section 4.3).
- **Case A3 (Self-Evolution).** A GitHub PR research task showing the progression from natural-language SOP to compiled Python code and a transferability note, used to demonstrate how GA converts experience into reusable executable assets (Section 4.4).
- **Case A4 (Web Browsing).** A BrowseComp-ZH multi-hop reasoning chain where GA identifies a historical figure through iterative search refinement, used to illustrate structured browser extraction in practice (Section 4.5).

3.1 Case A1. Tool Use: API Procurement Workflow

Purpose and Task Setup

```
Purpose: to examine whether GA's atomic-tool design can support a long-horizon, multi-step
workflow,
we provide a laboratory procurement task over public LLM pricing pages.
This task is useful because it requires browsing, structured extraction, numerical reasoning,
and file generation in one workflow.
The goal of the case is to show that GA's minimal atomic-tool set can complete the full chain
and reach the correct recommendation, providing a qualitative counterpart to Table 12.
Task: investigate laboratory procurement for text-only LLM APIs using these public pricing
pages:
- OpenAI API pricing: https://openai.com/api/pricing/
- Anthropic pricing: https://www.anthropic.com/pricing
- Gemini API pricing: https://ai.google.dev/gemini-api/docs/pricing
Constraints:
- only text models are considered
- laboratory size: 6 students
- monthly input tokens: 80M
- monthly output tokens: 20M
- budget: $300 / month
Requirements:
1. record candidate flagship text models and their input/output prices
2. test whether a single-model plan fits the budget
3. if not, trigger fallback and test a dual-model plan
4. if that still fails, output infeasible
5. final answer must state the primary option, whether it is feasible, whether fallback was
triggered, the recommended plan, the estimated monthly cost, and the reason
Required output files: cost_comparison.csv and decision_04.json.
```

Artifact 1: Cost Comparison File

```
cost_comparison.csv
provider,model,input_cost_per_1m,output_cost_per_1m,estimated_monthly_cost_usd,meets_budget
OpenAI, GPT-5.4, 2.5, 15.0, 500.0, No
OpenAI, GPT-5.4 mini, 0.75, 4.5, 150.0, Yes
OpenAI, GPT-5.4 nano, 0.2, 1.25, 41.0, Yes
Anthropic, Claude Opus 4.6, 5.0, 25.0, 900.0, No
Anthropic, Claude Sonnet 4.6, 3.0, 15.0, 540.0, No
Anthropic, Claude Haiku 4.5, 1.0, 5.0, 180.0, Yes
Google, Gemini 2.5 Pro, 1.25, 10.0, 300.0, Yes
Google, Gemini 2.5 Flash, 0.3, 2.5, 74.0, Yes
Google, Gemini 2.5 Flash-Lite, 0.1, 0.4, 16.0, Yes
```

Artifact 2: Decision File

```
primary_option: Gemini 2.5 Pro
primary_option_feasible: true
fallback_triggered: false
recommended_plan: single_model
recommended_models: [Gemini 2.5 Pro]
estimated_monthly_cost_usd: 300.0
reason: Gemini 2.5 Pro is a strong single-model option that still fits within the budget.
Cost calculation: 80M input x $1.25/MTok + 20M output x $10.00/MTok = $300.0/month.
Other flagship models exceed the budget: GPT-5.4 at $500/month and Claude Opus 4.6 at
$900/month.
```

Table 12 Cross-system comparison on the long-horizon procurement task.

System	Requests	Total Tokens	Time (s)	Success
GenericAgent	21	364,385	324.64	Yes
Claude Code	29	485,335	224.53	Yes
OpenClaw	12	428,402	108.42	Yes

```
No fallback to a dual-model plan is needed.
```

Observed Outcome

```
GA completed the task in 21 requests and 364,385 total tokens (324.6s).
All grading criteria passed: cost_comparison.csv and decision_04.json both created with
correct schema.
Final recommendation: Gemini 2.5 Pro as a single-model plan at exactly $300/month.
Cost breakdown: 80M input × $1.25/MTok + 20M output × $10.00/MTok = $300.0.
Other flagship models exceeded the budget: GPT-5.4 at $500/month, Claude Opus 4.6 at
$900/month.
No fallback to a dual-model plan was needed.
```

3.2 Case A2. Memory: Dangerous-Goods Hazard Classification

Purpose and Task Setup

```
Purpose: to examine whether memory quality depends more on information density than on raw
information volume,
we provide a dangerous-goods classification task under three memory variants.
This task is useful because the decision rule stays fixed while only the memory
representation changes.
The goal of the case is to show that the main effect comes from how the rules are packaged
and retrieved, rather than from whether the rules exist at all.
Task: determine hazard_class from product_id and the four source texts.
Inputs:
- SDS label text
- handling/storage guidance
- transportation requirements
- disposal guidance
The output must contain both hazard_score and hazard_class.
```

Artifact 1: Condensed Memory

```
Required rules:
1. First validate product_id. It must be resolvable and follow format P_XXXXX.
If invalid, stop and output hazard_score=0 and hazard_class="Unable to Decide".
2. Get four component scores:
- safety score from SDS label text
- handling score from handling/storage guidelines
- transportation score from transportation requirements
- disposal score from disposal guidelines
Each valid component score must be in [1,5].
3. Compute hazard_score:
- If one component is missing or 0, replace it with the max of the other available scores.
- If more than two components are missing or 0, output hazard_score=0 and hazard_-
```

class="Unable to Decide".
- Otherwise hazard_score = sum of the four component scores.
- Final valid range is 4 to 20.
4. Map hazard_score to class:
- 4-7 to Hazard Class A
- 8-12 to Hazard Class B
- 13-16 to Hazard Class C
- 17-20 to Hazard Class D
5. Final output must include both hazard_score and hazard_class.

Artifact 2: Full Memory

1. Purpose
To establish a standardized methodology for the systematic identification and classification of dangerous goods hazard classes through multi-source data integration and quantitative severity assessment protocols.
2. Scope
This procedure encompasses all dangerous goods shipment classification processes within the organization's supply chain operations.
3. Definitions
SDS = Safety Data Sheet
HS = Handling and Storage Guidelines
TR = Transportation Requirements
DG = Disposal Guidelines
AIP = API Integration Protocol
HCM = Hazard Classification Matrix
SAS = Severity Assessment Score
4. Input
Product ID (format: P_XXXXX)
Source documentation: SDS / Handling and Storage / Transportation / Disposal
API access credentials: endpoint URLs / authentication tokens / backup authentication protocols
5. Main Procedure
Validate product identification documentation completeness.
If product ID fails format requirements, no further action should be taken.
The hazard score should be marked as 0 and hazard class as Unable to Decide.
Extract four scores, each between 1 and 5.
hazard_score = safety score + handling score + transportation score + disposal score
If any component is missing or 0, impute it by taking the max of the other scores.
If more than two component scores are missing, mark Unable to Decide.
6. Output
Final hazard class designation
Digital record in Hazard Classification Registry
API response logs for all scoring calculations
Classification audit trail documentation
Final output should be in XML format with tags <hazard_score> and <hazard_class>

Artifact 3: Redundant Memory

You are following a standardized dangerous-goods classification procedure designed for supply-chain compliance and multi-source hazard assessment.
Definitions:
- SDS = safety data sheet
- HS = handling and storage guidance
- TR = transportation requirements
- DG = disposal guidelines

```
- SAS = severity assessment score from 1 to 5
- HCM = hazard classification matrix
Input scope: product_id, SDS text, handling/storage text, transportation text, disposal
text, API access and validation context.
Decision workflow:
1. Validate documentation completeness and product identifier validity.
2. product_id must match format P_XXXXX and be resolvable.
3. If invalid, do not continue. Set hazard_score to 0 and hazard_class to "Unable to
Decide".
4. Analyze SDS, handling/storage, transportation, and disposal sources separately.
5. Derive one component score per source, each between 1 and 5.
6. If exactly one component is missing or 0, impute it using the maximum of the other
component scores.
7. If more than two components are missing, return "Unable to Decide" and hazard_score 0.
8. Sum the four components into a cumulative hazard_score and validate that total score lies
in the range 4-20.
9. Convert the total score into Hazard Class A / B / C / D.
10. Return the final result in structured form with hazard_score and hazard_class,
preserving an audit-ready trail.
Background note: the broader SOP also mentions registry logging, API integration,
source-document handling, and record retention, but the operationally decisive rules are
the identifier check, missing-value handling, score summation, and class mapping.
```

Observed Outcome

The memory ablation becomes visually intuitive once the three variants are shown side by side. The condensed file places every behavior-changing rule in a single short block, whereas the full file spreads the same operational core across purpose statements, definitions, credential notes, registry requirements, and output-format details. Table 5 can therefore be read primarily as an information-placement result rather than a simple information-volume result.

3.3 Case A3. Self-Evolution: GitHub PR Research

Purpose and Task Setup

```
Purpose: to examine whether repeated execution can be distilled into reusable assets,
we provide a GitHub PR research task that evolves from an SOP into executable code.
This task is useful because the representation shift is directly observable: the agent first
stabilizes a procedure as text and then compiles that procedure into code.
The goal of the case is to show that what changes across iterations is not merely token
count, but the form of reusable knowledge, which serves as the qualitative counterpart of
Table 8.
Investigate the five most recent merged bug-fix PRs in a GitHub repository.
For each PR: recover the linked issue, identify the affected module, and verify whether the
module has troubleshooting coverage in the documentation site.
Produce a structured JSON report.
```

Artifact 1: SOP

```
Scenario: batch investigation of GitHub PRs, including PR details, linked issues, affected
modules, and documentation coverage.
Core strategy:
DO NOT browse PR pages one by one - efficiency is extremely low and the run may exceed the
turn budget.
PREFER batch extraction - use a script or API to obtain multiple PR records at once.
```

```
Recommended solution:  use github_pr_analyzer.py to complete the whole chain in one pass.
Example command:
python3 ../memory/github_pr_analyzer.py langchain-ai/langchain
-doc-url https://python.langchain.com
-limit 5
-output report.json
Script advantages:
- pure Python, no browser environment required
- one-click execution of the full workflow
- flexible command-line parameters
- built-in error handling
Manual fallback flow:
1.  build a filtered PR URL
2.  use document.querySelectorAll(.js-issue-row) to extract PR numbers and titles
3.  batch fetch PR HTML
4.  recover issue links with the rule /issues/\d+, then Fixes/Closes #\d+, then standalone
#\d+
5.  infer module names from PR title prefixes such as community:
6.  verify documentation coverage against troubleshooting and integration paths
7.  dump the final report with json.dump()
Pitfalls:
1.  do not use window.location.href to visit PR pages one by one
2.  prefer browser fetch() when manual HTML collection is needed
3.  distinguish issue links from pull-request links
4.  different projects may use different documentation subdomains
```

Artifact 2: Compiled Code

fetch_pr_list(...)

```
url = https://github.com/{repo}/pulls?q={filters}
response = session.get(url, timeout=30)
soup = BeautifulSoup(response.text, 'html.parser')
pr_elements = soup.select('.js-issue-row')
collect number, title, and url for the first limit PRs
```

extract_pr_details(pr)

```
module_match = re.match(r'^([^:\[]+)', pr['title'])
pattern 1:  full GitHub issue URL
pattern 2:  Fixes|Closes|Resolves #N
pattern 3:  standalone #N excluding the PR number itself
return pr_number, bug_module, issue_link
```

check_doc_coverage(module)

```
probe multiple candidate paths:
/docs/troubleshooting/
/docs/troubleshooting/{module}
/docs/integrations/{module}/
/docs/modules/{module}/
/api_reference/{module}/
/docs/integrations/providers/{module}/
accept the module if the page contains both the module name and troubleshooting
```

analyze(...)

```
fetch PR list, then extract details, then check documentation, then call json.dump(results,
f, indent=2, ensure_ascii=False)
```

Artifact 3: Transferability Note

```
Original task core abilities:
- batch retrieval of GitHub PR lists with filters
- extraction of PR details, titles, modules, and linked issues
- verification of external documentation coverage
- generation of structured JSON reports
Reusable asset set:
- SOP document: github_pr_research_sop.md
- Python script: github_pr_analyzer.py
Transfer targets described in the case note:
1. GitHub issue batch research
2. contributor analysis
3. release tracking
4. workflow and CI inspection
5. code search and pattern extraction
Example reuse in the issue-analysis case:
- keep the batch-fetch strategy
- replace /repos/{repo}/pulls with /repos/{repo}/issues
- reuse regex extraction and concurrent request handling
Estimated adaptation cost in the transferability note: about 30% code changes, mainly
endpoint replacement and field mapping.
```

Observed Outcome

```
LangChain verification case (2026-04-08):
python3 github_pr_analyzer.py langchain-ai/langchain -doc-url https://python.langchain.com
-limit 5

5 merged bug-fix PRs retrieved
modules found: community (2), partners, openai, core/anthropic
issue links recovered: 4/5
troubleshooting coverage found: 4/5
runtime: about 30-40 seconds
```

3.4 Case A4. Web Browsing: BrowseComp-ZH Reasoning Chain

Purpose and Task Setup

Purpose: to examine multi-hop web browsing under a compressed browser interface,
we provide a BrowseComp-ZH question whose answer must be recovered through iterative query refinement.
This example is useful because the answer is not available from a single lookup and the chain of evidence must be assembled step by step.
The goal of the case is to make the narrowing process visible: identify the *shihua* pioneer, confirm Ouyang Xiu, and then recover Wang Anshi from the quotation.
Original question (BrowseComp-ZH):
"There is a writer A who participated in a literary reform movement,
advocated a plain but disciplined style, opposed overly strange and risky writing,
and pioneered the *shihua* form of poetic criticism.
Another contemporary writer B commented that he had "deep character" and "broad knowledge".
Who was writer B?"
Recorded case metadata: dataset = browsecomp, score = 1.0, time = 130.4s, tokens = input 30050 / output 1296, turns = 7, tools = 6.

Artifact 1: Interaction Trace

```
Turn 1 summary: start by searching "first shihua writer" to identify writer A.
Action query: shihua inventor plain style
Turn 2 summary: scan the Google result page and narrow the candidate set.
Turn 3 summary: because the first search does not uniquely resolve the target, search
"Ouyang Xiu + Liuyi Shihua".
Action query: Ouyang Xiu Liuyi Shihua
Turn 4 summary: confirm writer A as Ouyang Xiu.
Turn 5 summary: search the evaluative phrase together with Ouyang Xiu.
Action query: deep character broad knowledge Ouyang Xiu
Turn 6 summary: verify that the quoted evaluator is Wang Anshi.
Turn 7 summary: return the final answer with confidence.
```

Artifact 2: Final Reasoning Excerpt

```
Explanation:
Through the searches on "first shihua writer" and related keywords, the agent identified
writer A as Ouyang Xiu.
He participated in the Northern Song literary reform movement, advocated a plain but
disciplined style,
opposed strange and risky writing, and pioneered the form through Liuyi Shihua.
After searching the evaluative phrase together with Ouyang Xiu,
the agent found that Wang Anshi, in a commemorative text for Ouyang Xiu,
described him as having "deep character" and "broad knowledge".
Therefore the writer B in the question is Wang Anshi.
```

Observed Outcome

```
score: 1.0
turns: 7
tool calls: 6
time: 130.4s
Exact Answer: Wang Anshi
Judge result: correct = yes, confidence = 95%
Reasoning note: the extracted final answer exactly matches the gold answer "Wang Anshi".
```

4 General Capability Showcase

Beyond the four evaluation-aligned cases above, we present five additional cases drawn from a pool of more than 500 historical sessions. They are selected to maximize coverage of non-benchmark capability dimensions rather than to reflect average-case performance, and are intended as qualitative demonstrations rather than statistical evaluation. All cases come from user-authorized sessions; sensitive identities, document contents, account details, and message contents are anonymized or paraphrased when presented here, and any autonomous behaviors were executed under user-configured triggers or preset operating policies. Together, these cases illustrate how GA's architectural choices — atomic tools, persistent memory, and self-evolution — support real-world workflows that are difficult to capture in standardized benchmarks. Each case follows the same "purpose and task setup → execution trace → observed outcome" format for consistency.

- **Case B1 (Cross-Device Control).** A mobile food-ordering task via ADB, followed by screen-recording retrieval and video post-processing, used to show how atomic-tool composition can bridge the PC–smartphone boundary within one execution chain.
- **Case B2 (Cross-Platform Orchestration).** A message-forwarding task from a local WeChat database to a Weibo post, used to show how GA composes heterogeneous local and web environments within a single workflow.

- **Case B3 (Autonomous Operation).** An overnight session triggered after the user leaves, used to show how self-evolution, working checkpoints, and preset patrol routines can support bounded autonomous operation.
- **Case B4 (Remote Infrastructure).** An SSH-based remote file-server deployment task, used to show how atomic tools can support end-to-end DevOps workflows including dependency installation, file transfer, service deployment, and iterative troubleshooting.
- **Case B5 (Long-Horizon Academic Workflow).** A multi-session NSFC grant proposal assistance task spanning figure design, citation verification, and batch error correction, used to show how persistent memory supports extended, multi-phase academic work.

4.1 Case B1. Cross-Device Control: Mobile Food Ordering via ADB

Purpose and Task Setup

```
Purpose:  to examine whether GA can operate across the PC-smartphone boundary through
ADB-based device control,
we provide a real-world mobile food-ordering task followed by screen-recording retrieval and
video post-processing.
This task is useful because it requires coordinated control of a physical mobile device, GUI
interaction with a commercial app, and multimedia processing on the host PC within a single
workflow.
The goal of the case is to show that a small atomic-tool set can be composed into a
cross-device execution chain under a realistic consumer-app workflow.

User instruction (verbatim, translated):
"Order two cups of milk tea on my phone via Meituan Waimai.  Stop at the final payment page --
do not pay."

Follow-up instructions:
1.  "Pull the screen recording from the phone."
2.  "Trim everything before 1:10, black out the top half after 3:40."
3.  "Convert it into a sped-up GIF."
```

Artifact 1: Mobile Ordering Execution Trace

```
Step 1:  Establish ADB connection to the Android device.
Step 2:  Launch the Meituan Waimai (food delivery) app via ADB shell am start.
Step 3:  Dismiss pop-up advertisements by locating and tapping the close button coordinates.
Step 4:  Navigate to the "Desserts & Drinks" category.
Step 5:  Select a nearby store (Hushang Ayi, a popular milk-tea chain).
Step 6:  Locate the "Must-Try Milk Tea -- Pick Any Two" combo at ¥23.9.
Step 7:  Select flavor 1:  Thick Taro Paste; select flavor 2:  Brown Sugar Boba.
Step 8:  Confirm selections and add to cart.
Step 9:  Proceed to the checkout page and halt -- no payment action taken.
```

Artifact 2: Video Post-Processing Trace

```
Step 1:  Pull the screen recording from the phone via adb pull.
Step 2:  Trim the first 1 minute 10 seconds using ffmpeg -ss 00:01:10.
Step 3:  Black out the top half of the frame after the 3:40 mark using ffmpeg drawbox filter
with overlay logic.
Step 4:  Re-encode and speed up the video (4x) to produce a compact GIF using ffmpeg with
palettegen and paletteuse filters.
Step 5:  Verify the final GIF file size and frame count.
```

Observed Outcome

In this session, GA completed the full pipeline: ADB device connection, commercial app navigation (7 GUI interactions), screen-recording retrieval, video trimming, region-specific masking, and GIF generation.
The workflow was executed through a composition of code_run (ADB commands, ffmpeg), file_read / file_write (media files), and update_working_checkpoint (tracking multi-phase progress).
No task-specific mobile-automation framework (e.g., Appium, UIAutomator) was introduced in this workflow.
This case specifically illustrates the paper's claim that a compact atomic-tool set can be composed into a long execution chain that extends beyond the host PC when ADB is available as a bridge.

4.2 Case B2. Cross-Platform Orchestration: WeChat to Weibo Message Forwarding

Purpose and Task Setup

Purpose: to examine whether GA can orchestrate data flow across heterogeneous local and web platforms,
we provide a task that requires reading from a local encrypted database and publishing to a social-media platform via browser automation.
This task is useful because it chains together local database decryption, contact resolution, message extraction, and browser-based content publishing -- capabilities that belong to different software stacks.
The goal of the case is to show that GA can compose heterogeneous data sources and output channels into a single workflow under explicit user instruction.

User instruction (verbatim, translated):
"Forward the latest message from Professor Xiao in WeChat to Weibo."

Presentation note: contact identity and forwarded message content are anonymized here, and the operation was performed only after the user explicitly requested the forwarding action.

Artifact 1: Execution Trace

Phase 1 -- WeChat Message Extraction:
Step 1: Locate the local WeChat database files (EnMicroMsg.db).
Step 2: Decrypt the database using SQLCipher with the derived key.
Step 3: Query the contact table to resolve the anonymized alias "Professor Xiao" to the intended contact record.
Step 4: Query the message table for the most recent private message from this contact.
Step 5: Extract the latest message content (paraphrased here for privacy).

Phase 2 -- Weibo Publishing:
Step 6: Open the Weibo compose page in the browser via web_scan.
Step 7: Inject JavaScript to locate the post editor textarea and fill in the extracted message.
Step 8: Trigger the submit button via web_execute_js.
Step 9: Verify that the post appears on the user's Weibo timeline.

Observed Outcome

In this session, GA bridged two heterogeneous platforms within one workflow:
(1) local encrypted WeChat database → SQLCipher decryption → contact resolution → message extraction;
(2) browser-based Weibo → JS injection → post creation → publication verification.
The tools used were code_run (SQLCipher operations, key derivation), file_read (database file access), and web_scan + web_execute_js (Weibo interaction).

This case specifically illustrates the paper's claim that atomic tools can be composed across heterogeneous local and web environments, allowing one session to move from private local data handling to browser-side execution under explicit user intent.

4.3 Case B3. Autonomous Operation: Unsupervised Overnight Session

Purpose and Task Setup

Purpose: to examine whether GA can operate productively with limited oversight under a preset autonomy policy,
we present an autonomous session triggered by the system detecting that the user has been away for over 30 minutes.
This task is useful because it demonstrates self-directed task selection, execution, and self-correction over an extended period.
The goal of the case is to show that GA's self-evolution capability can extend beyond benchmark settings into bounded autonomous operation.

Trigger condition:
[AUTO] User has been away for more than 30 minutes.
Operational boundary: the session was restricted to inspection, reporting, tool creation, and reversible local maintenance actions; external publication, payment, and destructive system modification were excluded by policy.
No explicit task instruction was given after the trigger; GA selected and executed tasks based on its built-in patrol and self-improvement routines within the preset policy.

Artifact 1: Autonomous Task Inventory (Rounds 125–140+)

Category 1 -- System Security Audit:
- Scanned all listening ports on the host machine.
- Discovered a stock-trading application occupying 939 UDP ports and 509 MB of memory at 3:00 AM.
- Audited startup items: 11 registry entries + 267 scheduled tasks + 86 auto-start services.

Category 2 -- Tool Creation:
- Wrote port_monitor.py: real-time port monitoring utility.
- Wrote process_watchdog.py: process resource watchdog with alerting.
- Wrote startup_auditor.py: startup-item auditing and reporting tool.

Category 3 -- Web Patrol:
- Visited tech community sites (Guohe Boke, V2EX) and GitHub Trending.
- Generated a summarized patrol report covering trending topics and notable repositories.

Category 4 -- Environment Hygiene:
- Audited the Python environment: found 294 installed packages occupying 1.9 GB.
- Identified 2 CVE vulnerabilities in the filelock package; wrote pip_audit.py.
- Discovered and fixed an inconsistency in its own history filenames (.md vs .txt extension mismatch).

Observed Outcome

Over approximately 15+ autonomous rounds, GA performed system security auditing, created three reusable utility scripts, conducted web patrol with report generation, audited the Python environment for vulnerabilities, and self-corrected its own file-management errors.
After the trigger fired, the session proceeded without additional user input and remained within the preset autonomy policy described above.
Tools used: code_run (system commands, script creation and testing), file_read / file_write (script files, reports), web_scan (web patrol), and update_working_checkpoint (tracking

progress across rounds).
This case specifically illustrates the paper's claim that self-evolution, working checkpoints, and patrol routines can support bounded autonomous operation over extended periods, rather than only repeated benchmark-style tasks.

4.4 Case B4. Remote Infrastructure: SSH-Based File Server Deployment

Purpose and Task Setup

Purpose: to examine whether GA can manage end-to-end remote infrastructure tasks, we provide a real-world deployment scenario requiring SSH access, file transfer, and iterative service configuration.
This task is useful because it involves dependency management on an unfamiliar remote machine, multi-step troubleshooting, and responding to changing user requirements mid-task.
The goal of the case is to show that an atomic-tool workflow can support a complete DevOps-style execution chain in this setting, without relying on infrastructure-as-code tooling.

User instruction (verbatim, translated):
"SSH into user@*[IP address]*. Set up a file server on it and upload the zip file."
Follow-up instruction: "Make it publicly accessible."

Artifact 1: Execution Trace

Phase 1 -- Environment Setup:
Step 1: Install the paramiko library for SSH connectivity (pip install paramiko).
Step 2: Establish an SSH connection to the remote Linux server.
Step 3: Upload the 19 MB zip file via SFTP.

Phase 2 -- Service Deployment:
Step 4: Deploy a Python HTTP file server on the remote machine (python3 -m http.server).
Step 5: Encounter and fix a Chinese filename encoding issue (UTF-8 locale configuration).

Phase 3 -- Requirement Change:
Step 6: User requests public access → reconfigure the server to remove authentication.
Step 7: Add directory isolation so that only the target folder is exposed.

Phase 4 -- Verification:
Step 8: Test the download from the local machine.
Step 9: Confirm file integrity: downloaded size = 19.01 MB, matching the original.

Observed Outcome

In this session, GA completed the full DevOps workflow: SSH connection, dependency installation on the remote machine, file upload (19 MB), HTTP server deployment, encoding bugfix, mid-task requirement change (public access), directory isolation, and end-to-end verification.
Tools used: code_run (paramiko SSH, SFTP, remote shell commands), file_read / file_write (local zip file, configuration), and update_working_checkpoint (phase tracking).
This case specifically illustrates the paper's claim that atomic tools plus lightweight state tracking can support a complete "connect → configure → deploy → troubleshoot → verify" workflow over SSH in a realistic remote environment.

4.5 Case B5. Long-Horizon Academic Workflow: NSFC Grant Proposal Assistance

Purpose and Task Setup

Purpose: to examine whether GA can sustain coherent support over a long-horizon, multi-session academic task,
we present a real-world NSFC (National Natural Science Foundation of China) grant proposal assistance case that spanned multiple sessions over several days.
This task is useful because it involves reading and comprehending a full-length proposal, identifying structural weaknesses, producing academic figures, performing large-scale citation verification, and iteratively correcting errors -- capabilities that must be coordinated across sessions with persistent memory.
The goal of the case is to show that GA's memory system and self-evolution capabilities can support sustained, multi-phase academic assistance beyond a single session.

Presentation note: the proposal materials were user-provided, and document-specific details are anonymized here when they are not essential to the technical point.
Task phases (emerged organically from user interaction):
1. Read and analyze the complete NSFC proposal document.
2. Identify that the proposal contains no figures; design a figure plan.
3. Generate academic figures (overview diagram + per-section illustrations).
4. Verify all bibliographic entries against external sources.
5. Correct discovered citation errors and regenerate the PDF.

Artifact 1: Figure Design and Generation

Observation: the entire proposal contained zero figures, which is a significant weakness for an NSFC application.
Action plan:
- Design a research overview figure showing the relationship among the three proposed research themes.
- Design per-section illustrations for each research content block.

Execution:
- Used Python (matplotlib, networkx) to generate vector-format overview diagrams.
- Composed Gemini API prompts to generate domain-appropriate academic illustrations.
- Integrated figures into the LaTeX source and verified rendering.

Artifact 2: Citation Verification and Batch Correction

Approach: parsed the .bib file and extracted all bibliographic entries.
For each entry:
- Queried arXiv API to verify title, authors, and year.
- Cross-referenced with OpenAlex API for published-venue entries.
- Compared extracted metadata against the .bib fields.

Findings:
- Discovered a significant number of arXiv preprint entries with incorrect metadata (wrong year, mismatched titles, incomplete author lists).
- Root cause: likely auto-generated .bib entries from LLM-assisted writing without manual verification.

Correction:
- Batch-updated the .bib file with verified metadata.
- Logged all changes for the user's review (change log with before/after comparisons).
- Regenerated the PDF with corrected references.

Observed Outcome

Across multiple sessions spanning several days, GA provided end-to-end assistance for:
(1) full-document comprehension and structural analysis;
(2) figure design and generation (Python vector graphics + API-generated illustrations);
(3) systematic citation verification against arXiv and OpenAlex, discovering and correcting a batch of erroneous entries;
(4) LaTeX integration and PDF regeneration.

Tools used: file_read (proposal document, .bib file), code_run (Python plotting, API queries, .bib parsing), file_write (corrected .bib, figures), web_scan (arXiv / OpenAlex verification), and update_working_checkpoint (cross-session state).

This case specifically illustrates the paper's claim that layered persistent memory (L0-L3), together with reusable procedures accumulated through self-evolution, can maintain continuity across multi-session academic workflows involving reading, generation, verification, and revision.