SAGE: From Reactive to Programmatic Web Agents via State Machine Memory

Anonymous authors
Paper under double-blind review

Abstract

Many web agents operate through a reactive execution loop: they observe the current interface, reason about the next action, execute it, and repeat. This design incurs latency and cost that grow with the number of actions, while requiring agents to repeatedly rediscover how the same web application works. We present SAGE, a novel architecture that replaces step-by-step reasoning with programmatic execution using reusable knowledge of the application. A Crawling Agent explores the application offline and constructs an updatable *state-machine memory* that represents its interface states, the operations available in each state, and the transitions between states. Unlike trajectory memory, this representation stores how the application works rather than solutions to individual tasks. At runtime, an Execution Agent uses this memory to synthesize a complete executable program in a single planning step, which is then executed deterministically without further planning calls. When the interface changes or the memory is incomplete, a reactive fallback repairs the failed action and updates the memory for future tasks. On 655 tasks across four WebArena domains, SAGE achieves a 91.2% success rate, outperforming the strongest reactive baseline, Claude Code, by 8.5 percentage points while reducing average task latency by $3.2\times$ and cost by $8\times$ using the same backbone model.

1 Introduction

LLM-based web agents automate browser tasks by observing web pages and performing actions such as clicking, typing, and extracting information Nguyen et al. (2025). Most web agents follow a reactive execution paradigm: the agent observes the current page, reasons about the next action, executes it, and repeats Yao et al. (2022). This is a natural approach to an unfamiliar task or environment, where the agent must explore and adapt to what it observes, much like a user encountering an application for the first time.

However, web applications are not entirely new each time they are used. Tasks within the same application share a relatively stable structure: the same pages, controls, operations, and navigation paths recur across tasks. Reactive agents do not fully exploit this structure. They repeatedly rediscovers how the application works, invoking the LLM throughout execution. As a result, their cost and latency grow with the number of actions, and errors can accumulate over long trajectories Cheng et al. (2024); Tao et al. (2023); Shinn et al. (2023); He et al. (2024).

Prior work reuses experience by storing demonstrations or trajectories from previous executions Zheng et al. (2023); Gu et al. (2024); Chae et al. (2024). However, a trajectory packages together the decisions made for one particular task; a new task may reuse only some of its actions, possibly in a different order or context. We argue that representing the *application itself* offers a more compositional form of reuse. For example, knowledge of how to search for a product, read its reviews, and edit its description can be recombined into workflows that were never previously executed. This leads to our central question: *Can web agents replace step-by-step reactive reasoning with offline application knowledge and one-shot planning?*

We present SAGE, an architecture that moves application discovery offline and uses the resulting knowledge to plan and execute new tasks programmatically (Figure 1). A Crawling Agent explores a web application and constructs an updatable *state-machine memory*. The memory represents the application's interface states, the operations available in each state, and the transitions between

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

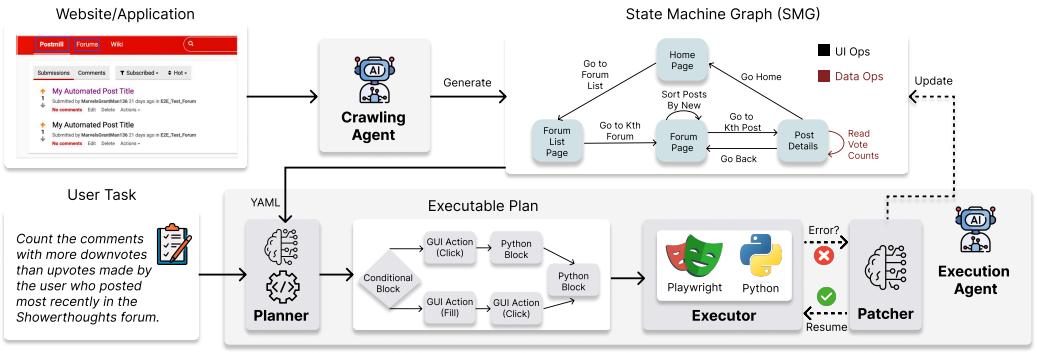


Figure 1: Overview of SAGE. The **Crawling Agent** (top, offline) systematically explores the website to construct the state-machine graph (SMG). The **Execution Agent** (bottom, online) uses the SMG to plan and execute tasks: the *Planner* synthesizes an executable program from the SMG and user task; the *Executor* runs it deterministically without further planning calls; and the *Patcher* intercepts failures, invokes a reactive fallback to recover, and updates the SMG.

states. To keep this memory representation compact, SAGE separates page templates from dynamic content: for example, different product pages are represented as instances of the same underlying state. Unlike trajectory memory, state-machine memory captures task-independent knowledge of how the application works, allowing operations learned from different interactions to be composed into programs for new tasks.

At runtime, an Execution Agent uses this memory to synthesize a high-level program sketch in a single LLM call. A deterministic compiler then grounds each step of the sketch against operations in the state-machine memory to produce an executable plan, which runs without further model calls. In the common case, this changes the unit of LLM reasoning from one call per action to one call per task. Planning against an application-level model also gives the agent access to states and transitions beyond its current observation, rather than requiring each decision to be made locally from the current page. When the memory is incomplete, the interface has changed, or execution depends on information that could not be anticipated, SAGE falls back to reactive reasoning. Successful repairs to stale or incorrect application knowledge are incorporated back into the memory for future tasks.

We evaluate SAGE on 655 tasks across four WebArena domains Zhou et al. (2023). Using the same backbone model, SAGE achieves a 91.2% success rate, 8.5 percentage points higher than the strongest reactive baseline, Claude Code, while reducing average task latency by $3.2\times$ and query-time cost by $8\times$. The offline construction cost is amortized over repeated use: across the evaluated applications, SAGE recovers the one-time crawling cost after 39–101 tasks through lower per-task execution cost. Our analysis also reveals the boundary of programmatic execution: tasks that depend on unpredictable or noisy search results remain difficult to plan completely in advance and often benefit from reactive observation and adaptation.

Our contributions are threefold. First, we introduce *state-machine memory*, a task-independent and compositional representation of an application's interface states, operations, and transitions that allows application discovery to be amortized across tasks. Second, we present SAGE, an architecture that automatically constructs and maintains this memory and uses it to synthesize and execute complete task programs with a single planning call, falling back to reactive reasoning when necessary. Third, across 655 WebArena tasks, we show that reusing application knowledge improves success rate while substantially reducing latency and cost, and characterize search-dependent interactions as an important boundary where reactive execution remains useful. Our results show that reactive reasoning need not be the default for every action: a web agent can learn how an application works, reuse that knowledge across tasks, and react only when the environment demands it.

2 Related Work

Reactive Web Agents. Most existing web agents follow a reactive execution paradigm: the agent observes the current state, invokes an LLM to select the next action, executes it, and repeats until task completion. These approaches differ in how they perceive and interact with the application.

Vision-based agents such as ScreenAgent Niu et al. (2024) and SeeClick Cheng et al. (2024) reason directly over screenshots, providing a general interface that closely resembles human GUI interaction. Text-based agents such as WebWISE Tao et al. (2023) and MindAct Deng et al. (2023) instead operate over structured webpage representations such as HTML or DOM trees. Multimodal agents like WebVoyager He et al. (2024) combine visual and textual information to make more robust decisions. AgentOccam Yang et al. (2025) demonstrates that a simple reactive agent can achieve strong performance without task-specific strategies or additional agent modules by carefully designing its observation and action spaces.

Despite these differences, these agents share the same step-by-step execution loop: each action requires observing the result of the previous action and reasoning about what to do next. As task trajectories grow longer, this repeated inference increases both latency and cost. Other work improves step-wise reasoning by incorporating world models to enhance agent foresight. For instance, WebDreamer Gu et al. (2024) simulates the outcomes of potential actions before execution to select the optimal step, while Gur et al. (2023) explores model-based planning for web navigation. Such approaches provide greater foresight when choosing actions, but still interleave LLM reasoning with execution at each step. SAGE instead investigates whether the repeated planning loop can be replaced by one-shot planning followed by programmatic execution.

Memory for Web Agents. Several web and GUI agents create and maintain a memory as they execute tasks. For example, agents like Synapse Zheng et al. (2023) use a simpler form of memory by retrieving flat, past action sequences ("Trajectory-as-Exemplar") as context. Other work has explored graph-based memory representations. PageGraph Chen et al. (2025) stores page transition relationships in an offline graph structure. AutoDroid Wen et al. (2024) represents mobile apps as GUI transition graphs and uses LLM-generated functional summaries to annotate GUI elements. AutoDroid focuses on optimizing accuracy through richer GUI annotations (e.g., injecting task descriptions into HTML 'onclick' properties). These approaches differ from SAGE along two dimensions. First, memory is used primarily as context for reactive action selection and the agent still reasons about and executes one step at a time. SAGE instead uses memory to synthesize a complete executable program before execution. Second, existing graph memories primarily capture GUI navigation and page transitions. The SMG represents reusable application operations, including both operations that transition between states and operations that access data without changing the application state. This richer operational representation allows the Planner to reason about both control flow and intermediate data when composing a program.

3 State-Machine Memory

To plan a complete task before execution, the agent needs knowledge beyond its current observation: what application states exist, what operations are available in each state, what information those operations return, and where they lead. SAGE represents this knowledge as a state-machine graph (SMG). This representation separates the application's finite set of interface templates from its potentially unbounded dynamic content, keeping the memory compact for planning.

3.1 Representing an Application as a State Machine

SAGE represents a web application as a directed graph $M = (S, O, T)$: states S are stable application views (*e.g.,* a forum's `Home` and `Forum` pages), operations O represent reusable capabilities available in those states (*e.g.,* `GoToKthPost`, `ReadPosts`), and the transition function T records how operations change state.

States as interface templates. A state represents a stable GUI view, such as a webpage. Each state consists of *atoms*, which group related GUI elements that appear together, such as the buttons in a navigation bar or the fields describing a post. Atoms may be shared across states (*e.g.,* navigation bar appears on many pages), but each state is uniquely defined by its specific combination of atoms.

Importantly, states capture recurring application structure rather than individual page instances. As Figure 2 illustrates, `/f/wallstreetbets` and `/f/AskReddit` contain different posts but correspond to the same `ForumPage` state because they share the same interface structure. The state is identified by the atom signature {`Navigation`, `TypeSelection`, `Filter`,

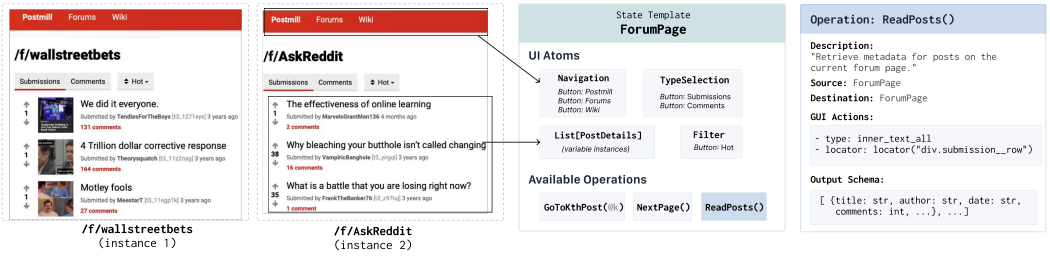


Figure 2: State-machine representation of a web application. Pages with the same interface structure map to a shared state template, which consists of reusable UI atoms and exposes operations over them. Operations store how to access live application data rather than the data observed during construction; for example, `ReadPosts` specifies how to retrieve posts and their output schema.

`List[PostDetails]`}. Here, `List[PostDetails]` specifies that the page may contain zero or more instances of the `PostDetails` atom; the number and contents of these instances do not affect the identity of the state. Consequently, *the graph grows with the number of distinct interface templates rather than the amount of content displayed by the application*. Appendix A.1 describes the atom representation and how SAGE resolves state equivalence in more detail.

Operations as reusable capabilities. Interactions with a state are exposed as *operations*. An operation is a functional unit composed of one or more primitive GUI *actions*, such as clicking, typing, or reading an element. Each operation specifies its inputs, source and destination states, the primitive GUI actions that implement it, and output schema if the operation returns data. For example, Figure 2 shows `ReadPosts` operation, which reads the current post instances and returns their metadata according to a fixed schema.

Operations form SMG's edges. Navigation and update operations may transition to another state, while data-collection operations generally produce self-loops because they read information without changing the current view. Data-access operations additionally specify the schema of their outputs, which can be bound to variables and used by subsequent operations during execution.

Crucially, for dynamic content the SMG stores *access paths*, *i.e.,* the mechanisms for retrieving live data, rather than the particular data observed during construction. For example, `ReadPosts` stores the GUI actions needed to locate and read posts together with its `List[PostDetails]` output schema; it does not store specific posts observed during crawling. Executing this operation later retrieves the current contents of the application. Thus, the same materialized graph can be reused across tasks even as the application's data changes.

3.2 Automated SMG Construction

We assume black-box access to the target application: SAGE can interact with the application through its user interface, but does not have access to its source code or an existing model of its structure. The Crawling Agent constructs the SMG automatically through exploration of the live application. Starting from a seed state, it discovers reachable interface states, identifies the operations available in each state, and records their implementations, inputs, outputs, and transitions. It then executes the discovered operations against the live application to validate them and enrich their descriptions with behavioral details that are not apparent from the interface alone, such as pagination behavior or valid form options. Appendix A.2 describes the construction procedure in detail. Although our implementation focuses on web applications, this construction procedure extends naturally to other graphical interfaces (such as mobile and desktop apps).

Importantly, initial SMG construction is *task-independent*: the Crawling Agent observes only the application and its interactions, not the tasks that the Execution Agent will later solve. The resulting graph therefore captures reusable application knowledge rather than solutions or shortcuts for particular tasks. We serialize this knowledge as YAML containing the states, operations, actions, locators, descriptions, inputs, outputs, and transitions used by the Execution Agent. Figure 3(b–c) shows an excerpt of this representation, including a UI operation (`GoToKthPost`) and its referenced UI atom (`PostDetails`).

```python
async def execute_task():
    OP: NavigateToForum(
        @name="Showerthoughts")
    OP: SortPostsByNew()
    OP: GoToKthPost(@k=0)
    OP: NavigateToComments()
    vote_counts = OP:
     ReadVoteCounts()
    count = sum(1 for v in
      vote_counts if int(v) < 0)
    return count
```

(a) Sketch program

```yaml
name: GoToKthPost
description:
  "Go to Kth post on the
  current forum page."
src_state: ForumPage
dst_state: PostDetailPage
actions:
  - input: ['@k']
    action: PostDetails.
            PostLink.click
```

(b) SMG Operation

```yaml
name: PostDetails
ui_elements:
- name: PostLink
  actions:
  - type: click
    input: '@k'
    locator: locator(
      "nav.submission__nav"
      ).nth(${k})
...
```

(c) Referenced UI Atom

Figure 3: (a) The Planner generates a sketch that expresses application interactions as abstract `OP` calls while using ordinary Python for task logic. (b) Each `OP` refers to an operation in the SMG, which specifies its source and destination states and references actions on UI atoms. (c) The corresponding atom stores the primitive GUI action and locator used to execute the operation.

Because the Crawling Agent is LLM-driven and black-box exploration may not exercise every application behavior, the initial SMG can be incomplete. For example, sparse application data or insufficient interaction coverage may prevent the crawler from activating a control that reveals a hidden dialog. SAGE therefore supports subsequent *SMG refinement*: when execution exposes missing or incorrect application knowledge, the Patcher can repair the corresponding states or operations and commit the repairs back to the SMG for reuse by subsequent tasks (Section 4.3).

4 Online Planning and Execution

At runtime, the Execution Agent uses the SMG to turn a user task into a program and execute it with minimal step-by-step reasoning. As shown in Figure 1, this process involves three components: the *Planner* generates an executable program based on the task and the SMG; the *Executor* runs this program deterministically; and the *Patcher* reactively handles failures and patches the SMG.

4.1 Planner: From Tasks to Executable Plans

Given a user task and the SMG, the Planner constructs a complete program before execution rather than deciding one action at a time. The planner separates the what from the how: it first uses a single LLM call to generate a high-level *sketch program* over operations in the SMG, then deterministically links these operations through the SMG and compiles them into an executable plan.

Sketch generation. Figure 3 illustrates this process for the task "Count the comments with more downvotes than upvotes made by the user who posted most recently in the Showerthoughts forum." The sketch is ordinary Python, except that interactions with the application are expressed as abstract `OP` calls. For example, the Planner generates `GoToKthPost(@k=0)` rather than the low-level GUI actions needed to open the first post. The sketch can freely combine these operations with ordinary Python control and data flow: in this example, the vote counts returned by `ReadVoteCounts` are processed locally to compute the final answer.

Operation arguments are represented as *symbolic inputs*, denoted by an `@` prefix. Their values may be specified directly in the sketch or bound at runtime to values produced earlier in the program. For example, a sketch may extract a username from the current page and later invoke `GoToUser(@name=username)`, where `username` is resolved from the execution context. For tasks requiring semantic reasoning over runtime values, the sketch may additionally invoke an LLM as an explicit helper function.

To generate the sketch, the Planner calls a coding model with the user task, the relevant SMG operations (including their names, descriptions, inputs, and output schema), and a system prompt containing code generation rules and guidance, and explanations related to the structure of the SMG. The model is also provided an LLM-call interface that the generated program can invoke for execution-time semantic reasoning. Importantly, the Planner sees the interface exposed by SMG operations, but not their primitive GUI implementations. The initial planning call therefore determines the program logic and structure even when parts of its computation depend on runtime values.

Planning optimizations: State Filtering and Prefix Caching. For large applications, the SMG may contain thousands of operations, while an individual task may require only a dozen. Exposing the full set of operations to the Planner is therefore both costly and unnecessary. W e observe that the number of states is much smaller, so SAGE uses states as a coarse-grained index into the SMG: a lightweight LLM call first selects a small set of task-relevant states, after which the Planner is given the operations reachable from those states. This reduces both the filtering space and the resulting planning context, without requiring the filter to reason over individual operations.

State filtering creates a challenge for prefix caching because different tasks retrieve different portions of the SMG, producing different planning-prompt prefixes. To recover cache reuse, we introduce a *layered prefix-caching* algorithm that groups related states into a small number of clusters. The planning prompt places globally shared content first, followed by the operations from the best-matching cluster and then any remaining task-specific operations. Tasks that access similar regions of the SMG can therefore share most of their prompt prefix even when their selected states differ. Appendix C.2 describes both optimizations in detail.

Linking and Compilation. The sketch specifies the operations needed to solve the task, but may omit intermediate operations needed to connect them. The linker uses the source and destination states of each operation to identify such gaps, and uses breadth-first search (BFS) over the SMG to find and insert connecting paths. For loops, it additionally connects the end of one iteration to the start of the next.

Once linked, the program contains a valid sequence of SMG operations but still refers to the application through abstract `OP` calls. The compiler deterministically replaces each operation with its implementation stored in the SMG. As shown in Figure 3(b–c), `GoToKthPost(@k=0)` resolves to `PostDetails.PostLink.click`, whose atom definition contains the primitive `click` action and GUI locator. Symbolic inputs such as `@k` are preserved during compilation rather than resolved immediately, since their values may depend on runtime data.

4.2 Executor: Deterministic Program Execution

The compiled program is represented as a control-flow graph (CFG), with GUI actions and Python computations isolated into individual executable nodes. GUI-action nodes are translated into the corresponding Playwright calls, while Python nodes implement the computation and control flow specified by the sketch.

The Executor executes this plan node-by-node without further planning calls. It follows the plan's control flow in a sandboxed environment using a restricted Python interpreter that exposes only whitelisted built-ins (*e.g.,* `math` and `json`) and blocks unsafe system calls such as file and network access. To support data flow across nodes, the Executor maintains an execution context that tracks variable scope and values produced during execution. When a GUI action contains a symbolic input, the Executor resolves it against this context before issuing the corresponding Playwright call.

Executing at node granularity also provides precise failure localization: a runtime error can be attributed to the individual GUI action or Python computation that produced it. This allows failures to be handed to the Patcher without discarding successful execution of the rest of the plan.

4.3 Patcher: Reactive Failure Recovery

When execution fails, the Patcher reactively diagnoses and repairs the failure rather than regenerating the entire plan by default. Because failures are localized to individual execution nodes, the Patcher can localize the failure and resume execution from the repaired region.

For code-related failures (*e.g.,* `JSONDecodeError` or incorrect Python logic), the Patcher uses the stack trace and execution context to edit the faulty code block. For GUI-action failures (*e.g.,* a timeout caused by a stale locator), it can inspect the current application with the GUI automation tool and re-ground the failed action. If the failure reveals stale or incorrect knowledge in the SMG (*e.g.,* wrong locator, poor operator description), the Patcher also commits the repair back to the SMG so that subsequent tasks reuse the corrected operation. In contrast, task-specific plan repairs remain local to the current execution. More substantial planning errors can trigger local plan edits or, when

Table 1: Performance on WebArena by domain. All systems use Claude Opus 4.6. Overall, SAGE improves success rate by 8.5 percentage points over the strongest reactive baseline while reducing average latency by 3.2×

		Success Rate (%) ↑			Avg. Latency (s) ↓		
Domain	**# Tasks**	**AgentOccam**	**Claude Code**	**SAGE**	**AgentOccam**	**Claude Code**	**SAGE**
Reddit	106	76.2	92.4	**95.0**	536	59	**18**
Shopping	187	42.7	79.7	**87.0**	363	97	**28**
Shop. Admin	182	58.4	74.2	**95.3**	1665	101	**31**
GitLab	180	58.3	89.0	**89.0**	499	79	**28**
Overall	655	56.8	82.7	**91.2**	787	87	**27**

necessary, regeneration of a larger portion of the plan. After repair, execution resumes from the affected region.

5 Evaluation

5.1 Experimental Setup

Benchmark and methods. We evaluate SAGE on four domains from WebArena Zhou et al. (2023): Postmill, Shopping, Shopping Admin, and GitLab. retrieval and aggregation, form-based workflows, and repository management. We compare against two strong reactive agents: *AgentOccam (AOccam)* Yang et al. (2025), a published web agent that uses vision-based observations, and *Claude Code with Playwright MCP*, which interacts with websites through programmatic browser commands and direct DOM access. Both agents plan and act step-by-step without a persistent model of the environment. All methods use Claude Opus 4.6 as the backbone model.

For SAGE, we first construct an initial SMG using the offline crawler and then refine it during a separate warm-up phase. During warm-up, we generate and execute one task for each WebArena task template and use the Patcher to update the SMG when execution exposes missing or incorrect application knowledge. Warm-up uses the task templates but not the evaluation task instances or their instantiated parameters. The Patcher is disabled during evaluation, so each reported task reflects a single execution attempt without online repair. Unless otherwise noted, SAGE uses the refined SMG and prefix caching. We analyze both in Section 5.3.

Evaluation protocol. We report task success rate using WebArena's evaluator, end-to-end latency, and LLM API cost computed from input and output token usage. WebArena's reference-based evaluator can reject correct answers because of formatting differences or multiple valid answers, a known limitation Thakkar et al. (2025); Lù et al. (2025); Zhu et al. (2025). We therefore manually review failed tasks for all methods. We count objectively correct answers that differ only in formatting or from the reference answer as successful; for subjective tasks, we retain the original evaluator judgment. We apply the same correction criteria to all methods.

5.2 End-to-End Performance

Table 1 summarizes success rate and latency across all four domains; Figure 4 breaks down average token usage and estimated cost per task.

Success rate. SAGE achieves 91.2% overall success, outperforming Claude Code (82.7%) by 8.5 percentage points and AgentOccam (56.8%) by 34.4 points using the same backbone model. The gain over Claude Code is modest on Reddit (95.0% vs. 92.4%), but grows to 7.3 points on Shopping and 21.1 points on Shopping Admin, which contain more complex multi-step form workflows. On GitLab, SAGE matches Claude Code at 89.0%.

Latency. SAGE reduces average task latency to 27 seconds, a 29× speedup over AgentOccam (787s) and 3.2× over Claude Code (87s). This improvement follows from replacing iterative reactive

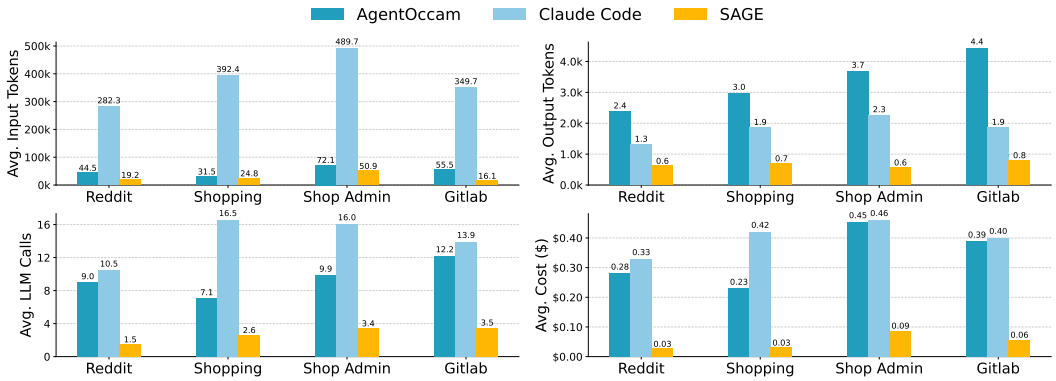


Figure 4: Average input tokens, output tokens, LLM calls, and estimated API cost per task. SAGE reduces input tokens by 14–22× and LLM calls by 4–6× compared to Claude Code.

execution, where each step requires a separate LLM call and browser round-trip, with one-shot plan generation followed by deterministic execution. The reduction in the number of LLM calls also reflects this: from 7–16 calls per task for reactive baselines to 1.5–3.5 for SAGE (Figure 4). Calls beyond the initial planning call arise from two sources: a generated program may explicitly invoke an LLM helper for semantic reasoning that depends on values available only at runtime, and large SMGs may require an additional lightweight state-filtering call before planning (Appendix B.1). The speedup is most dramatic on Shopping Admin, where SAGE completes tasks in 31s compared to 1,665s for AgentOccam—a 54× reduction—as this domain's deep multi-page workflows incur the highest per-step overhead under reactive execution.

Cost. Token costs are estimated using Anthropic API pricing ($5/M input tokens, $25/M output tokens) and token usage metrics from the Claude Agent SDK age (2026). AgentOccam incurs an average cost of $0.34 per task. Although Claude Code consumes significantly more input tokens (282k–490k per task), its effective cost ($0.40) is not proportional to raw token counts because it leverages prompt caching, where repeated context prefixes are served at 10% of the standard rate.

SAGE reduces average input tokens by 14–22× compared to Claude Code and 2× compared to AgentOccam, and output tokens by 2–4× and 4–6×, respectively. Together with its prefix-caching optimization, this reduces the average API cost to $0.05 per task—8× lower than Claude Code and 6.8× lower than AgentOccam.

Qualitative takeaways. The SMG is particularly useful for multi-step tasks that require planning across pages or structured extraction from complex interfaces: the SMG exposes available operations and their outputs upfront, allowing the Planner to compose them without repeatedly rediscovering the application. After online refinement, none of the observed failures are caused by broken selectors or structural deficiencies in the SMG. Instead, the remaining failures primarily involve ambiguous task intent (∼23 cases) or search-dependent tasks (∼22 cases). The latter expose a key limitation of plan-then-execute agents: the Planner must choose search terms before observing the results and therefore cannot adapt when search behavior is brittle, results are noisy, or the site's terminology differs from the task. Detailed success and failure cases are provided in Appendix D.

5.3 Detailed Analysis and Ablation Studies

SMG construction and refinement. Table 2(a) reports both the one-time cost of constructing an SMG and the quality of the resulting memory. Across the four applications, crawling takes 36–260 minutes and costs $11.6–$37.2. Because the resulting SMG is reused across tasks, this cost is amortized over subsequent executions. Relative to Claude Code, SAGE's lower per-task execution cost recovers the one-time crawling cost after 39–101 tasks.

The crawler-produced SMGs already support substantial task success. Using the crawler SMG, SAGE achieves 73.1% overall success rate, on par with Claude Code performance without any online correction except in the Gitlab domain. Refinement further improves success across all four domains, with an average improvement of 18.1 percentage points. The largest improvement appears

Table 2: (a) Cost and quality of the SMG produced by offline crawling. Parenthesized operation counts report operations gained or lost after refinement; refinement does not change the number of states. Break-even is the number of tasks required to amortize the one-time crawling cost relative to Claude Code. (b) Effect of prefix caching on average API cost per task.

(a) SMG construction and refinement

Domain	Cost ($)	Time (min)	Break-even (# Tasks)	States	Ops	Succ. (%)
Reddit	11.6	36	39	57	130 (+2)	89.0
Shopping	16.9	132	44	59	105 (+2)	80.7
Shop Admin	37.2	260	101	184	1,182 (-4)	74.0
GitLab	29.5	230	88	219	1,148 (+66)	55.0

(b) Prefix caching

Domain	w/o Cache	Cache
Reddit	$0.11	$0.03
Shopping	$0.14	$0.03
Shop Admin	$0.26	$0.09
GitLab	$0.10	$0.06
Overall	$0.16	$0.05

on Shopping Admin (74.0% $\rightarrow$ 95.3%) and GitLab (55.0% $\rightarrow$ 89.0%), both of which have the most complex interaction patterns (multi-level forms, conditional fields, dynamic GUI elements). Simpler domains like Reddit (89.0% $\rightarrow$ 95.0%) and Shopping (80.7% $\rightarrow$ 87.0%) show a smaller gap, as the crawler's initial graph is already close to complete. Both variants use the same execution pipeline, with the Patcher disabled during evaluation, so these differences isolate the quality of the SMG. The Patcher is restricted to repairing general application knowledge and cannot introduce task-specific shortcuts or composite operations, so refinement improves reusable application knowledge rather than memorizing solutions to evaluation tasks.

We also note that these gains require little change to the structure of the SMG. Refinement does not add any states, and changes the number of operations only modestly: Reddit and Shopping each gain two operations, Shopping Admin loses four through merging redundant operations that are confusing to the planner, and GitLab gains 66. Most improvements instead come from repairing or enriching existing operations. Using Shopping Admin as a representative example, these repairs fall into three categories: *(1) selector and locator repairs*, which replace interactions that fail to trigger the intended GUI behavior; *(2) timing fixes*, which add waits for asynchronous GUI updates; and *(3) description enrichment*, which records application behavior discovered during execution, such as noting that a save action stays on the edit page rather than returning to the grid.

Prefix caching. Table 2(b) isolates the effect of our layered prefix-caching optimization. Without caching, SAGE costs $0.16 per task on average; caching reduces this to $0.05, a $3.2\times$ reduction. The savings are largest on Reddit and Shopping, where the SMG is small enough to expose all operations directly to the Planner and tasks therefore share most of the planning prompt. Caching remains effective on the larger Shopping Admin and GitLab domains, reducing cost from $0.26 to $0.09 and from $0.10 to $0.06, respectively. In these domains, state filtering makes the SMG context task-dependent, reducing the prefix naturally shared across tasks; layered prefix caching recovers reuse by organizing related states into shared prompt prefixes.

6 Conclusion

In this work, we introduced SAGE, an architecture that transitions Web agents from reactive, step-by-step interaction to programmatic execution. By capturing reusable application knowledge in an updatable state-machine memory, SAGE synthesizes complete executable programs in a single planning step and executes them largely without further reasoning. On WebArena, this approach improves both task success and efficiency, achieving 91.2% success while reducing latency by $3.2\times$ and cost by $8\times$ relative to Claude Code.

AI use statement

In this work, we used generative AI tools to polish the writing of the manuscript and to assist with the implementation of the research prototype. All AI-assisted edits to the manuscript were reviewed by the authors to ensure that they preserved the intended meaning. Code developed with AI assistance

was reviewed and tested by the authors for correctness. We take responsibility for the final content of this work, including text or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

SAGE is a system for learning and executing reusable workflows in web applications. Although this work is intended to improve the reliability and efficiency of user-directed automation, systems with these capabilities may also create risks related to unauthorized actions, privacy, security, data rights, and misuse.

Data and experimental environments. Our experiments use researcher-controlled environments and accounts (public benchmarks). We do not collect data from human subjects or evaluate the system using private data belonging to uninvolved individuals. SAGE may record interface state, action trajectories, page content, and screenshots when constructing an action index or training data. Such artifacts can contain personal information, copyrighted material, authentication state, or other sensitive content if the system is applied to uncontrolled websites. Researchers and deployers are responsible for ensuring that data collection is authorized, consistent with applicable terms and licenses, and subject to appropriate minimization, redaction, retention, and access-control practices. We do not release authentication credentials, browser storage state, or other secrets as part of this work. Any published trajectories or action indexes should be reviewed for sensitive information prior to publication.

User authorization and potential harms. Automation can amplify both beneficial and harmful activity. Possible misuse includes unauthorized account activity, spam, manipulation of online content, privacy-invasive collection, and automation of deceptive or otherwise prohibited workflows. The crawler excludes tasks that involve authentication, account creation or deletion, payment, and certain other sensitive interactions. SAGE also supports human confirmation before designated actions. These measures reduce but do not eliminate risk: task classification can fail, and human confirmation is configurable rather than an absolute security boundary. The system should therefore be operated only on applications and accounts for which the operator has authorization. Deployments should require confirmation for actions with external side effects—such as purchases, submissions, communications, account changes, or deletion—and should apply additional domain-specific controls in consequential settings. We do not recommend deploying the research prototype for medical, legal, financial, employment, public-benefit, or safety-critical decisions.

Privacy and security. SAGE can use authenticated browser sessions and can optionally retrieve user memories to personalize task execution. These capabilities may expose sensitive information through model inputs, screenshots, execution results, telemetry, or logs. Production deployments should obtain informed user consent, limit collection to information necessary for the requested task, prevent sensitive content from being written to logs, encrypt retained data, enforce least-privilege access, and provide deletion and retention controls. The memories retrieved should be treated as contextual information rather than instructions and should not override the current request of the user.

Limitations, fairness, and integrity of the research. Our evaluations measure performance in a limited set of applications and benchmark conditions and should not be interpreted as establishing equivalent reliability across websites, languages, geographies, accessibility configurations, or user populations. Interface changes, ambiguous instructions, adversarial content, and classification errors can cause incorrect actions. We report the experimental scope and evaluation methodology so that results can be interpreted within these limitations. To the best of our knowledge, the work was conducted in accordance with applicable institutional policies and the ICLR Code of Ethics.

REPRODUCIBILITY STATEMENT

Although we are unable to release the SAGE source code, we have taken necessary steps to support independent reproduction and rigorous verification of our results. The main paper and appendix provide a complete description of the system's architecture and methodology, document the experimental setup, datasets, task configurations, model and dependency versions, evaluation metrics, baselines, and statistical analysis. Additionally, we can provide representative execution traces, intermediate artifacts, detailed ablation results, and the configuration parameters required to recon-

struct each reported experiment. Together, these materials are intended to enable researchers to re-implement the proposed approach, reproduce the evaluation protocol, and compare independently-obtained results with those reported in the paper.

REFERENCES

2026. Claude API Docs - Track cost and usage. https://platform.claude.com/docs/en/agent-sdk/cost-tracking. Accessed: Mar 2026.

Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (2008), P10008.

Hyungjoo Chae, Namyoung Kim, Kai Tzu-iunn Ong, Minju Gwak, Gwanwoo Song, Jihoon Kim, Sunghwan Kim, Dongha Lee, and Jinyoung Yeo. 2024. Web agents with world models: Learning and leveraging environment dynamics in web navigation. *arXiv preprint arXiv:2410.13232* (2024).

Weizhi Chen, Ziwei Wang, Leyang Yang, Sheng Zhou, Xiaoxuan Tang, Jiajun Bu, Yong Li, and Wei Jiang. 2025. PG-Agent: An Agent Powered by Page Graph. In *Proceedings of the 33rd ACM International Conference on Multimedia*. 6878–6887.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. Seeclick: Harnessing gui grounding for advanced visual gui agents. *arXiv preprint arXiv:2401.10935* (2024).

Edward G Coffman, Jr, Michael R Garey, and David S Johnson. 1978. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* 7, 1 (1978), 1–17.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems* 36 (2023), 28091–28114.

Yu Gu, Kai Zhang, Yuting Ning, Boyuan Zheng, Boyu Gou, Tianci Xue, Cheng Chang, Sanjari Srivastava, Yanan Xie, Peng Qi, et al. 2024. Is your llm secretly a world model of the internet? model-based planning for web agents. *arXiv preprint arXiv:2411.06559* (2024).

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2023. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856* (2023).

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919* (2024).

Xing Han Lù, Amirhossein Kazemnejad, Nicholas Meade, Arkil Patel, Dongchan Shin, Alejandra Zambrano, Karolina Stańczak, Peter Shaw, Christopher J Pal, and Siva Reddy. 2025. Agentrewardbench: Evaluating automatic evaluations of web agent trajectories. *arXiv preprint arXiv:2504.08942* (2025).

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, Xintong Li, Jing Shi, Hongjie Chen, Viet Dac Lai, Zhouhang Xie, Sungchul Kim, Ruiyi Zhang, Tong Yu, Mehrab Tanjim, Nesreen K. Ahmed, Puneet Mathur, Seunghyun Yoon, Lina Yao, Branislav Kveton, Jihyung Kil, Thien Huu Nguyen, Trung Bui, Tianyi Zhou, Ryan A. Rossi, and Franck Dernoncourt. 2025. GUI Agents: A Survey. arXiv:2412.13501 [cs.AI] https://arxiv.org/abs/2412.13501

Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. 2024. Screenagent: A vision language model-driven computer control agent. *arXiv preprint arXiv:2402.07945* (2024).

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

Heyi Tao, Sethuraman TV, Michal Shlapentokh-Rothman, and Derek Hoiem. 2023. Webwise: Web interface control and sequential exploration with large language models. *arXiv preprint arXiv:2310.16042* (2023).

Megh Thakkar, Nicolas Chapados, Christopher Pal, et al. 2025. Webarena verified: Reliable evaluation for web agents. In *Workshop on Scaling Environments for Agents*.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th annual international conference on Mobile computing and networking*. 543–557.

Ke Yang, Yao Liu, Sapana Chaudhary, Rasool Fakoor, Pratik A Chaudhari, George Karypis, and Huzefa Rangwala. 2025. Agentoccam: A simple yet strong baseline for llm-based web agents. In *International Conference on Learning Representations*, Vol. 2025. 97533–97565.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2022).

Ruichen Zhang, Mufan Qiu, Zhen Tan, Mohan Zhang, Vincent Lu, Jie Peng, Kaidi Xu, Leandro Z Agudelo, Peter Qian, and Tianlong Chen. 2025. Symbiotic Cooperation for Web Agents: Harnessing Complementary Strengths of Large and Small LLMs. *arXiv preprint arXiv:2502.07942* (2025).

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2023. Synapse: Trajectory-as-exemplar prompting with memory for computer control. *arXiv preprint arXiv:2306.07863* (2023).

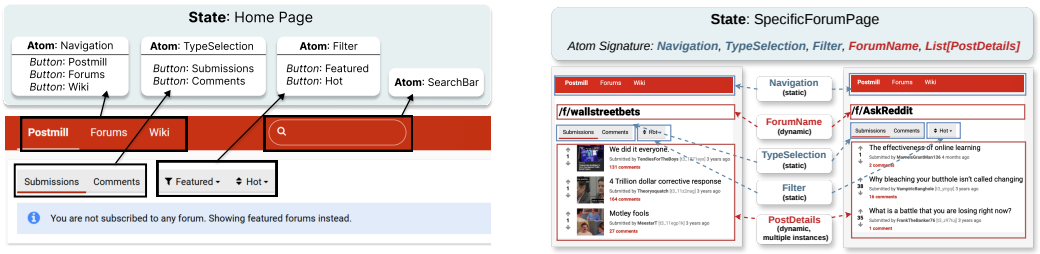
Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854* (2023).

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. 2025. Establishing best practices for building rigorous agentic benchmarks. *arXiv preprint arXiv:2507.02825* (2025).

A Offline Memory Construction

The Crawling Agent simulates a user trying to "learn" the GUI application – *What is this application useful for and how do I use it?* Rather than saving and trying to reuse task-specific solutions Zheng et al. (2023); Zhang et al. (2025), the Crawling Agent tries to learn the entire topology of the GUI application (the distinct pages/windows and their constituent GUI elements) and the meaning of every GUI element (e.g., clicking on this next button will navigate to the next page of posts under this forum). It then organizes this knowledge as a state machine graph (SMG) memory, which is an intuitive representation that developers already use to design GUIs.

The Crawling Agent constructs the SMG offline through systematic exploration, capturing the intrinsic structure of the application independent of any downstream task. The design explicitly decouples the static topology (bounded by a finite set of templates) from dynamic data content (potentially unbounded) to keep the state space compact and reusable. The resulting compact, symbolic representation serves as the foundation for the Execution Agent's one-shot planning capabilities at runtime.



(a) Atoms in the `Home Page` state.

(b) State equivalence across dynamic page instances.

Figure 5: Application states and their Atoms.

A.1 State Machine Graph

The state machine graph represents a GUI application as a directed graph whose nodes correspond to symbolic states and whose edges correspond to executable operations. Formally, we define the graph as a state machine $M = (S, O, T)$, where S is a set of discrete states, O is a set of executable operations, and $T : S \times O \to S$ is the transition function.

A **State** $s \in S$ represents a unique and stable *view* of a GUI container (e.g., a webpage, a window) for GUI elements (e.g., buttons, menus, tabs, text fields). It is formally defined as a collection of atoms, where an **Atom** represents a set of related GUI elements that appear atomically (i.e., either all of the GUI elements in an atom are present in the view or none of them). Although atoms may be shared between states, a state is *uniquely* identified by the *combination* of its atoms. For example, the `HomePage` state in Figure 5a, is characterized by four atoms: the `Navigation` atom containing three buttons (Postmill, Forums, Wiki), the `SearchBar` atom containing a text input and a button, the `PostTypeSelection` atom containing two buttons (Submissions and Comments) and the `Filter` atom containing two selection lists. The `Navigation` and `SearchBar` atoms are shared between all the states of the Postmill/Reddit application.

Interactions with the GUI application are modeled similarly using a two-level hierarchy of Actions and Operations. An **Action** captures a *primitive* interaction with a *single* GUI element in an atom (e.g., click on a button, type in a text field, or move the scroll bar). An **Operation** $op \in O$ is a functional, goal-oriented unit composed of one or more low-level actions. For example, the operation `Sort Posts By New` includes two actions: `click('Sort Dropdown')` followed by `click('New Option')`.

An operation is represented as an edge in the SMG. It operates on atoms pertaining to a *single* state and may result in a *single* state transition. It is possible to define *compounded operations* with multiple state transitions (spanning multiple edges in the SMG), but in the current implementation we only experimented with single state operations. The functionality of these operations vary from simple navigation (e.g., `NextForumPage`, `GoToForumList`, etc.), to reading or updating content (`ReadAllPosts`, `ReadKthPost(k)`, `UpdateUserBio(new_bio)`). GUI manipulation operations typically induce a transition to a new state (e.g., `GoTo ForumList` from `HomePage` leads to the `ForumListPage` state), whereas Data-collection operations introduce self-loops because they do not change the agent's state in the graph (e.g., `ReadAllVoteCounts` on the `PostDetails` page). Instead of triggering a state transition, these operations extract information from the GUI and store it in symbolic variables that can be referenced in the online execution stage.

Preventing State Explosion. A central design challenge in the construction of the SMG is preventing state explosion. For example, a large application such as Amazon contains billions of distinct items customers can inspect, yet modeling each item-specific page as a separate state would make the state graph intractable. To remain usable for planning, the graph size should scale with the number of distinct *state templates*, not with the number of *data instances* rendered within those templates.

Figure 5b illustrates this key property of our state abstraction: a single state template may have an unbounded number of instantiations, each populated with different data. In this example, the two forum pages for "wallstreetbets" and "AskReddit" contain different posts, but they correspond to

the same `ForumPage` state because they share an identical page structure. This state equivalence is achieved by distinguishing *static* and *dynamic* atoms. Static atoms encode invariant interface elements (*e.g.,* `Navigation`), while dynamic atoms represent elements with data-dependent content (*e.g.,* `ForumName` and `PostDetails`). Each dynamic atom essentially defines a "type" (analogous to a class definition in object-oriented programming) that can be instantiated zero or more times with different values.

Under this formulation, the `ForumPage` template is defined by its atom signature: {`Navigation`, `TypeSelection`, `Filter`, `ForumName`, `List[PostDetails]`}. The typed collection `List[PostDetails]` specifies that the page expects zero or more instances of the `PostDetails` atom. Although the number of and the content of these instances differ across pages, the state identity is determined by this fixed structural signature, not by the specific data rendered within it.

A.2 SMG Construction

The SMG is constructed offline through a fully automated crawl-and-validate pipeline. Starting from a seed state (e.g., the home page), the Crawling Agent incrementally builds a complete and consistent representation of the application's interaction structure through four phases:

(1) State discovery. The agent systematically explores the application with the help of a GUI automation tool to enumerate distinct states and transitions. It performs a depth-first traversal of the interface, identifying and classifying outgoing links on each view into: (i) navigation links, explored exhaustively; (ii) data-item links, where representative instances are sampled; and (iii) action-triggering controls (e.g., buttons that reveal hidden forms or wizards). This phase yields an initial set of states and transitions capturing the application's reachable topology.

(2) Operation definition. For each discovered state, the agent defines GUI operations (actions that manipulate the GUI) and data operations (read-only extractions). Each operation is associated with a concrete grounding (e.g., selectors or locators). Data operations additionally record an expected output schema (e.g., a list of strings or a flat array) so the Execution Agent knows the format of returned data and can correctly incorporate it into downstream logic.

(3) Description enrichment. The agent probes the live site and enriches each operation's description with behavioral details that cannot be inferred from the GUI structure alone. For example, a search operation's description may specify whether matching is exact, fuzzy, or prefix-based; a pagination operation may record the default sort order and items per page; a form operation may list valid dropdown options and index conventions. These details are critical because the Planner relies solely on operation descriptions to understand what each operation does and how to invoke it correctly.

(4) Validation. Each operation is tested against the live site: GUI operations verify that selectors resolve to existing elements; data operations execute the read and validate both the selector and output format. Failures trigger a fix-and-retest loop until all operations pass.

The current design treats the application as a black box, relying solely on live UI interaction to discover functionality; inspecting application source code, when available, could supplement crawling to uncover hard-to-reach or conditionally rendered states.

GUI-driven design. Our design philosophy is to ensure that the memory graph captures the intrinsic structure of the GUI application, rather than overfitting to the specific tasks. Operations are therefore defined purely based on observable GUI elements and their available interactions, independent of any task semantics. We explicitly avoid introducing "virtual" operations that encode composite or task-specific intent. For example, if a post has three clickable entry points (Title, Thumbnail, "Read More"), each is recorded as a distinct operation inducing a parallel transition, even if they lead to the same destination state. The Execution Agent is responsible for reasoning over these alternatives during planning.

While the initial SMG enforces a strict one-to-one mapping between operations and GUI elements, it can be extended over time based on usage. In particular, frequently repeated sequences of operations (e.g., `Login` $\rightarrow$ `Open Settings`) can be identified and cached as reusable *compounded operations*, similar to materialized views. This reduces the planning overhead by allowing the agent to reuse common interaction patterns instead of rediscovering them each time.

```yaml
# ===== STATES =====
- !State
 name: ForumPage
 atoms:
   - name: Navigation
   - name: TypeSelection
   - name: ForumName
   - name: PostDetails
     type: List
# ===== ATOMS =====
- !Atom
  name: PostDetails
  type: dynamic
  ui_elements:
    - name: "PostLink"
      actions:
        - type: click
          input: '@k'
          locator:
            locator("nav.submission__nav").nth(${k})
    - name: "PostInfo"
      actions:
        - type: inner_text
          input: '@k'
          locator: locator("div.submission__row").nth(${k})
        - type: inner_text_all
          locator: locator("div.submission__row")
# ===== OPERATIONS =====
- !Operation
  name: GoToKthPost
  description: "Navigate to the Kth post on the current forum page. Index starts from
      0."
  src_state: &ForumPage
  dst_state: &PostDetailPage
  actions:
    - input: ['@k']
      action: PostDetails.PostLink.click
- !Operation
  name: ReadPosts
  description: "Retrieve titles and metadata for posts on the current forum page."
  src_state: &ForumPage
  dst_state: &ForumPage # self-loop
  actions:
    - action: PostDetails.PostInfo.inner_text_all
  output_format: "['Post book depression\n\nSubmitted by bertiewoooster t3_124f4e2 3
      years ago\n\n185 comments\n965', ...] - ['Title\nPost Author\nDate\nNumber
      comments\nNumber Votes', ...]"
```

Listing 1: Excerpt from the YAML encoding of the Reddit state-machine graph, illustrating a UI operation (`GoToKthPost`) and a data operation (`ReadPosts`).

Serialization Format. The finalized graph is serialized into a YAML format optimized for the Execution Agent. Listing 1 shows an excerpt from the Reddit SMG, illustrating two representative operations: `GoToKthPost`, which transitions from `ForumPage` to the `PostDetails` state, and `ReadPosts`, which returns a list of strings with information about each post displayed on the current forum page (e.g., self loop on the `ForumPage` state). The locators and action types for every Atom's GUI element, which uniquely identify the GUI element and the possible interactions, are specific to the GUI automation tool used, in our case Playwright. While operations' inputs and outputs are not strictly typed, the Planner reliably infers how to parse and use them from the natural-language descriptions and example output formats provided in the SMG.

B Online Planning and Execution

With the SMG in place, providing a global understanding of the GUI app, the Execution Agent's *Planner* is now able to create an *executable plan* for the task with *a single reasoning step* (one call to the model). The executable plan is essentially the control flow graph (CFG) of a (Python) program, except that basic blocks may be further split such that each action (GUI interaction) is isolated in its own basic block. The Execution Agent's *Executor* then executes this plan one basic block at a time, for better error isolation and efficient repair. Any error encountered while executing the

plan or failure to produce an executable plan (e.g., due to missing operations in the SMG, or poor descriptions) causes the Execution Agent to fallback on the reactive approach: the error/failure is presented to the Execution Agent's *Patcher*, which operates in an observe–reason–act loop until the issue is resolved (by either simply editing the code in the faulty basic block, or changing the plan, or updating the SMG).

B.1 Generating a Sketch Program

The first step towards an executable plan is to generate a *sketch program*: a (Python) script that specifies the task's logical structure (e.g., control and data flow) while abstracting concrete GUI interactions (actions) into operation placeholders. The Planner achieves this with a call to a coding model, which receives as context: a) the (relevant) operations from the SMG (their names, unique ids, descriptions, any inputs, and any outputs), b) a carefully engineered system prompt, which contains code generation rules and guidance, and explanations related to the structure of the SMG, c) *LLM call as a tool*, for tasks that require reasoning at execution time (e.g., determining if a post recommends a single book), and d) the user task.

The sketch program serves as a *logically complete* blueprint of the task. Providing operations as context (omitting the actions), allows the coding model to focus on the higher-level solution, not on a full implementation (which will be realized through the next deterministic stages). The operation placeholders use the format `OP: [OpID]: OpName`. The `OpID` uniquely identifies an operation in the SMG, avoiding ambiguity when multiple operations share similar names. Data-collection operations are tagged with metadata describing their output format, which enables the coding model to incorporate retrieved data into program logic.

The sketch program supports *symbolic variables*, denoted by an `@` prefix (e.g., `@username`), to represent values that are only available at runtime. For example, a data-collection operation may first extract raw text into a local variable (e.g., `comments`); a Python Node—a locally executed code block (e.g., for-loops, parsing, filtering, aggregation, and other data-processing logic)—then processes `comments` to derive a target value (e.g., the manager's name). The Planner specifies how this derived value should bind to symbolic arguments `@username` in downstream GUI operations.

Finally, the sketch program also supports *helper functions*, specialized (Python) functions that encapsulate complex logic (e.g., LLM-based inference, multi-step parsing) separate from the main GUI-driven control flow, improving modularity when tasks require sophisticated data processing.

```python
def find_forum_index(forums_data, name):
    # returns index of forum matching name, or None
    ...

async def classify_posts(post_details):
    prompt = f"""For each post, determine if it recommends exactly one specific book.
    Posts: {post_details}
    Return JSON: {{"urls": ["url1", "url2"]}}"""
    parsed_json = await LLMCall(prompt)
    return parsed_json["urls"]

async def execute_task():
    OP: [1]:GoToForums()
    forums_data = OP: [15]:ReadForums()
    idx = find_forum_index(forums_data, "books")
    while idx is None:
        has_next = OP: [16]:GoToNextPageOfForums()
        if has_next == "0":
            break
        forums_data = OP: [15]:ReadForums()
        idx = find_forum_index(forums_data, "books")
    OP: [6]:GoToKthForum(@k=@idx)
    OP: [12]:SortPostsByTop()
    post_details = []
    while len(post_details) < 10:
        page_posts = OP: [19]:ReadPosts()
        for i in range(len(page_posts)):
            OP: [29]:GoToKthPost(@k=@i)
            title = OP: [21]:ReadPostTitle()
            body = OP: [22]:ReadPostBody()
            url = OP: [0]:ReadURL()
            post_details.append({"title": title, "body":
                body, "url": url})
            OP: [4]:GoBackToPreviousPage()
```

```
            if len(post_details) >= 10:
                break
        has_more = OP: [17]:GoToNextPageOfPosts()
        if has_more == "0":
            break
    result = await classify_posts(post_details)
```

Listing 2: Example sketch program with OP placeholders, symbolic variables, and helper functions.

Optimization: State Filtering. For small applications, all operations are included directly in the planning prompt. Larger applications, however, can expose hundreds of states and thousands of operations—far more than a single task requires. Including the full set wastes tokens and introduces irrelevant context that can degrade plan quality. In such cases, we invoke a *state filter*: a lightweight LLM call that takes the full list of state names and descriptions along with the user task, and returns a small subset of relevant states (typically 3–6). Only the operations associated with these states are then passed to the Planner. Filtering on states rather than operations keeps the filter prompt compact, as the number of states is often at least an order of magnitude smaller than the number of operations. State filtering is triggered when the total operation count exceeds 300; below this threshold the overhead is unnecessary. Since state filtering produces variable prompt subsets across tasks, we design a layered prefix caching scheme to enable efficient cache reuse under this variable-prefix setting (§ C.2).

Example (Sketch Program). Consider the example task: *Among the top 10 posts in "books" forum, show me the post URLs that recommend a single book.* The Planner generates the sketch program shown in Listing 2 (simplified for clarity). The sketch demonstrates several key features: (1) `OP` placeholders with operation IDs (e.g., `[16]`, `[19]`) that will be expanded to concrete actions after compilation; (2) the helper function `classify_posts` encapsulates LLM-based reasoning as a specialized Python function, keeping complex inference logic separate from the main GUI-driven control flow; (3) symbolic variables like `@idx` whose values are computed at runtime and bound dynamically; (4) control flow (`if-else`) is fixed at plan time based on the task structure, while the actual branch taken depends on runtime data (e.g., collecting enough posts).

B.2 Compiling into Executable Plans

The Planner transforms the abstract sketch into a fully executable plan in two steps: *linking*, which resolves each `OP` placeholder into concrete execution paths in the SMG, and *compilation*, which assembles the result into an execution-ready plan with runtime symbolic binding. Both steps are performed programmatically and deterministically via a custom compiler without any model calls.

First, the Planner expands each `OP` placeholder into a valid sequence of graph edges in the SMG while preserving the control-flow semantics established in the IR. Before expansion, it validates connectivity: for each consecutive pair of operations, a breadth-first search (BFS) checks whether a valid transition exists from the previous operation's end state to the next operation's start state. For operations inside loops, the check additionally verifies that the path returns to the loop's entry state after each iteration. If any disconnection is detected, the specific break points are reported to the Planner's model, which applies a targeted local edit to repair the affected portion of the sketch. In our evaluation with Claude Opus 4.6, generated sketches were always connected and this repair step was never triggered; however, it serves as a safety net for weaker planning models that may produce inconsistent operation sequences.

Once all `OP`s have been expanded, the system compiles the resulting program into an executable plan that is type-safe, execution-ready, and supports runtime symbolic binding. The sketch's control-flow graph (CFG) is constructed with each action isolated into its own basic block (CFG node). Each `OP` is then replaced with its corresponding implementation (e.g., Playwright API call). For example, ⟨`ACTION` type=`click`, locator=`locator("nav.submission__nav").nth(@i)`⟩ will be replaced with the `page.locator("nav.submission__nav").nth(i).click()` Playwright call, ready for execution. Each basic block carries its execution context, code logic, and data binding requirements.

C IMPLEMENTATION

C.1 CRAWLING AGENT

The Crawling Agent is implemented as a set of skills on top of Claude Code with the Playwright MCP server, which provides programmatic browser control via the Model Context Protocol. The agent reads page structure through Playwright's accessibility snapshots and interacts with elements using standard Playwright selectors and locators.

The crawling pipeline combines LLM-driven exploration with hard-coded validation scripts. The LLM handles tasks requiring semantic understanding, such as identifying page templates, classifying interactive elements, and composing operation descriptions. Deterministic scripts enforce convergence: phases 1 and 2 are each gated by a validation script that must pass before the next phase begins, with failures triggering a fix-and-retest loop. For example, the phase 2 convergence check performs a BFS from the seed state and enumerates all interactive and data-bearing DOM elements on each reachable page. It flags two types of gaps: elements with no corresponding operation, and elements where an existing operation's selector fails to match. The agent iterates—adding missing operations or repairing selectors—until the script reports full coverage. Phase 4 is itself a validation pass that executes every operation on the live site; failures are fed back to the LLM for repair until all checks pass.

For the most complex application in our evaluation (GitLab, 103 states, 1,082 operations), the full four-phase pipeline completes in approximately 1.5 hours, consumes 704k input tokens, and costs approximately $5—a one-time offline investment amortized across all subsequent task executions.

C.2 EXECUTION AGENT

We implemented the Planner and Executor with 11,017 lines of Python code. The Planner produces a `MixedActionPlan`—a tree of typed nodes that the Executor traverses at runtime. Each `OP` expands into its SMG action sequence; Python lines aggregate into Python block nodes; and control-flow statements are converted via AST analysis into recursive container nodes preserving the original nesting. Helper functions are hoisted to a root-level node so they are defined before the main execution flow.

The Patcher is implemented as a Claude Agent SDK agent equipped with Playwright MCP as a built-in tool, and read/write access to the SMG. The runtime invokes the Patcher asynchronously when a failure occurs, passing it a detailed execution trace including visited URLs, executed actions, and their success/failure status. The failure handling logic is described in § **??**; SMG edits made by the Patcher are persisted so that subsequent task executions automatically benefit from the fix.

Optimization: Layered Prefix Caching. The state filter (§B.1) selects task-relevant states, but different tasks select different subsets, yielding distinct prompt prefixes that prevent cache reuse. We introduce a three-layer prompt structure that restores a shared, cacheable prefix across tasks.

GUI apps expose *global navigation operations* (e.g., `go_to_dashboard`) available from every state. These create a star topology in the state transition graph that prevents meaningful partitioning. We first extract global operations into a shared layer, then partition the residual graph into K balanced clusters (Algorithm 1). Louvain community detection Blondel et al. (2008) splits large connected components along natural business-domain boundaries, so that states commonly needed together remain in the same community. An LPT bin-packing step Coffman et al. (1978) then assigns communities to K size-balanced buckets. The procedure runs once at application load time in $O(|V|+|E|)$; on a 143-state application, wall time is under 15 ms.

Given the state filter output $\mathcal{S}_{\text{rel}}$, we select the best-matching cluster k^* (covering the most states in $\mathcal{S}_{\text{rel}}$) and assemble the prompt in three layers: **Layer 1** (shared across all tasks) contains global operations and system instructions; **Layer 2** (shared within cluster k^*) contains all operations in the best-matching cluster; **Layer 3** (unique per task) contains operations from spillover states not covered by k^* (typically 0–2) plus the task description. If every cluster is substantially larger than $\mathcal{S}_{\text{rel}}$, Layer 2 is omitted to avoid adding irrelevant context, falling back to a two-layer scheme. Layers 1 and 2 are each marked with a cache breakpoint, so tasks sharing the same cluster reuse both layers from cache. Since Layer 2 has only K variants, the theoretical Layer 2 cache hit rate

Algorithm 1: Clustered Prefix Caching

Input: State-action graph $\mathcal{G}=(\mathcal{S}, \mathcal{O}, \mathcal{E})$, clusters K
Output: Global ops $\mathcal{O}_g$, clusters $\{C_1, \ldots, C_K\}$, mapping $\phi: \mathcal{S} \to \{1, \ldots, K\}$

$\mathcal{O}_g \leftarrow \{o \in \mathcal{O} \mid \texttt{src}(o)=\varnothing\}$;
$\mathcal{O}_l \leftarrow \mathcal{O} \setminus \mathcal{O}_g$;
$H \leftarrow$ undirected graph on $\{\texttt{src}(o) \mid o \in \mathcal{O}_l\}$ with edges from $\mathcal{O}_l$;
$\bar{n} \leftarrow |\mathcal{O}_l|/K$;
$\mathcal{P} \leftarrow \varnothing$;
foreach connected component $\mathcal{C}$ in H **do**
    if $|\mathrm{ops}(\mathcal{C})| > 1.5\,\bar{n}$ ***and*** $|\mathcal{C}| > 3$ **then**
        $\mathcal{P} \leftarrow \mathcal{P} \cup \textsc{Louvain}(H[\mathcal{C}],\ \lceil |\mathrm{ops}(\mathcal{C})|/\bar{n} \rceil)$;
    else
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{C}\}$;
Sort $\mathcal{P}$ by $|\mathrm{ops}(\cdot)|$ descending;
Init buckets $C_1, \ldots, C_K \leftarrow \varnothing$;
foreach $P \in \mathcal{P}$ **do**
    $k^* \leftarrow \arg\min_k |\mathrm{ops}(C_k)|$;
    $C_{k^*} \leftarrow C_{k^*} \cup P$;
return $\mathcal{O}_g,\ \{C_1, \ldots, C_K\},\ \phi(s)=k\ \forall s \in C_k$

over N tasks is $(N-K)/N$ assuming no cache expiration, as each cluster incurs at most one cold miss.

D Detailed Task Performance

D.1 Representative successes

The majority of tasks complete with a single LLM call; some additionally require lightweight runtime LLM calls for intermediate reasoning (e.g., determining which commenter is the website manager, or selecting a target product from order items). We highlight four tasks where SAGE succeeds but Claude Code fails, illustrating two structural advantages of programmatic planning over reactive execution.

Global planning from complete application knowledge. Because the Planner sees the full SMG upfront, it can design multi-step strategies that span multiple states without iterative runtime discovery or backtracking. For example, in Task 107 (*"Present the monthly count of successful orders from May to December 2022,"* Shopping Admin), the Planner generates a `for` loop over 8 months, where each iteration applies a status filter, a date-range filter, reads the record count from a summary operation, and resets filters for the next iteration. This strategy requires knowing that filter and reset operations exist, that they compose correctly, and that a summary operation can return the count directly without reading individual rows. Claude Code, seeing only the current snapshot at each step, does not know these operations exist until it discovers them interactively; it forgets to reset filters between months and misidentifies controls whose accessibility labels are auto-generated IDs rather than semantic names.

Similarly, in Task 464 (*"Update the Antonia Racer Tank description with positive user reviews,"* Shopping Admin), the Planner designs a three-state workflow: navigate to the Reviews grid to read and filter positive reviews via a runtime LLM call, then to the Products grid to open the product form, then write the composed description. Because the Planner sees operations across all states simultaneously, it plans the optimal data-gathering-then-writing workflow upfront. Claude Code navigates to the product page first, realizes it needs review content, backtracks to the Reviews page, then returns—but the grid retains the previous search filter, preventing it from finding the target product.

Grounded operations enabling efficient data extraction. Because each SMG operation carries validated selectors and known output schemas, SAGE extracts data cleanly without parsing noisy page snapshots. In Task 321 (*"How much refund from orders cancelled in 2022, including shipping?"*

Shopping), the Planner reads each table column (ID, date, status, total) via separate data operations with precise selectors, producing aligned lists that a helper function zips and filters. Claude Code must extract the same fields from full-page accessibility snapshots at each pagination step, parsing ∼2,000 lines mixing headers, data rows, and navigation elements, leading to row-column misalignment and an incorrect total.

Task 284 (*"Show the least expensive shoe storage with minimum 12-pair capacity,"* Shopping) illustrates how the two advantages work together to generate a more efficient execution strategy. Knowing that a single data operation can return all product names and prices at once, the Planner reads the full list in one shot and passes it to a single batch runtime LLM call that evaluates all candidates simultaneously. Claude Code, unaware of this operation, clicks into each product individually (one reasoning step per product), misses candidates, and reports an incorrect result.

D.2 Failure analysis

Notably, after online SMG refinement, zero failures are caused by selector or graph structural issues. The system also correctly identifies 7 tasks that are impossible on the target site (e.g., *"Change the delivery address for my most recent order"*, which Magento does not support), reporting them as infeasible rather than producing incorrect actions.

Ambiguous tasks (∼23 cases): the task intent or judgment criteria are ambiguous, and the system produces a reasonable answer that differs from the benchmark reference. For example, *"Thumbs down the top 2 posts ever in history"* (Reddit) is interpreted as the top posts site-wide rather than posts in the forum named "history." On Shopping, *"How much did I spend on hair care in January 2023?"* requires classifying which products count as "hair care"—the system includes items like body butter and headbands, yielding a different total than the benchmark. On Shopping, *"What are the main criticisms of this product?"* requires selecting which review sentences constitute "main criticisms"—which review sentences constitute "main criticisms" is a matter of interpretation. *"List reviewers mentioning ear cups being small"* (Shopping) finds 4 out of 5 reviewers but misses one who expresses the same complaint with indirect language rather than the exact phrase. On Shopping Admin, *"5 blue Cronus yoga pants arrived, update the stock"* is ambiguous about whether to add 5 to current stock or set stock to exactly 5.

Search-dependent tasks (∼22 failures): the plan-then-execute model struggles when tasks depend on keyword search results that cannot be anticipated at planning time. For example, *"List all opened issues about OPT model"* (GitLab) searches "OPT model," which returns zero results because the actual issues use titles like "OPT-175B inference" and "Open Pre-trained Transformer"—the site's keyword search cannot bridge this semantic gap. *"Show the least expensive switch card holder with minimum 15 card capacity"* (Shopping) searches "switch card holder," but the site's search engine returns irrelevant products (phone wallets) alongside the intended Nintendo Switch card holders, and the Planner cannot filter out noise without observing the actual results. On Reddit, *"Open the thread of a trending post on the forum 'machine learning' "* fails because the forum is named "MachineLearning" —the exact-string search does not match.

These failures reflect a core limitation of the programmatic paradigm: the Planner must commit to search keywords at plan time without runtime observation. We intentionally disable the Patcher during evaluation to stress-test this limitation; with the Patcher enabled, SAGE can observe search results and iteratively refine keywords, functioning similarly to a reactive agent for search-dependent tasks.