Scalable and Privacy-Enhanced Multimodal LLM Agents for Cross-Platform GUI Automation

Anh-Nhat Nguyen[2034311]

University of Mannheim, Germany

Abstract. *We present a comprehensive study of multimodal GUI-automation agents that operate without reliance on proprietary cloud APIs. Our initial prototype combined OmniParser's layout-aware UI parsing with a local GPT-4o-mini instance, but suffered from prohibitive resource consumption, frequent timeouts, and only 40% end-to-end success. To address these limitations, we migrated to a cloud-hosted Anthropic Claude 3.5 pipeline—augmented with automated evaluation, retry management, and a Streamlit review interface—which improved success to 56% (TSR), 89% step accuracy (SSR), and 57% efficiency (AE) over 20 tasks. Detailed error analyses reveal six core failure modes (e.g. viewport drift, element misidentification, form-flow fragility), while targeted prompt refinements (e.g. "Tab" navigation hints, explicit step-by-step plans) convert intermittent failures into robust successes. We discuss the trade-offs between lightweight local agents and scalable cloud solutions, outline remaining challenges (security widgets, long-horizon planning, vendor lock-in), and release our open-source framework and benchmark suite to foster reproducible, privacy-preserving research in cross-platform GUI automation.*

Keywords: Multimodal, LLM Agents, GUI Automation, Privacy Preserving, Cross Platform, Open Source, Cost Effective, Autonomous Agents.

1 Introduction (4P)

1.1 Problem Statement

Modern GUI-automation agents promise to liberate users from repetitive workflows by interpreting natural language commands and interacting directly with applications. However, most state-of-the-art systems rely on proprietary, cloud-hosted vision-language APIs—such as GPT-4V or Gemini Vision—to parse screens and generate actions. This dependence incurs three critical drawbacks:

First, it raises serious privacy concerns. Sending screenshots or command logs to third-party servers exposes potentially sensitive data and conflicts with regulations or corporate policies that mandate on-premise processing.

Second, the high per-call cost of commercial APIs makes large-scale or real-time deployment economically prohibitive, particularly for organizations without deep pockets or in domains with stringent cost constraints.

Third, vendor lock-in stifles innovation and portability: agents built against one provider's opaque interface cannot easily migrate to alternative models or run in air-gapped environments.

On the other hand, fully local solutions—pairing self-hosted LLMs with open-source UI parsers—often falter under real-world conditions. Limited compute resources lead to sluggish response times, out-of-memory failures, and brittle performance on complex, multi-step tasks. Moreover, common web and desktop widgets (e.g. infinite-scroll lists, CAPTCHA challenges) expose gaps in purely vision-based pipelines and flat prompting strategies.

This work addresses these intertwined challenges by (1) quantifying the trade-offs between local and cloud agents on a representative suite of desktop and web tasks; (2) isolating principal failure modes through systematic error analysis; and (3) proposing a hybrid, reproducible framework that balances privacy, cost, and robustness—leveraging on-device UI parsing where feasible and judicious use of cloud capacity where necessary, all within a unified evaluation pipeline.

1.2 Related Work

Computer control agents (CCAs) represent a growing class of artificial intelligence systems designed to operate computing environments—such as desktops, mobile platforms, or web interfaces—via natural language instructions. These agents must perceive dynamic user interfaces, interpret high-level goals, and perform sequences of low-level actions (e.g., clicks, keystrokes) to complete tasks. This problem domain lies at the intersection of GUI automation, instruction following, and generalist agent design. In this section, we review the evolution of CCA research, focusing on methodological developments, core system architectures, and evaluation frameworks.

Early Approaches. Initial research into instruction-based agents primarily adopted reinforcement learning (RL) paradigms in controlled environments. For instance, Branavan et al. [**?**] used RL to map documentation to editor commands, showing early signs of grounding language in action. This line of work evolved into environments like MiniWoB [**?**], which offered standardized, low-complexity web tasks for instruction following. However, these early agents were typically specialized for specific domains, dependent on handcrafted observation and action spaces. They performed reasonably well in narrow settings but lacked the scalability and temporal reasoning required for complex, real-world GUI tasks.

The Shift to Foundation Models. The introduction of large-scale foundation models—particularly large language models (LLMs) and vision-language models (VLMs)—has shifted the paradigm from task-specific to more generalist agents. These models offer powerful out-of-the-box capabilities for language understanding and reasoning. Rather than training policies from scratch, recent agents use pre-trained models to interpret tasks and plan action sequences. This transition is often analyzed across three dimensions: the agent's environment, its mode of interaction, and its internal architecture [**?**, **?**]. For example, agents may observe

the environment visually (screenshots), structurally (HTML), or via accessibility APIs, and may act through pixel-level controls or direct API calls.

Concrete examples include WebGPT [**?**], which augments an LLM with a Browse interface, and Mind2Web [**?**], which generalizes task execution across unseen websites. While these models showcase impressive generalization, their reliance on large, proprietary APIs often entails significant computational cost and raises data privacy concerns, motivating the exploration of more accessible and self-contained alternatives.

Architectural Components and Memory. As CCAs grow in capability, modular architectures have become increasingly common. A typical agent pipeline includes components for observation encoding, instruction parsing, action planning, and execution. Many systems integrate these components with a central LLM using techniques like chain-of-thought prompting [**?**]. The literature distinguishes between memoryless agents, history-based agents, and state-based agents depending on how they encode temporal context [**?**]. Many current agents overlook long-term memory, relying only on recent observations. This limitation, often coupled with a dependency on a cloud-hosted model for reasoning, contributes to brittle performance and reinforces the cost and privacy challenges we aim to address.

Evaluation and Benchmarks. The field has also matured in its evaluation methodologies. Early benchmarks like MiniWoB++ [**?**] were suitable for RL but inadequate for testing generalization. More recent benchmarks, such as Mind2Web and WebArena [**?**], offer open-domain tasks on realistic websites, enabling the assessment of zero-shot performance and robustness. The community has developed a range of metrics beyond simple task success, including step-level accuracy, action grounding error, and robustness to UI changes [**?**]. Standardized and reproducible evaluation setups are crucial for fairly comparing different approaches, especially when assessing trade-offs between proprietary models and open-source solutions.

Recent Advances and Lingering Gaps. Recent developments continue to push the boundaries of what CCAs can achieve. Anthropic, for instance, demonstrated that its Claude 3.5 model can operate computers in a sandboxed environment using only visual input and simulated controls, mirroring human-like interaction [**?**]. While technically impressive, this capability remains locked within a proprietary, closed-source model, highlighting the exact vendor lock-in problem this paper seeks to solve.

In parallel, progress in interface abstraction, such as Microsoft's OmniParser [**?**], offers a more open path forward. OmniParser creates a unified, layout-aware encoding for GUIs that is model-agnostic. This type of modular, interface-level standardization is a key enabler for building robust agents that are not tied to a specific foundation model, aligning directly with our goal of promoting platform versatility.

In summary, research in computer control has evolved from specialized, RL-based agents to generalist systems powered by foundation models. While these modern agents demonstrate remarkable capabilities, they often inherit the limitations of the large, proprietary models they depend on, namely high operational costs and significant privacy concerns. Key challenges remain in long-horizon planning, robust action grounding, and adapting to novel interfaces. The trend towards modular components like OmniParser suggests a viable path toward creating more open and versatile systems. This work builds on that trajectory, focusing specifically on developing an agent architecture that is effective, cost-efficient, and fundamentally privacy-preserving.

1.3 Structure

2 Experimental Design (3P)

2.1 Dataset

To evaluate the agent's capabilities across diverse real-world computing scenarios, we introduce a benchmark suite comprising 20 desktop interaction tasks. The dataset was curated to balance coverage and feasibility, reflecting a realistic distribution of user-level challenges on personal computers. Tasks span a spectrum of difficulties (Easy, Medium, Hard), and are classified into five functional domains: (1) Basic App/File Operations, (2) Programming and Scripting, (3) Communication and Forms, (4) Web Automation, and (5) System Monitoring and Data Extraction. These categories were selected to capture core interaction modalities encountered in desktop environments—ranging from GUI-based control to automated data collection—and were informed by prior work in instruction-following agents and task automation [**?**, **?**]. Each task includes a natural language instruction, a domain label (encoded numerically for compactness), and a zero-shot prompt suitable for LLM-based agents. A detailed breakdown of all tasks is provided in Table **??**.

We also manually identify the Optimal Actions for computing Action Efficiency (AE) (see Section **??**); for Task ID 1 (*Launch Calculator and compute* $123 + 456$), this yields four steps: open Calculator, enter "123+456," press "=," and read the result.

2.2 Local LLM Agent

In the initial phase of our study, we developed a purely local agent by coupling OmniParser with a locally hosted instance of GPT-4o-mini. OmniParser operated on high-resolution macOS screenshots, performing object detection to identify UI elements (including bounding boxes, element types, and any visible text) and emitting a structured JSON representation of the current interface. This JSON payload, together with the user's natural-language instruction, was fed into GPT-4o-mini, which generated a sequence of executable actions (e.g.,

`click(x,y)`, `type("...")`) to accomplish the task, the architecture shown in **??**.

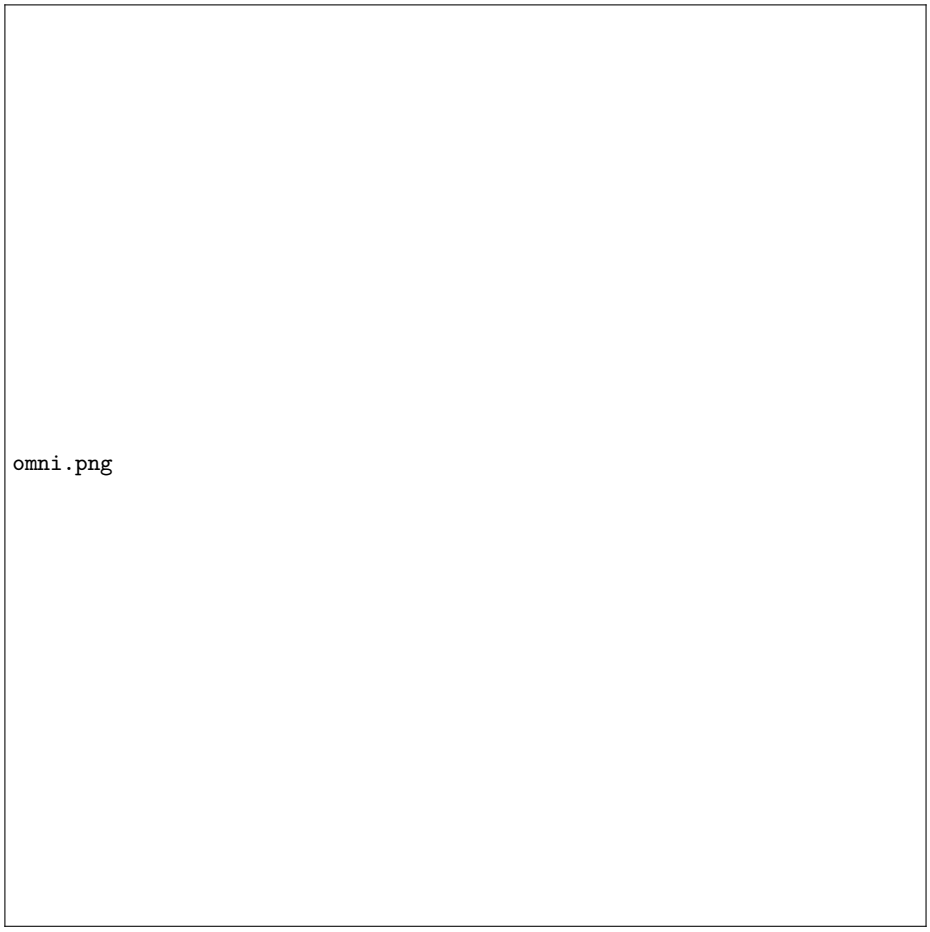

Fig. 1: Architecture for experimental design.

However, during experimentation we encountered severe performance bottlenecks. Inference with GPT-4o-mini on a local machine Apple Macbook M1 Pro consumed nearly 100% of available CPU resources and exhausted 16 GB of RAM, resulting in repeated out-of-memory failures and multi-minute runtimes per task. These technical limitations prevented reliable computation of finer-grained metrics such as Action Efficiency (AE) and Step Success Rate (SSR), since each evaluation run frequently timed out or crashed before completion.

Accordingly, we restricted our evaluation to the Task Success Rate (TSR) in **??**, the percentage of instructions fully executed to termination. We bench-

marked the Local LLM Agent on dataset **??**. The agent achieved an overall TSR of 40%, with performance stratified as 60% on easy tasks, 30% on medium tasks, and 20% on hard tasks,

To further gauge robustness, we then evaluated the same local agent on an additional set of production-like tasks drawn from the Claude Computer Use benchmark. In this extended evaluation, the Local LLM Agent failed to complete any of the new tasks, yielding a 0% TSR. Given these underwhelming results, compounded by prohibitive computational demands and unstable execution, we discontinued the local-only approach and migrated our evaluations to the cloud-based Claude Computer Use system, which provides higher model capacity, more efficient scaling, and greater stability for complex GUI automation tasks

Task Success Rate (TSR) Out of the 20 tasks, the local pipeline fully completed $N_{\text{succ}} = 7$ before a terminal failure, yielding

$$\text{TSR}_{\text{local}} = \frac{7}{20} \times 100 = 40\%.$$

2.3 Anthropic Claude for Computer Use

Building on the local LLM environment described in Section **??**, we constructed a hybrid evaluation pipeline around Anthropic Claude 3.5 and a Streamlit-based interface. Key features include:

- **Automatic Evaluation:** Each execution trace is fed back into Claude 3.5, which—using a standardized prompt—labels each step as correct or incorrect and predicts overall task success. This automated scoring accelerates throughput and provides a consistency check against human annotations.
- **Streamlit Review System:** A custom Streamlit application (run_eval.py) presents screenshots and command logs in sequence, allows evaluators to confirm or override Claude's labels, and captures optimal action counts. Interactive charts of TSR, SSR, and AE update in real time.
- **Retry Management:** On task failure, the system can automatically requeue the same instruction with modified prompts or environmental tweaks (e.g. explicit focus hints), up to a configurable retry limit, enabling systematic prompt refinement experiments.
- **Trajectory Logging:** All agent–environment interactions—including natural-language instructions, Claude's responses, GUI/API actions, and full-screen snapshots—are recorded in a structured JSON trajectory. These logs support reproducible replay, error analysis, and offline batch evaluation via a CLI tool (eval_cli.py).

This architecture unifies high-volume automated scoring with human-in-the-loop oversight, supports iterative prompt engineering through managed retries, and ensures that every step of the agent's behavior is captured for detailed post-hoc analysis.

2.4 Evaluation Metrics

To assess agent performance under realistic and constrained computing conditions, we adopt three principal evaluation metrics: **Task Success Rate (TSR)**, **Step Success Rate (SSR)**, and **Action Efficiency (AE)**. These metrics are widely used in the literature [**?**, **?**, **?**] and provide complementary perspectives on the correctness, completeness, and efficiency of agent behavior.

Task Success Rate (TSR) measures how often the agent completes an entire task without a critical error. In settings where each of our T tasks is executed R times (e.g. different seeds or starting conditions), we treat each run as an independent trial. This lets us capture variability: a task that sometimes succeeds and sometimes fails contributes proportionally.

$$\mathrm{TSR} = \frac{1}{T\,R} \sum_{j=1}^{T} \sum_{r=1}^{R} \mathbf{1}\{\text{task } j \text{ succeeded on run } r\} \;\times\; 100.$$

Here $\mathbf{1}\{\cdot\}$ is the indicator function (1 if true, 0 otherwise). Reporting TSR in "mean ± std dev" or with confidence intervals then directly reflects run-to-run variability.

Step Success Rate (SSR) (also called action accuracy or partial match) evaluates how many of the agent's individual actions align with a reference trajectory. By summing over all tasks and runs, we avoid biasing toward tasks with more or fewer steps:

$$\mathrm{SSR} = \frac{\sum_{j=1}^{T} \sum_{r=1}^{R} N_{\text{correct steps}}^{(j,r)}}{\sum_{j=1}^{T} \sum_{r=1}^{R} N_{\text{total steps}}^{(j,r)}} \;\times\; 100.$$

A high SSR (e.g. > 90 %) indicates reliable, per-action accuracy even if entire tasks sometimes fail.

Action Efficiency (AE) measures how concisely the agent reaches success, penalizing redundant or mistaken steps. Again we aggregate over tasks and runs:

$$\mathrm{AE} = \frac{\sum_{j=1}^{T} \sum_{r=1}^{R} N_{\text{useful actions}}^{(j,r)}}{\sum_{j=1}^{T} \sum_{r=1}^{R} N_{\text{total actions}}^{(j,r)}} \;\times\; 100.$$

3 Results and Analysis (3P)

3.1 Overall Evaluation

Across 20 distinct tasks, we performed 34 total executions—including manual retries to refine prompts or adjust the environment. Of these 34 runs, 19 succeeded and 15 failed, yielding an aggregate Task Success Rate of 0.56. At the

task level, 18 tasks succeeded at least once, while 2 tasks (IDs 5 and 11) failed in every attempt.

The agent's performance on these 34 executions is summarized in Table **??**. Although individual action precision is strong, end-to-end success rate and efficiency metrics reveal significant gaps that merit further investigation.

Table 2: Overall Evaluation Metrics (34 executions across 20 tasks)

Metric	**Mean**	Std. Dev.	Min	25%	Median	75%	Max
TSR	**0.56**	0.50	0.00	0.00	1.00	1.00	1.00
SSR	**0.89**	0.24	0.00	0.96	1.00	1.00	1.00
AE	**0.57**	0.30	0.00	0.33	0.50	0.73	1.00
Total Steps	9.59	7.74	0.00	4.00	6.00	14.75	27.00
Actual Actions	7.26	5.72	0.00	4.00	4.00	12.75	20.00
Optimal Actions	4.21	1.49	1.00	3.25	4.00	6.00	6.00

The Task Success Rate is only moderate (TSR = $\mathbf{0.56} \pm 0.50$), meaning that 56% of execution attempts finish successfully while 44% fail outright. Because each run's TSR is strictly 0 or 1, even a few incorrect or misordered actions can cause the entire task to fail.

The mean Step Success Rate (SSR = $\mathbf{0.89} \pm 0.24$) demonstrates that the agent correctly executes individual low-level operations in nearly 9 out of 10 cases. Such a high SSR is on par with or exceeds figures reported in prior GUI automation research, where per-action accuracies typically range from 0.75 to 0.85.

Action Efficiency further underscores inefficiencies: AE = $\mathbf{0.57} \pm 0.30$ means the agent takes nearly twice the optimal number of actions on average (mean Actual = 7.26 vs. Optimal = 4.21). The wide interquartile range (IQR = 0.33–0.73) reveals that while some executions are near-optimal, others suffer severe redundancy, suggesting that the agent's planning heuristics lack consistency and may over-explore GUI states or fail to exploit affordances effectively.

3.2 Evaluation by Category

Table **??** breaks down performance across five domain categories (n=34 tasks, categories available for 34).

Table 3: Performance Metrics by Task Category

Category	n	TSR (mean ± SD)	SSR (mean ± SD)	AE (mean ± SD)
Basic App/File Operations	9	0.78 ± 0.44	0.85 ± 0.24	0.65 ± 0.34
Communication & Forms	10	0.20 ± 0.42	0.90 ± 0.32	0.56 ± 0.21
Programming/Script Tasks	3	1.00 ± 0.00	0.69 ± 0.34	0.75 ± 0.43
System Monitoring / Data Extraction	3	1.00 ± 0.00	0.92 ± 0.14	0.61 ± 0.35
Web Automation	9	0.44 ± 0.53	0.96 ± 0.11	0.42 ± 0.27

Category-Level Findings

- **Basic App/File Operations** (n = 9) achieve relatively high end-to-end success (TSR = 0.78 ± 0.44) but still display moderate step errors (SSR = 0.85 ± 0.24) and inefficiency (AE = 0.65 ± 0.34). Improving UI-state perception and path-planning could reduce redundant steps.
- **Communication & Forms** (n = 10) suffer the lowest TSR (TSR = 0.20 ± 0.42) despite strong per-step accuracy (SSR = 0.90 ± 0.32). This decoupling underscores the need for robust sequence-control and field-validation mechanisms in form-filling and text-entry contexts.
- **Programming/Script Tasks** (n = 3) reach perfect end-to-end success (TSR = 1.00 ± 0.00) but show the lowest step success (SSR = 0.69 ± 0.34), indicating challenges with precise syntactic operations despite coarse success criteria. Grammar-aware action models may improve reliability.
- **System Monitoring / Data Extraction** (n = 3) also hit perfect success (TSR = 1.00 ± 0.00), with high SSR (SSR = 0.92 ± 0.14) and moderate AE (AE = 0.61 ± 0.35). Deterministic environments facilitate robust performance, suggesting that improved environment modeling directly boosts accuracy and efficiency.
- **Web Automation** (n = 9) yields mid-range TSR (TSR = 0.44 ± 0.53) but the highest SSR (SSR = 0.96 ± 0.11) and lowest AE (AE = 0.42 ± 0.27). Dynamic page layouts and asynchronous content allow correct individual actions but compound small errors into brittle workflows—advanced DOM-aware planning and checkpointing are needed.

3.3 Error Analysis

Table **??** summarizes the principal failure cases, embedding each Task ID with a descriptive hint for clarity.

Table 4: Key Failure Modes by Task

ID – Task	Error Mode	Underlying Cause
5 – Using Safari to extract data on Web	Viewport drift	Fixed scroll increments without speed control
8 – Inserting receipt title in a email	Element mis-identification	Fragile selector and missing focus verification
11 – Solving CAPTCHA on 2captcha demo site	External-service limitation	No integration with CAPTCHA solver or human fallback
13 – Filling online payment form	Form flow fragility	Missing precondition checks and adaptive field sequencing
19 – Checking system CPU/RAM status	Instruction divergence	Ambiguous specification of preferred environment

Task 5 – Scroll Control and Viewport Drift In Task 5, fixed scroll steps caused the agent to overshoot and miss the target content. Implementing adaptive, feedback-driven scrolling based on element visibility would prevent such drift.

Task 8 – Element Identification and Focus Navigation In Task 8, imprecise selectors led to appending text to the wrong element until a Tab-based focus hint corrected the field selection. Combining robust XPath/ARIA selectors with explicit focus verification would eliminate these phantom successes.

Task 11 – External Service Interactions In Task 11, the agent failed to bypass or solve the CAPTCHA widget under all prompt variations. Automating secure web widgets requires integrating specialized solving APIs or human-in-the-loop interventions.

Task 13 – Dynamic Form Flows and Resource Checks In Task 13, the agent became stuck in scroll loops and encountered payment blocks due to low credit. Adaptive field sequencing and precondition checks (e.g., account balance verification) are necessary to navigate dynamic forms reliably.

Task 19 – Instruction Divergence in Deterministic Tasks In Task 19, the agent defaulted to Python scripts instead of using shell commands as specified. Clear, unambiguous environment and language specifications in prompts can align agent behavior with user expectations.

These error modes—viewport drift, element mis-identification, granularity mismatch, service limitations, form fragility, and instruction ambiguity—highlight key avenues for improving adaptive control, selector robustness, metric design, and domain-specific integrations to enhance agent performance.

3.4 Case Study Analysis

Iterative Refinement and Partial Successes To investigate how prompt engineering and minimal code adjustments can remedy failures, we examined three tasks that initially failed but ultimately succeeded after refinement.

Task 8 – Email Composition in Mail App The first prompt (*"Compose and send an email"*) led to mis-placed text entry in the recipient field. By refining the prompt to *"fill the content in the correct textbox"*, the agent improved but still entered the subject line incorrectly. Finally, an explicit step-by-step execution plan (*"Open Mail → New Message → Click 'To' → Enter recipient → Click 'Subject' → Enter subject → Click body → Enter body → Send"*) produced a flawless run, demonstrating that detailed action sequences can resolve field-targeting errors.

Task 13 – Microsoft Forms Filling Initially, the agent stalled on an "invalid URL" error. Adding *"slowly scroll down"* to reveal hidden inputs enabled partial progress, yet a required checkbox was skipped. Switching to a Tab-key navigation strategy (*"use Tab to traverse each input field in order"*) ensured all elements were addressed and the form submitted successfully, highlighting the efficacy of focus-agnostic traversal in dynamic web contexts.

Task 19 – System Resource Plotting A Python script that wrote output directly to ' /Desktop' failed due to a missing directory. Revising the script to save locally then move the file to the Desktop directory corrected the FileNotFoundError. The agent subsequently generated, saved, and relocated the resource plot as intended, illustrating that small environment-specification changes can resolve filesystem errors.

Persistent Failures Two tasks remained unsolvable despite extensive prompt variations, revealing fundamental limitations beyond prompt engineering.

Task 5 – Safari News Extraction Across three prompt variants—basic extraction, conditional scroll, and *"slowly scroll down until you see the News section"*—the agent consistently missed the dynamic news element. The absence of viewport feedback or element-visibility checks prevented reliable localization, indicating a need for visual context integration or direct DOM querying.

Task 11 – CAPTCHA Solving on 2captcha Demo Six distinct prompts, including role reframing (*"You are a white-hat hacker"*), direct click instructions (*"Click to Verify"*), and switching to reCAPTCHA v2, all failed due to policy guardrails. The agent's refusal behavior underscores that secure widgets cannot be bypassed via prompt design alone, suggesting the necessity of specialized solver APIs or human-in-the-loop interventions.

These case studies underscore two conclusions: (1) detailed, plan-based prompts and environment clarifications can convert intermittent failures into consistent

successes; (2) certain capability gaps—dynamic content handling and security policy constraints—require architectural enhancements or external integrations rather than further prompt refinement.

4 Discussion and Conclusion

4.1 Summary of Findings

Our initial OmniParser and GPT-4o-mini agent proved impractical: it consumed excessive local resources, timed out frequently, and achieved only a 40 % TSR, preventing stable SSR and AE measurement. Migrating to Anthropic Claude 3.5 with a Streamlit-based orchestration, automated scoring, and retry management yielded marked improvements with TSR = 56 %, SSR = 89 %, and AE = 57 % over 20 tasks (34 runs).

Beyond these headline metrics, our experiments uncovered several operational insights. First, incorporating a simple "Tab" navigation hint dramatically improved form-filling and web-automation success, demonstrating the value of focus-agnostic traversal in dynamic interfaces. Second, Claude consistently preferred issuing Bash commands for system and programming tasks, suggesting that shell pipelines are its de facto scripting idiom. Third, tasks in deterministic domains (scripting, system monitoring) exhibited near-perfect TSR, indicating strong promise for automation in controlled environments. Finally, the "heavy-human" style of explicit, step-by-step execution plans proved effective: detailed guidance that mirrors human protocols reliably converted partial failures into successes.

4.2 Limitations and Future Work

Despite the stability gains of our cloud-hosted pipeline, challenges remain. The reliance on a proprietary API entails ongoing cost and potential vendor lock-in. Secure widgets (e.g. CAPTCHAs) remain impervious to prompt engineering alone and require either solver APIs or human-in-the-loop intervention. Lack of direct DOM or accessibility-API hooks limits robust element visibility checks, and long-horizon tasks continue to suffer from error compounding under flat prompting.

To address these issues, future work should pursue more efficient on-device parsers and LLMs, integrate direct interface-level feedback (DOM/accessibility APIs), develop hierarchical memory and planning modules, and experiment with hybrid human–machine or authorized-solver integrations for protected services. Moreover, expanding our benchmark to mobile and legacy desktop contexts will test generalization across a wider array of GUIs.

4.3 Concluding Remarks

Our comparative study of a lightweight local agent and a scalable Claude 3.5 pipeline establishes a clear performance baseline and illuminates practical tech-

niques—such as Tab navigation hints and explicit execution plans—that significantly enhance multimodal GUI automation. By standardizing TSR, SSR, and AE metrics, building a reproducible Streamlit evaluation framework, and conducting in-depth failure analyses, we provide both a toolkit and a roadmap for advancing privacy-preserving, cost-effective, and robust multimodal agents across computing environments.

5 Technical Detail

This work is experimental and conducted as part of Seminar CS 715[1]. We highly recommend consulting our GitHub repository for a complete demonstration of our implementation and tech stack[2].

[1] CS 715 seminar course page
[2] CS 715 ComputerUse GitHub repo

Table 1: Computer Use Task Dataset

ID	Difficulty	Task	Category	Prompt
1	Easy	Launch the Calculator app (via Spotlight or Launchpad) and compute 123 + 456.	1	Launch Calculator, perform 123 + 456, then report the result.
2	Easy	In Finder, go to your Documents folder and create a new plain-text file named experiment_notes.txt.	1	Create ~/Documents/experiment_notes.txt.
3	Easy	Open System Settings → bluetooth and turn it on	1	Turn on bluetooth
4	Easy	Use the macOS screenshot utility to capture the active window	1	Save to ~/Desktop/screenshot.png
5	Easy	Navigate to university website and verify page title contains "Data Science."	1	Visit `https://www.uni-mannheim.de/dws/` and extract title
6	Medium	In Terminal, rename all files by prepending today's date	2	Shell script to rename using YYYY-MM-DD_ format
7	Medium	Write a Python script that computes 123 + 456	2	Create plus.py and write Python program to compute sum
8	Medium	Send an email with subject and body	3	Compose email: Subject = "CS715_Test", Body = "This is a demo."
9	Medium	Log in at saucedemo.com and verify Logout is visible	4	Navigate, login, and confirm logout button visible
10	Medium	Copy data.csv and confirm size match	1	Copy file and compare size using ls -l
11	Medium	Solve CAPTCHA on demo site	3	Go to `2captcha.com/demo/geetest-v4` and solve
12	Medium	Schedule a script to extract news from university site	2	Use cron to run script and parse HTML for <news> content
13	Medium	Fill out and submit a Google Form	4	Load form, fill fields, and submit
14	Medium	Fetch weather using API and save JSON	4	Fetch from `goweather.xyz` and save result on Desktop
15	Medium	Create desktop shortcut (alias) for an app	1	Create alias for [App] on Desktop
16	Hard	Delete a file via Terminal and confirm deletion	1	Use 'rm' and confirm with 'ls'
17	Hard	Extract text from a PDF and export to CSV	5	Use script to extract and convert PDF to CSV
18	Hard	Ping Google 5 times and compute avg RTT	5	Run 'ping', parse RTTs, and average excluding failures
19	Hard	Log CPU/memory every 5s and plot results	5	Script to monitor, plot with matplotlib, and save image
20	Hard	Search "Headphones" on Amazon and report top 3 items	4	Extract name, price, reviews, shipping from top results

Ehrenwörtliche Erklärung

Ich versichere, dass ich die beiliegende Bachelor-, Master-, Seminar-, oder Projektarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und in der untenstehenden Tabelle angegebenen Hilfsmittel angefertigt habe.

Declaration of Used AI Tools

Tool	Purpose	Where?	Useful?
ChatGPT	Rephrasing	Throughout	+
DeepL	Translation	Throughout	-
ResearchGPT	Summarization of related work	Sec. 2	+
GPT-4	Code generation	notebook.ipynb	+
ChatGPT	Related work hallucination	Most of bibliography	++
Claude	Code generation	notebook.ipynb	++

Unterschrift
Mannheim, 14.01.2025