WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

Bachelor Thesis

Towards Agentic LLM frameworks for automating GUI-Tasks: Comparing a single LLM to an Agentic Architecture

Frederik Bauer

Date of Birth: 12.02.2002
Student ID: 12117401

Subject Area: Information Business

Studienkennzahl: 033561

Supervisor: Prof. Dr. Axel Polleres

Date of Submission: 16.07.2025

Department of Information Systems & Operations Management, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR WIRTSCHAFTSINFORMATIK
UND OPERATIONS MANAGEMENT DEPARTMENT
OF INFORMATION SYSTEMS AND OPERATIONS
MANAGEMENT

Contents

List of Figures

Abstract

This thesis investigates whether a hierarchical agent architecture can improve Graphical User Interface (GUI) automation. We compared a manager-worker system, using GPT-4o for high-level planning and UI-TARS-7B-DPO model for low-level execution, against a baseline where UI-TARS-7B-DPO operated as a single end-to-end agent. Both systems were evaluated on a set of tasks from the OSWorld benchmark, measuring Task Success Rate (TSR) and analyzing failure modes.

The hierarchical system achieved a modestly higher overall TSR (23.3% vs. 16.7%), demonstrating a clear advantage on simple, structured tasks. However, this advantage disappeared with increasing complexity; both systems failed all medium difficulty web-based tasks, and the monolithic baseline performed better on hard tasks. Qualitative analysis revealed that even with correct high-level plans, the worker agent consistently failed at fundamental UI interactions like handling web forms and pop-ups.

We conclude that while architectural improvements in planning offer some benefits, they are ultimately undermined by persistent failures in low-level action execution. Robust GUI automation requires foundational improvements in the agent's core interaction capabilities, highlighting that reliable execution is a prerequisite for high-level strategies to be effective.

1 Introduction

Recent advancements in Large Language Models (LLMs) have unlocked unprecedented capabilities in natural language understanding and generation, sparking significant interest in their application to automate complex, human-centric tasks. Automating interactions with Graphical User Interfaces (GUIs) remains a pivotal challenge, traditionally tackled with brittle scripts or complex computer vision [10]. However, the ability of modern LLMs to process multi-modal information and reason about actions has led to the emergence of novel approaches, where models interpret user requests in natural language and directly interact with interface elements to achieve goals [30].

As research into LLM-powered GUI automation accelerates, distinct architectural patterns are surfacing. One emerging approach relies on a specialized single LLM paradigm, where the model directly processes the task description and screen context to generate interaction steps end-to-end. Conversely, agentic architectures adopt a more modular design, frequently employing LLMs within a framework that includes explicit components for planning, task decomposition, tool usage, memory management, and potentially self-correction. These agentic systems aim to mimic more complex reasoning processes to handle intricate, multi-step tasks [24].

Recognizing the limitations inherent in monolithic approaches, recent research increasingly explores compositional or hierarchical frameworks for GUI automation. These agentic systems, which often delegate responsibilities like high-level planning, sub-task execution, and visual grounding to different modules, are demonstrating improved capabilities [3]. While this marks significant progress, the optimal design and interaction patterns within such multi-component systems, particularly concerning the synergy between high-level reasoning models and specialized execution agents, remain an active area of investigation [2].

Building on this direction, this thesis proposes and evaluates a specific hierarchical architecture. We employ a generalist Large Language Model as a high-level manager tasked with decomposing complex user instructions into a sequence of low-level instructions. These instructions are then dispatched to UI-TARS [18], a capable vision-language model specializing in GUI interaction, which acts as the low-level worker. UI-TARS is responsible for the execution of these discrete, state-dependent instructions, potentially leveraging its own inherent planning capabilities for finer-grained actions within the scope of each instruction.

This study will implement this manager-worker framework and evaluate its performance specifically on the OSWorld benchmark [26], using both quantitative metrics and qualitative analysis, contributing insights into effectively combining broad

reasoning abilities with specialized visual grounding and interaction for robust task automation.

1.1 Research question

Stemming from the exploration of hierarchical agent architectures for GUI automation, this thesis aims to address the following research question:

> *Can a hierarchical manager-worker architecture, employing a generalist Large Language Model as a high-level planner and UI-TARS as a specialized low-level worker, improve task automation performance and influence operational behavior for GUI automation tasks on the OSWorld benchmark when compared to a single-agent UI-TARS baseline?*

This will be assessed by:

- Implementing the proposed hierarchical system (GPT-4o planner and UI-TARS-7B-DPO worker) and a baseline system (UI-TARS-7B-DPO operating as a single, end-to-end agent).
- Executing a selected subset of tasks from the Chrome domain of the OSWorld benchmark, covering a spectrum of difficulty levels, with multiple runs for each system to ensure robustness of findings.
- Quantitatively evaluating and comparing the Task Success Rate (TSR) of the hierarchical system against the baseline, with success determined by OSWorld's execution-based validation scripts.
- Qualitatively analyzing the interaction logs from both systems. This will involve a manual review of action sequences, planner-generated instructions (for the hierarchical system), and GUI screenshots to identify and compare operational strategies, common patterns of interaction, and specific failure modes. Failures will be categorized (Planning, Grounding, Interaction, Navigation, Infeasible) to understand the root causes and how they differ between the two architectures.

This comparative evaluation will provide insights into the potential advantages, limitations, and behavioral characteristics of the proposed hierarchical approach in the context of complex GUI automation.

2 Background

The automation of human-computer interaction is being transformed by Graphical User Interface (GUI) agents, particularly those powered by Large Foundation Models. Nguyen et al. (2024) define a GUI agent as "an intelligent autonomous agent that interacts with digital platforms... through their Graphical User Interface. It identifies and observes interactable visual elements displayed on the device's screen and engages with them by clicking, typing, or tapping, mimicking the interaction patterns of a human user" [17]. These agents aim to autonomously navigate and manipulate software applications across diverse platforms by emulating human actions. While related to fields like GUI testing and Robotic Process Automation, which often focus on automating predefined workflows [10], the focus here is on agents capable of dynamic interaction with the GUI environment, often driven by complex user goals or instructions.

2.1 LLM-Powered GUI Automation

The core concept behind LLM-Powered GUI Automation centers on employing Large Foundation Models (LFMs), including Large Language Models (LLMs) and Multimodal Large Language Models (MLLMs), as the intelligent core of the agent [20]. This marks a paradigm shift from conversational chatbots towards using LFMs to perform actions and interact with digital systems via GUIs in a human-like manner. These agents leverage the extensive pre-trained knowledge, reasoning, and multimodal processing capabilities acquired by these models during their general pre-training to interpret user instructions and perceive the GUI environment [24].

Perception occurs through various observations like visual screenshots [18, 9] or textual/structural data [11, 22] (e.g., HTML, DOM, Accessibility Trees). However, reliably bridging this perception-action loop across the sheer diversity of GUIs and task complexities remains a central challenge, driving much of the ongoing research in the field. Based on this understanding, often guided by carefully crafted prompts, the model generates the subsequent action required to progress towards the user's goal. This action might range from low-level mouse/keyboard inputs to interactions with specific UI elements or even the generation of executable code. This approach represents a significant shift, harnessing the general problem-solving abilities of foundation models to handle the complexities and dynamics of GUI interaction across diverse domains [17].

2.2 Origin and Evolution of GUI Automation

The evolution of GUI automation reveals a clear progression towards achieving greater autonomy, flexibility, and generalization, driven by the need to minimize direct hu-

man intervention and overcome the limitations of preceding approaches [18]. This trajectory highlights a shift from rigid, narrowly defined systems to more adaptive and intelligent models.

The origins of GUI automation are deeply embedded in software testing and early Robotic Process Automation (RPA). Initial strategies, such as random 'monkey' testing, script-based record-and-replay (pioneered by tools like Selenium), and rule-based planners [16], sought to mechanize human interaction. These systems treated GUIs primarily as predictable, structural entities. They excelled within constrained environments, automating repetitive validation checks and fixed business processes [15]. However, their fundamental limitation was their brittleness. Lacking semantic understanding or visual adaptability, they frequently broke with minor UI updates or failed entirely when encountering unforeseen states [8]. This reliance on predefined logic and explicit element identifiers highlighted a critical gap: these systems could execute known sequences but could not understand or generalize tasks, necessitating the shift towards more intelligent approaches.

The inherent rigidity of early automation spurred a shift towards embedding intelligence. Machine learning, particularly computer vision, enabled agents to recognize UI elements and adapt to visual changes [5]. Concurrently, early Natural Language Processing allowed rudimentary command interpretation [14], while Reinforcement Learning explored by Lan et al. [12] also showed potential. While these ML-driven approaches offered greater adaptability than their predecessors, they often required extensive task-specific training, struggled with true generalization across diverse applications, and their natural language capabilities remained limited, highlighting the need for more powerful reasoning engines.

The arrival of Large Foundation Models, with their unprecedented language understanding, marked the next major shift, leading to agent frameworks. These systems moved beyond single-purpose ML models, using LLMs as a central 'brain' within modular architectures [30]. Frameworks like OmniParser [30] explicitly combined visual processing and language understanding, orchestrated through carefully designed prompts and workflows. This offered a leap in flexibility and enabled agents to tackle more complex, multi-step tasks. Wang et al. [24] point to the challenges in designing these frameworks and ensuring efficient operation, particularly regarding inference latency. Nguyen et al. [17] underscore the limitations posed by LLM context windows and the inherent difficulty of designing effective prompts, alongside the persistent challenge of reliably grounding actions in diverse GUIs. These developments underscore both the promise and complexity of LLM-centered architectures, highlighting a crucial trade-off between generalization and system efficiency.

The limitations of agent frameworks spurred the development of Native Agent Mod-

els. These systems aim to learn core capabilities like perception, reasoning, and action end-to-end, directly within the model's parameters [18]. Models such as UI-TARS [18] and CogAgent [9] operate primarily on visual GUI screenshots, promising better generalization and reduced manual engineering. These powerful native models can also function as specialized components within compositional frameworks, like Agent S2 [3], which delegates distinct cognitive tasks to various models for enhanced performance. Nevertheless, challenges in developing robust native agents persist, including substantial data needs and the complexities of GUI understanding [18].

2.3 Current Paradigms and Approaches

This evolution from rule-based systems through agent frameworks has led to distinct paradigms currently shaping research in LLM-powered GUI automation. Two prominent approaches stand out: native agent models aiming for end-to-end, data-driven learning, and agentic architectures focusing on compositional or hierarchical structures [20].

2.3.1 Native Agent Models

One major direction involves developing native agent models, which strive for an end-to-end, data-driven architecture where workflow knowledge and core capabilities like perception, reasoning, memory, and action are learned and unified within the model's parameters. This paradigm, aims to improve adaptability and generalization while reducing reliance on handcrafted rules or prompts [18].

Vision Language Models (VLMs) like CogAgent [9], for example, demonstrate strides in native perception by enabling robust recognition of tiny page elements and text crucial for GUI understanding. More comprehensive native agents such as UI-TARS [18] and Aguvis [27] further embody this end-to-end approach, operating directly on visual GUI input to perform human-like interactions. UI-TARS, for instance, integrates several key innovations: enhanced visual perception from large-scale GUI data, unified cross-platform action modeling with precise grounding, reasoning patterns involving explicit intermediate thoughts for planning and reflection.

These capabilities have enabled UI-TARS (72B-DPO) to achieve state-of-the-art results on complex benchmarks like OSWorld [26] (introduced in detail later), scoring 22.7 (15 steps). Similarly, industry models like Anthropic's Claude-Computer-Use also point towards this trend, achieving a score of 14.9 (15 steps) on OSWorld, while Aguvis (72B) scored 10.3 on the same benchmark, showcasing the varying degrees of success in this challenging environment.

2.3.2 Agentic Architectures

Concurrent with the development of native models, an alternative and often complementary paradigm centers on agentic architectures. These systems employ compositional or hierarchical frameworks, strategically delegating different cognitive responsibilities across multiple, potentially specialized, components or models. This approach is motivated by the observation that a single monolithic model, even a powerful generalist one, may not optimally perform all sub-tasks required for complex GUI automation, such as high-level planning, fine-grained visual grounding, and low-level action execution [3]. Wang et al. (2024) propose a generalized framework for such (M)LLM-based GUI agents, typically encompassing distinct components: a GUI perceiver to interpret UI elements, a task planner for decomposing goals, a decision maker to select actions, an executor to interact with the environment, and a memory retriever to leverage past experiences or knowledge [24].

Agentic systems like Agent S2 [3] exemplify this compositional philosophy, utilizing a hierarchical framework with a high-level generalist manager for planning and a low-level worker for execution. Agent S2 achieves 27.0% on the OSWorld benchmark 15-step evaluation. Critically, such architectures can leverage capable native agent models as specialized components, for instance, Agent S2 employs strategies like a 'Mixture of Grounding' where the worker can route actions to specific grounding experts, potentially including highly adept native visual models, to achieve precise element localization [3]. This modularity allows for dynamic plan refinement and aims to overcome challenges in grounding and long-horizon planning by distributing cognitive load. Frameworks like OmniParser [22] and Navi [4] also adopt such modular or hierarchical designs. While offering benefits like the integration of specialized modules and potentially more interpretable reasoning flows, these architectures must manage the complexity of inter-component communication and ensure coherent collaboration between diverse models.

The exploration of optimal designs within these multi-component systems continues. This thesis builds directly upon this trajectory by proposing and evaluating a specific hierarchical manager-worker architecture. We investigate whether employing a generalist Large Language Model as a high-level planner, tasked with decomposing complex instructions, can effectively direct a capable native vision-language model, to improve task automation performance in GUI environments. This research specifically aims to contribute insights into the synergistic potential and operational behaviors when combining broad reasoning abilities with specialized, learned visual grounding and interaction capabilities.

2.4 (M)LLM-based GUI Agent Components

LLM-based GUI agents, particularly those employing agentic or compositional designs, are typically structured around several interacting components or functional areas, each responsible for distinct aspects of the automation task. While the specific identification and naming of these components can vary across different research works and proposed frameworks for instance, generalized models presented by Wang et al. (2024) [24] and Zhang et al. (2024) [30] offer slightly different structural perspectives they generally encompass a similar set of core capabilities. These essential functions include perceiving the state of the graphical user interface, reasoning about the user's goal and planning the necessary steps, executing actions within the environment, and utilizing memory to maintain context and support learning. The following subsections will elaborate on these fundamental components based on common patterns identified in the field.

To make the following abstract components more concrete, we will use a practical running example throughout the following subsections. Imagine a user giving the agent the command: 'In Google Chrome, make Bing the default search engine.'

2.4.1 Perception Module

The GUI perceiver, or perception module, is fundamental to an agent's operation, tasked with interpreting the screen's visual and structural information to build an understanding of the current GUI state [24].

This interpretation often involves processing direct visual input like screenshots, which Multimodal Large Language Models (MLLMs) can increasingly handle. However, reliance on raw visual data presents challenges: Recognizing tiny but crucial elements often requires high-resolution image processing, as demonstrated by models such as CogAgent [9] which utilize specialized encoders for this purpose, but this can lead to significant computational overhead. Additionally, transmitting full sensitive screen data can raise significant privacy concerns and increase the risk of data exposure, making reliable systems a prominent area of recent research [21].

Alternatively, agents can utilize structural data such as HTML/DOM for web applications, or accessibility and widget trees for desktop and mobile platforms, which provide element properties and hierarchy. While these sources can offer precise semantic information, Wang et al. [23] address their limitations, noting that their effectiveness often depends on specific implementations and may be insufficient for custom UI components or dynamic content. Furthermore, raw HTML or DOM trees can be noisy and verbose, often requiring substantial preprocessing or specialized models for effective interpretation [17].

To overcome these individual limitations, hybrid approaches that integrate visual and structural data have also been investigated, although effectively merging these streams presents complexities [31]. Advanced systems, utilizing vision-language foundation parsers like OmniParser V2 [29], aim to generate structured UI representations directly from screenshots. While promising for unifying parsing, such systems can face challenges in precision and interpretability with novel or dense GUIs, alongside issues of computational cost, and potential misinterpretations [30]. Reliably bridging this perception to the agent's reasoning across the sheer diversity of GUIs and task complexities, however, remains a central research challenge.

Example 1 *The Perception Module would process the browser screenshot to locate and identify key UI elements. This includes finding the 'more options' icon, recognizing the 'settings' menu item, and later, on the settings page, identifying the 'search engine' section and the dropdown menu to reveal the list of available search engines.*

2.4.2 Planning Module

Following perception, the planning and reasoning module serves as the agent's cognitive core. Its primary function is to interpret the user's overall goal, analyze the current GUI state provided by the perception module, and determine a coherent strategy and sequence of actions to achieve task completion [24]. A central capability of this module is task decomposition, breaking down complex, high-level user instructions into more manageable sub-tasks or steps. This decomposition is vital for tackling long-horizon problems, a significant challenge for GUI agents [3].

Several strategies facilitate this planning and reasoning process. Techniques like Chain-of-Thought (CoT) prompting [25] encourage the underlying LLM to generate explicit intermediate reasoning steps, ostensibly leading to a more structured plan. However, the efficacy of such pre-generated plans can be limited in highly dynamic GUI environments where each action can alter the state unpredictably. More adaptive approaches, such as the ReAct framework [28], integrate reasoning tightly with action execution and observation cycles, enabling the agent to adjust its plan based on real-time environmental feedback. While ReAct enhances flexibility, it can sometimes lead to inefficient exploration or repetitive loops.

Planning can draw upon the LLM's inherent knowledge or leverage external knowledge bases, such as retrieved documents or tool descriptions, to refine strategies [17]. However, relying on internal knowledge risks using outdated or overly generic information, while effective utilization of external knowledge hinges on accurate and efficient retrieval mechanisms. To address these complexities, UI-TARS incorporates deliberate 'System-2' reasoning, featuring patterns such as task decomposition, re-

flection, and milestone recognition to guide decision-making [18]. Similarly, agentic frameworks like Agent S2 employ proactive hierarchical planning, allowing for dynamic plan refinement at multiple levels in response to evolving observations, rather than only reacting to failures [3]. The challenge remains to develop planners that are robust, efficient, generalizable across diverse tasks, and capable of sophisticated error recovery.

Example 2 *The planning module would decompose the instruction into a logical plan, generating an output like: "(1) Click the 'more options' menu in the top-right corner. (2) Click 'settings' from the menu. (3) Navigate to and click the 'search engine' tab. (4) Click the dropdown menu for the default search engine. (5) Select 'Bing' from the list of options."*

The distribution of planning responsibilities, particularly in hierarchical or compositional systems, mirrors the manager-worker structure investigated in this thesis.

2.4.3 Action Execution Module

Executing actions on a GUI requires translating the agent's plan into specific, often low-level interactions like clicks, key presses, or coordinate-based gestures, going beyond simple text commands or direct API usage. A primary challenge is determining precisely where on the screen to perform the action [24].

One common strategy involves leveraging structured environmental data provided by the platform. For web pages, agents can parse the DOM or HTML, while on desktop or mobile, they might utilize accessibility trees to identify UI elements based on their defined attributes, such as unique IDs or descriptive labels [17]. While effective when this metadata is accurate and complete, this approach can be fragile if the underlying structure is poorly defined, inconsistent, changes unexpectedly, or is simply unavailable for certain UI elements [7].

To overcome these limitations, there is significant research interest in visual grounding techniques. These methods attempt to directly map the agent's intent or a textual description onto the corresponding visual region within a screenshot, relying on understanding the visual appearance of elements rather than their declared structural properties [17]. However, visual grounding presents its own set of difficulties. Precisely localizing elements and determining exact coordinates from raw pixels is challenging due to variability in GUI layouts, dynamic content, and the fine-grained accuracy required [18]. Even advanced Multimodal Large Language Models can struggle with severe hallucination or misinterpreting spatial relationships on complex webpages, making grounding a key bottleneck [31]. Advanced compositional frameworks like Agent S2 propose a 'Mixture of Grounding' mechanism, routing actions to special-

ized experts to improve precision across diverse contexts [3].

Ultimately, accurately grounding the intended actionâ€"whether derived from structural, visual, or hybrid analysisâ€"to the correct screen target is crucial for successful execution. Ongoing research focuses on improving the robustness of these mapping techniques, especially for coordinate-based interactions vital for pure-vision approaches, often through training on large-scale action traces to enhance precision [18].

Example 3 *To execute the plan, the Action Execution module translates a step into a specific interaction. For the command "Select 'Bing' from the list of options," the module must ground it by visually identifying the 'Bing' option (which only became visible after the previous action), determining its precise coordinates, and issuing a click event at that location.*

2.4.4 Memory Module

To effectively handle multi-step tasks and maintain coherence, LLM-based GUI agents require a memory module for storing and retrieving relevant information, enabling statefulness and learning from experience [24]. This is typically categorized into two streams: Short-Term Memory (STM) holds immediate operational data like recent action-observation pairs and the history of prior interactions. Its capacity is often constrained by the LLM's limited context window [30], demanding effective strategies for information prioritization. For instance, UI-TARS retains the full history of previous actions and thoughts as STM but conditions its iterative predictions on a limited window of the last N (typically 5) screen observations to manage this constraint [18]. Long-Term Memory (LTM), conversely, provides persistent storage for accumulating knowledge across sessions, such as successful task trajectories, error patterns, or distilled operational guidelines [30]. Systems like Agent S, for instance, utilize specific narrative and episodic memories for continuous learning through retrieval and self-evaluation [2], while Agent S2 leverages a knowledge base built from prior interaction experiences [3].

However, the effective design and utilization of LTM present substantial research challenges. Retrieval-Augmented Generation (RAG) is common for explicit LTM stores, but ensuring the relevance and efficiency of retrieval from potentially vast or noisy memory, and integrating it without overwhelming the decision-making process, is nontrivial [30]. While some native agent models like UI-TARS aim to implicitly encode experiences within parameters through iterative learning from interaction traces [18], the scalability and generalizability of such implicit memory are still under investigation. Critically, transforming stored experiences into actionable, generalizable knowledge that genuinely enhances problem-solving beyond mere context repetition,

and deciding what to store or forget, remains a key research frontier for robust agent learning and adaptation.

Example 4 *Short-Term Memory is essential for maintaining context. The agent's action-observation history reminds it that after clicking 'settings', its goal is to find the 'search engine' link, preventing it from getting lost. Long-Term Memory could be leveraged if the agent has learned from prior tasks that browser settings are typically accessible via a 'three-dot menu', allowing it to initiate the task more efficiently.*

2.5 Evaluation of LLM-based GUI Agents

Evaluating LLM-powered GUI agents is critical not only for gauging performance across different dimensions but also for identifying weaknesses, guiding continuous improvement, and ensuring alignment with user expectations. The evaluation landscape involves assessing agents based on the diverse outputs they produce during task execution, such as action sequences, intermediate GUI states (often captured as screenshots or structural data), and the final environment state after task completion [30].

Key metric categories are commonly employed, though specific names and precise calculations can vary between studies. Task completion metrics, such as Task Success Rate (TSR), are primary for determining overall effectiveness [20]. Additionally, efficiency metrics (e.g., number of steps, time taken), generalization metrics (performance on unseen tasks/websites/domains), safety metrics (adherence to policies), and robustness metrics (e.g., to environmental distractions) are increasingly important for a holistic assessment [30].

To perform these evaluations consistently and enable comparison between different agents, the field relies heavily on standardized benchmarks. These benchmarks vary in several aspects, including the platform (web, mobile, desktop/OS), interactivity (static datasets versus dynamic, interactive environments), and world assumptions (closed-world versus open-world settings where external knowledge might be required) [30]. Prominent examples include web-focused environments like Mind2Web [6], which emphasizes generalist agents on real-world websites with diverse tasks and interaction patterns, and WebArena, which provides a highly realistic and reproducible environment with fully functional websites and focuses on evaluating the functional correctness of long-horizon tasks [32]. For mobile-centric evaluations, AndroidWorld offers a dynamic benchmarking environment [19].

OSWorld [26], the benchmark used in this thesis, marks a major step forward in evaluating agents on real-world computer tasks. Unlike earlier benchmarks that relied on static data and rigid evaluations, OSWorld offers a scalable, execution-based en-

vironment across Ubuntu, Windows, and macOS. It supports diverse, realistic tasks involving actual applications, file operations, and multi-app workflows. With 369 tasks and 134 custom evaluation scripts, it emphasizes outcome-based assessment over fixed action sequences, enabling fair evaluation of multiple valid approaches. Initial tests showed LLM/VLM agents perform poorly (best at 12.24% vs. humans at 72.36%), revealing significant gaps in GUI interaction and practical knowledge.

3 Methodology

This section details the methodology employed to evaluate the effectiveness of a specific manager-worker hierarchical architecture designed for automating complex interactions within Graphical User Interfaces (GUIs). The primary goal was to assess the performance of this proposed two-level system, comprising a high-level generalist Large Language Model (LLM) manager for task decomposition and the specialized UI-TARS agent acting as a low-level instruction executor. This evaluation was conducted using tasks from the OSWorld benchmark, and the hierarchical system's performance was quantitatively and qualitatively compared against a baseline configuration where the UI-TARS agent attempted the same tasks operating as a single, end-to-end system.

3.1 Prototypical System Design

To evaluate the research question, two distinct systems were designed. This section details the prototypical design of these two systems. The primary system is a hierarchical manager-worker architecture that distributes the cognitive load required for complex tasks between a high-level planner and a low-level executor. To assess its performance, this hierarchical system was compared against a baseline system where the worker agent attempts the same tasks operating as a single, end-to-end system. The following will elaborate on the specific architecture and operational flow of each configuration.

3.1.1 Hierarchical Manager-Worker System

The core of this research involved the implementation and evaluation of a hierarchical manager-worker architecture specifically designed for GUI automation within the OSWorld environment. As illustrated in Figure 1, this system distributes the cognitive load required for complex task completion between two distinct components. A high-level manager component is responsible for strategic planning and task decomposition, breaking down complex user goals into simpler steps. These steps are then delegated to a low-level worker component responsible for grounded execution and direct interaction with the graphical user interface.

For the high-level planner role, we utilized GPT-4o, a state-of-the-art generalist Large Language Model. The manager receives the overall task instruction and the current screen context as input. Its primary function, guided by specific prompting, is to decompose this high-level goal into a sequence of lower-level, state-dependent instructions suitable for the worker agent. The definition and granularity of these low-level instructions drew inspiration from the types of actions used for UI control agents

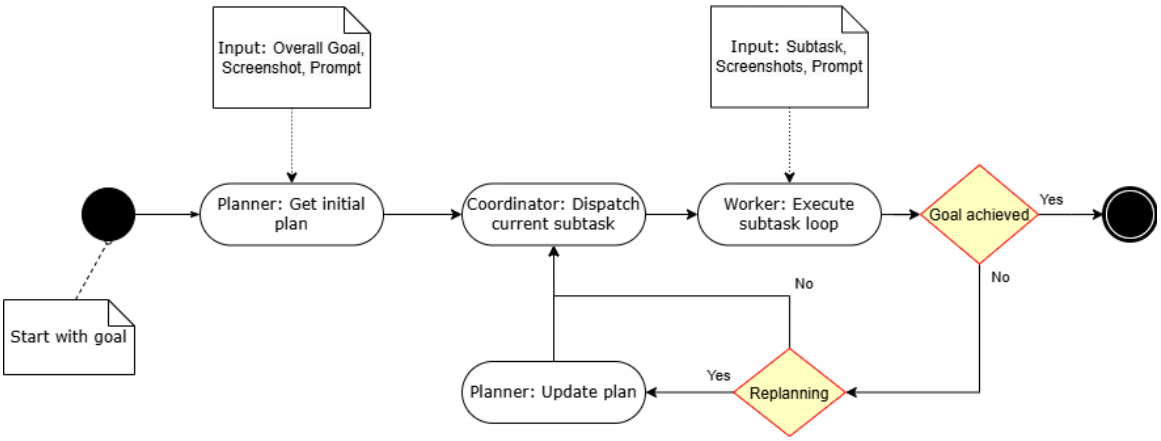

Figure 1: Hierarchical Manager-Worker Flow-Diagram

described by Li et al. (2024) [13], aiming for concrete sub-tasks like clicking specific elements, typing text, or navigating menus. The planner generates these instructions sequentially, planning the steps needed to progress towards the final objective. The output instructions are formatted as simple natural language commands like 'click the button labeled save', or 'open the Chrome menu'.

The low-level worker component was instantiated using UI-TARS-7B-DPO. UI-TARS, which operates under the direction of the planner, receives one low-level instruction at a time, along with the current GUI screenshot. Its role is to interpret this specific instruction and execute the corresponding low-level actions. This leverages UI-TARS's established capabilities in visual grounding and generating fine-grained keyboard and mouse operations. A slightly adapted version of the standard Bytedance prompt template [1] for computer interaction was employed to focus the agent on executing the single directive provided by the manager.

The interaction protocol is structured around an initial full-plan generation phase followed by sequential execution with periodic planning intervention. Upon receiving the task and initial state, the planner first generates a complete sequence of low-level instructions anticipated to fulfill the entire task goal. This full plan is then stored. Execution proceeds sequentially: a dispatcher provides the UI-TARS worker with the next instruction from this pre-generated plan after the previous step is attempted. To account for potential deviations or unexpected GUI changes, a periodic re-planning mechanism is triggered. After a set number of instructions from the plan have been dispatched and attempted by the worker, the planner is reactivated. At this checkpoint, the planner analyzes the current screen state. Based on this feedback, it

assesses progress against the original plan and generates a new, potentially adjusted, sequence of instructions for the remaining steps required to complete the task. This approach leverages the efficiency of upfront planning while incorporating necessary grounding and course correction based on the actual state observed at regular intervals or upon encountering errors.

3.1.2 The Baseline System

To effectively assess the performance and characteristics of the proposed hierarchical manager-worker architecture, a direct comparison against a relevant baseline is necessary (Figure 2).

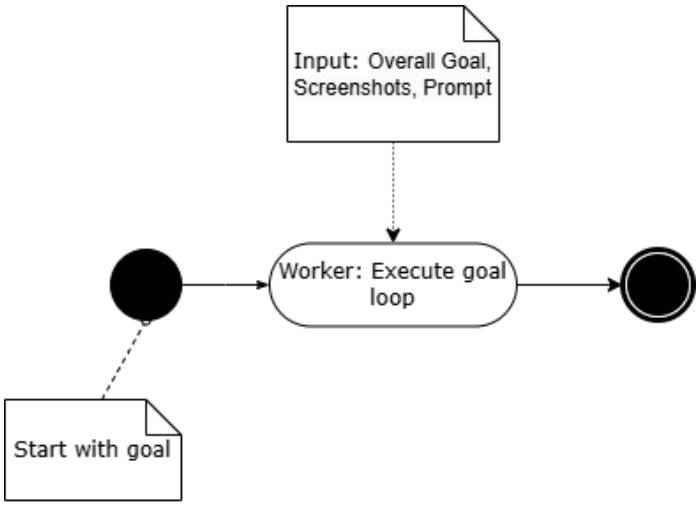

Figure 2: Baseline Flow-Diagram

For this purpose, the baseline system employed the UI-TARS-7B-DPO model as a single, end-to-end agent. This configuration represented the standard usage pattern for such a capable vision-language GUI agent. In the baseline setup, UI-TARS was directly provided with the same high-level task instructions from the OSWorld benchmark that were given to the planner in the hierarchical framework. The standard UI-TARS prompt template for computer interaction tasks, as described in [1], was used. This approach provided a benchmark to measure the relative effectiveness, success rates, and behavioral differences introduced by the explicit hierarchical structure compared to a monolithic agent approach.

3.2 Experiment Setup

The empirical evaluation compared a hierarchical agentic architecture with a baseline system, both operating within the OSWorld task environments. The experimental setup utilized a distributed configuration. The OSWorld virtual machine environment, where tasks were executed, ran on a local laptop. To ensure that each trial concluded within a reasonable timeframe, a maximum limit of 15 interaction steps was imposed on each task attempt. If an agent failed to complete its objective within this limit, the run was terminated and recorded as a failure.

Large Language Models (LLMs) used:

- Hierarchical architecture:
  - The planner component utilized OpenAI's Gpt-4o model, accessed via its external API.
  - The worker component employed the UI-TARS-7B-DPO model. This instance was hosted and executed on a separate dedicated server, with its specific setup detailed in Appendix 7.3.
- Baseline system:
  - This system utilized the UI-TARS-7B-DPO model operating as a single agent. This instance was also hosted and executed on the same dedicated server, configured as described in Appendix 7.3.

To ensure the meaningfulness of the results and account for potential variability, each of the 10 selected tasks was executed three times for each system (the hierarchical architecture and the baseline). The findings from these multiple runs were then used for the comparative analysis within this study.

It is important to note that the scope of the quantitative and qualitative analysis detailed in the Results chapter is strictly defined by this experimental setup. These experiments were designed to compare the two architectures using the specified model versions, which were representative of the state of the art when the study was initiated. Further preliminary experiments using newer model versions are discussed later in the Discussion chapter to provide context on the rapid evolution of the field.

3.2.1 Task Selection

The tasks for the empirical evaluation were selected from the OSWorld benchmark. This benchmark provides realistic Ubuntu virtual machine environments and a diverse

set of tasks designed to emulate real-world computer usage, often involving multiple applications and OS-level interactions. [26]

This evaluation was conducted on a selected subset comprising 10 distinct tasks, all chosen from the Chrome domain. This focus was deliberate: the Chrome domain offers a rich and highly complex environment, providing a rigorous testbed for agent performance due to its diverse UI elements, dynamic content, and multi-step processes. Furthermore, selecting tasks exclusively from Chrome allowed for an evaluation across a clear spectrum of easy, medium, and hard difficulty levels within a consistent application context. This approach facilitates a nuanced assessment of how each architecture scales to increasing task complexity without the confounding variable of switching between different application environments.

Importantly, all tasks in the selected pool were chosen such that a feasible and well-defined evaluator function existed, enabling reliable and automated assessment of task success.

The task difficulty was ranked as follows: easy tasks typically involved direct manipulation of Chrome's built-in settings or straightforward, single-action interactions with minimal steps and specific, verifiable outcomes. Medium tasks required a sequence of steps, such as navigating a website, searching, applying filters, or filling forms, resulting in a specific URL or page state reflecting the applied criteria. Hard tasks demanded a higher level of reasoning and more complex interactions, potentially involving deep navigation, information extraction and synthesis from various parts of a page, or interacting with less structured interfaces and more open-ended goals. A detailed list of the selected tasks can be found in Appendix 7.2.

3.3 Evaluation Metrics

To evaluate the performance of the hierarchical manager-worker system relative to the baseline and address the research questions, this study employed both quantitative and qualitative evaluation metrics.

3.3.1 Quantitative Metric

The primary quantitative metric for evaluating task completion effectiveness and comparing the two systems was the Task Success Rate (TSR). Success for each task attempt was determined strictly based on the execution-based evaluation mechanism inherent to the OSWorld benchmark. OSWorld provides task-specific validation scripts that programmatically verify if the final state of the virtual machine environment meets the required success criteria for that task [26]. A task attempt was deemed successful only if these conditions were met upon termination of the agent's

execution. The TSR for each system was calculated as the percentage of the 10 selected tasks that were successfully completed according to these criteria.

3.3.2 Qualitative Analysis

To provide deeper insights into the operational behavior, common strategies, and failure modes of each system, a qualitative analysis was performed. This analysis involved a manual review of the interaction logs generated during task execution for both the hierarchical and baseline systems. These logs include the sequence of actions performed by the system, the intermediate instructions generated by the planner, and screenshots depicting the GUI state.

For unsuccessful task attempts, failures were categorized based on their root cause using a framework adapted from the error analysis categories presented for the Agent S2 framework by Agashe et al. (2025) [3]. This involved classifying errors into the following types:

- Planning: errors originating from flawed high-level strategy or incorrect decomposition of the task. For the hierarchical system, this could be poor instructions from the manager, for the baseline, an illogical path chosen.
- Grounding: failure to correctly map an instruction or intention to the correct UI element on the screen (e.g. clicking the wrong button, unable to find an element).
- Interaction: errors occurring during the physical execution of an action, even if planning and grounding were correct (e.g. mistyping text, incorrect mouse drag).
- Navigation: difficulties specifically related to navigating the UI structure, such as improper scrolling or getting lost in menus/dialogs.
- Infeasible: cases where task completion is prevented by factors outside the agent's core logic, such as inherent limitations, tool errors, or benchmark environment issues.

Successful executions were also reviewed qualitatively to identify and compare the strategies employed by the hierarchical system versus the baseline end-to-end agent. This comparative analysis aimed to illuminate the practical advantages and disadvantages of the manager-worker approach.

4 Results

This section presents the findings from the comparative evaluation of the baseline system and the hierarchical planner-worker system on a selected subset of 10 tasks from the Chrome domain of the OSWorld benchmark. Each task was executed three times by each system, resulting in 30 attempts per system. The results are presented first through quantitative analysis of Task Success Rates (TSR), followed by a detailed qualitative analysis of the systems' operational behaviors, common strategies, and failure modes derived from interaction logs.

Both systems used pyautogui for action execution and screenshot-based observation. Performance was primarily measured by Task Success Rate (TSR), determined by OSWorld's execution-based validation scripts. Qualitative analysis of interaction logs, including agent thoughts, planner instructions (for the hierarchical system), and action sequences, was performed to understand the operational characteristics and root causes of successes and failures.

4.1 Quantitative Performance Analysis

This section presents the quantitative findings from the comparative evaluation of the baseline system and the hierarchical planner-worker system. Quantitative performance was primarily assessed using the Task Success Rate (TSR), determined by OSWorld's execution-based validation scripts.

4.1.1 Overall Task Success Rates (TSR)

Figure 3 presents the average TSR across all three runs of the 10 tasks for each system in the OSWorld benchmark.

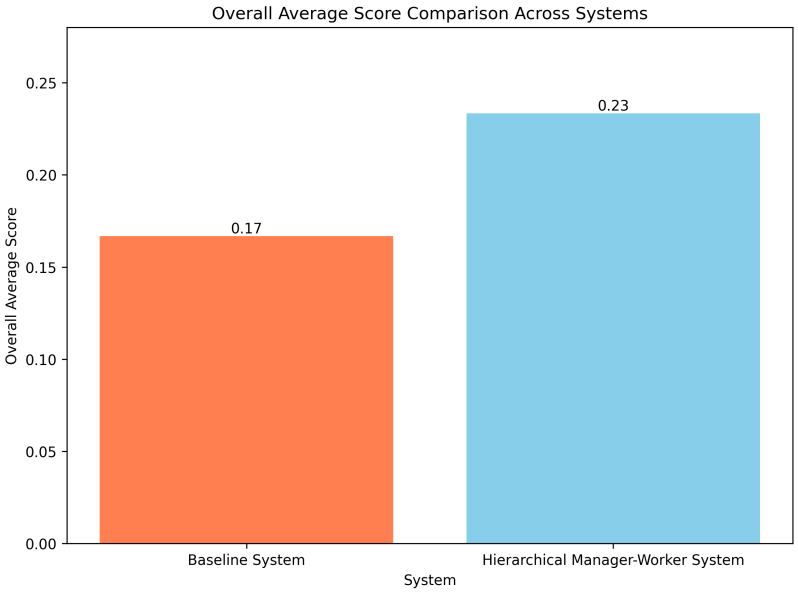

Figure 3: Overall average score comparison across systems

- The baseline system achieved 5 successful completions, resulting in an overall TSR of 16.7%.
- The hierarchical system achieved 7 successful completions, resulting in an overall TSR of 23.3%.

The hierarchical system demonstrated a moderately higher overall TSR by 6.6 percentage points compared to the baseline system.

4.1.2 Task Success Rates by Task Difficulty

The 10 selected tasks were categorized into easy (5 tasks), medium (3 tasks), and hard (2 tasks) difficulty levels. The performance of each system across these categories is detailed in Figure 4.

The hierarchical system achieved a notably higher TSR on easy tasks (46.7%) compared to the baseline system (20.0%). Neither system was successful in completing any of the medium difficulty tasks across any of the runs. The baseline system demonstrated some success on hard tasks (33.3%), while the hierarchical system did not succeed on any hard tasks.

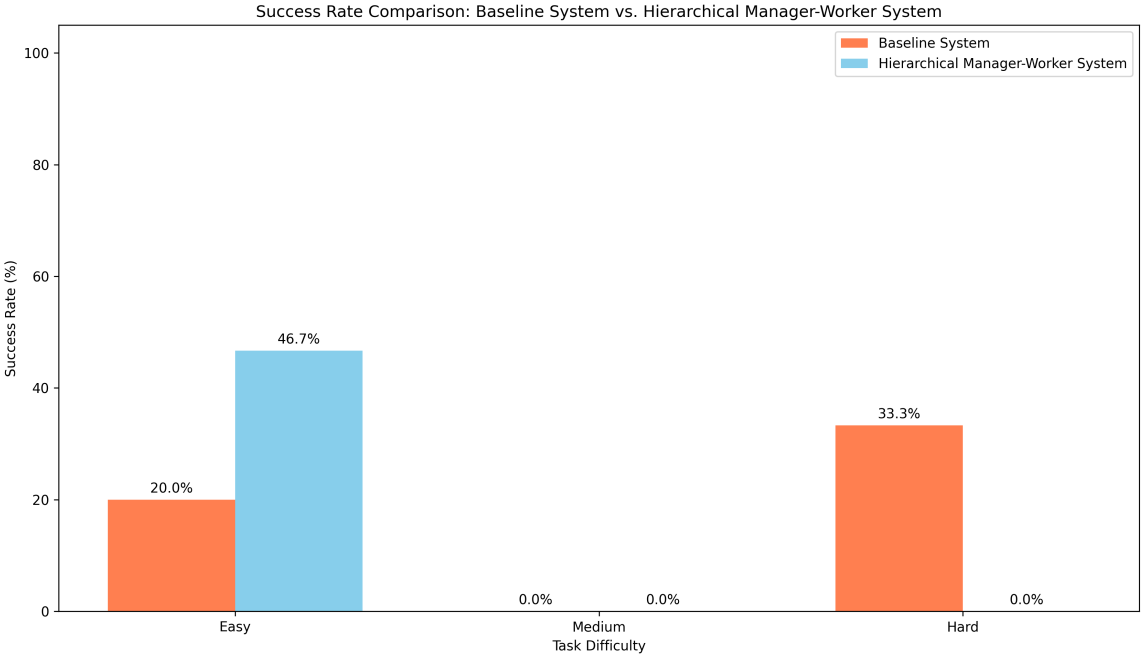

Figure 4: Success rate comparison: baseline System vs. hierarchical manager-worker system

4.1.3 Consistency of Task Success

Analysis of individual task success across the three runs reveals variations in consistency:

- Task 9656a811 (enable safety feature - easy): the baseline system was highly consistent, succeeding in 3/3 runs. The hierarchical system succeeded in 1/3 runs for this task.
- Task bb5e4c0d (set bing search - easy): the hierarchical system was highly consistent, succeeding in 3/3 runs. The baseline system failed in all 3 runs.
- Task 6766f2b8 (install extension - hard): The baseline system succeeded in 2/3 runs. The hierarchical system failed in all 3 runs.
- Other successes for the hierarchical system on easy tasks were single instances (1/3 runs) for 'enable DNT' (030eeff7), 'new bookmarks folder' (2ad9387a), and 'save bookmark' (7a5a7856).
- All medium difficulty tasks (1704f00f - car rental, 6c4c23a1 - flights, cabb3bae - spider-man toys) and one Hard task (121ba48f - Dota 2 DLC) were failed by both systems in all 3 runs

4.2 Qualitative Analysis

Beyond the quantitative Task Success Rates, a detailed qualitative analysis was performed on the complete set of interaction logs from all 60 experimental runs. This analysis aimed to provide deeper insights into the operational behaviors, common strategies, error patterns, and specific failure modes exhibited by each system when attempting the GUI automation tasks. The review involved a manual review of the recorded agent thoughts, action sequences, planner-generated instructions and plans (for the hierarchical system), and inferred UI states.

Failures were categorized based on their primary root cause according to the framework established in the methodology, which includes planning, grounding, interaction, navigation, and infeasible. Following a detailed review and classification, it was determined that all primary failure causes within this study's dataset could be attributed to planning, interaction, or navigation errors. Consequently, while 'grounding' and 'infeasible' remained part of the analytical framework as potential failure types, no instances were ultimately assigned to these two categories in the presented results, failures that might initially seem like grounding or infeasibility were found to have more dominant root causes within the other three categories upon closer inspection of the agent's decision-making process or task constraints.

4.2.1 Comparative Failure Mode Analysis

The distribution of primary failure causes across the observed categories for unsuccessful task attempts provides insights into the distinct vulnerabilities of each system. Figure 5 presents a summary of these failure distributions.

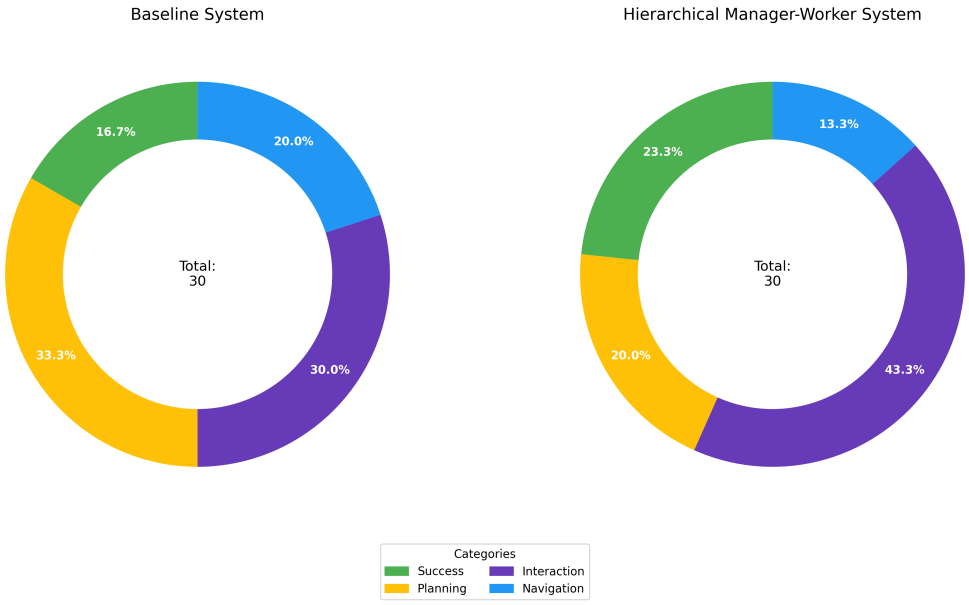

Figure 5: Task outcome distribution by system

Planning Failures

Planning failures encompassed issues in the agent's high-level strategy, logical sequencing of actions, understanding of task requirements, or premature termination.

Baseline system: planning failures were often severe and fundamental.

- A notable pattern was catastrophic context loss. In these cases, the agent's internal 'thought' process switched to Chinese, and it pursued entirely unrelated goals.
- The baseline system frequently demonstrated an incomplete or flawed internal procedure for specific sub-tasks.
- Premature DONE calls were common across various tasks, indicating a flawed assessment of task completion.

Hierarchical system: planning failures manifested differently, often related to the manager's strategic decisions or its interaction with the worker.

- The planner sometimes generated flawed high-level strategies or introduced detrimental steps.
- Ineffective re-planning was a common issue.
- The planner also made incorrect assumptions about the UI state or worker actions, leading to inefficient and confusing instruction sequences.
- Ambiguity in planner instructions sometimes contributed to failure.

Comparison: baseline planning failures often represented a fundamental lack of a coherent strategy or catastrophic context loss. Hierarchical planning failures were more about suboptimal high-level decisions, poor adaptation by the planner to worker struggles, or miscommunications/misassumptions within the manager-worker dynamic.

Interaction Failures

Interaction failures refer to instances where the agent, having a seemingly reasonable plan or intention for an element, failed in the mechanical execution of an action or in handling dynamic UI behaviors.

Baseline system: interaction failures were a significant impediment.

- The agent consistently struggled with pop-ups that lead to loops.
- Executed actions that navigated away from the task.
- Difficulties were observed in reliably typing into, clearing, or selecting from dropdowns associated with web form fields.
- The agent often got stuck in repetitive click loops if an element did not respond as expected.

Hierarchical system: The worker component exhibited similar interaction struggles, even under the guidance of the planner.

- When the planner directed the worker to a page with a pop-up, the worker often failed to handle it effectively, getting stuck in click loops.
- A primary bottleneck for the hierarchical system was the worker's difficulty in reliably executing instructions on specific web UI elements. This was evident in all medium tasks where, despite planner instructions to type in fields, select from dropdowns, or click sort buttons, the worker frequently failed these low-level interactions, leading to planner re-planning loops.

Comparison: both systems were highly susceptible to interaction failures with pop-ups. The hierarchical system's failures highlighted that even with a correct high-level plan and specific low-level instructions, the worker (UI-TARS) still faced significant challenges in reliably interacting with many dynamic web UI elements. The baseline's interaction failures often compounded its planning/navigation issues more directly.

Navigation Failures

Navigation failures involved the agent getting lost, taking illogical paths between UI states, or being unable to effectively use UI structures like menus or scroll functions to reach necessary elements.

Baseline system: navigation was a frequent failure point.

- The agent often got lost within website structures or even within Chrome's settings .
- Following an error or unexpected UI state, the agent's navigation often became chaotic, attempting to restart the browser or interacting with OS-level elements.

Hierarchical system: While the planner often provided a logical navigation sequence, failures still occurred.

- Worker navigation errors: the worker sometimes failed to correctly execute a navigation instruction or lost its context within a planned sequence.
- Planner-Induced navigation issues: In some cases, the planner's flawed re-planning led to confused navigation.

Comparison: The baseline system exhibited more frequent and severe macro-level navigation collapses. The hierarchical system's navigation failures were often more about the worker failing to accurately follow a specific planned navigation step or the planner getting stuck in a re-planning loop related to navigation when the worker was confused.

4.2.2 Analysis of Operational Strategies

Baseline System Strategies

The baseline UI-TARS system generally operated in a reactive, step-by-step manner.

- Successful strategies: on tasks where its internal model aligned well with a straightforward UI flow, such as 'enable safety feature' (3/3 successes) and 'install extension' (2/3 successes), it could be quite direct and efficient. For 'install

extension,' it demonstrated two different valid initial navigation strategies across successful runs, indicating some flexibility.

- Characteristic failure strategies: when faced with ambiguity, complex UIs, or unexpected elements (especially pop-ups), its strategy often devolved into:
  - Repetitive action loops: persistently clicking unresponsive elements or re-typing into fields.
  - Chaotic recovery attempts: if stuck, it might try seemingly random actions, like going to the OS desktop, opening new tabs, or re-typing entire task queries into inappropriate places.
  - Premature task abandonment: frequently calling DONE without completing the task or after only a few erroneous steps.
  - A lack of a discernible long-term plan was evident.

The hierarchical system's strategies were characterized by the interplay between the GPT-4o planner and the UI-TARS worker.

- Impact of planner-generated instructions:
  - Task decomposition: the planner generally provided logical, multi-step decompositions. For settings-based easy tasks like these plans were often precise and led to success when the worker executed them faithfully.
  - Clarity and actionability: instructions were typically simple natural language commands. However, their effectiveness was contingent on the worker's ability to execute them. Some instructions were occasionally ambiguous or based on incorrect assumptions about the UI state by the planner.
  - Problematic plan segments: instances of flawed planning included the injection of irrelevant goals or inefficient strategies occured.
- Worker (UI-TARS) execution patterns under hierarchical guidance:
  - Following discrete instructions: in successful runs, the worker demonstrated it could accurately follow the sequence of low-level instructions from the planner, especially within the more predictable Chrome settings UI.
  - Struggles despite specific instructions: a key observation was the worker's repeated difficulty with dynamic web UI elements even when the planner's

instruction for that specific element was clear. This was a major cause of failure in all Medium tasks.

  - Execution errors: the worker sometimes failed to execute all parts of a given instruction or made navigational errors while attempting to follow a plan segment.

- Effectiveness of the re-planning mechanism:

  - Adaptive attempts: the planner frequently re-planned when sub-tasks were not completed as expected. This demonstrated an attempt at adaptive behavior. In one instance ('save bookmark'), a series of different re-planned strategies eventually led to success.

  - Ineffective loops: more commonly, re-planning was ineffective, especially if the root cause was a persistent worker interaction/grounding failure or if the planner itself was confused about the state. The planner often re-issued similar instructions or slight variations that didn't address the core problem, leading to extended failure loops.

The operational differences between the two systems can be comprehensively characterized by the following key aspects, which highlight distinctions in planning strategy, error handling, efficiency, and constraint management.

- Planning approach: the hierarchical system exhibited a more structured, proactive planning approach due to the manager's decomposition, while the baseline was largely reactive.

- Resilience to errors: the baseline system had minimal error recovery. The hierarchical system attempted recovery via re-planning; this was sometimes successful for simpler errors or by eventually stumbling upon a working strategy but often failed to resolve persistent worker execution issues or deeper planner misconceptions.

- Efficiency: when the baseline succeeded (e.g., 'install extension'), its path could be very direct. Hierarchical successes sometimes involved more steps due to the planner's verbosity or re-planning detours ('save bookmark'). However, for some settings tasks, the planner's direct multi-step plan was more efficient than the baseline's fumbling.

- Handling of task constraints: The hierarchical planner was sometimes better at explicitly encoding all task constraints (e.g., selecting 'bookmarks bar' in its plan for 'save bookmark'), though the worker didn't always execute these perfectly. The baseline often missed such specific constraints in its internal

planning.

4.2.3 Impact of Task Complexity on Qualitative Behavior

The operational behaviors and failure types manifested differently across task complexities:

- Easy tasks:
  - The hierarchical system generally showed an advantage, achieving a 46.7% TSR versus the baseline's 20.0%. Successes for the hierarchical system often resulted from the planner providing a correct sequence for navigating Chrome's native UI, which the worker could follow. However, planner errors or worker execution/navigation errors still led to failures.
  - The baseline system was only consistently successful on 'enable safety feature.' On other easy tasks, it struggled with incomplete internal procedures or suffered catastrophic planning failures.
- Medium tasks:
  - Both systems failed all 9 attempts each (0% TSR). This was a critical area of shared difficulty.
  - Baseline failures were typically characterized by an inability to handle web pop-ups, leading to navigation collapse, getting stuck in extended WAIT states, or chaotic and illogical action sequences.
  - Hierarchical failures also involved pop-up issues. However, a dominant pattern was the worker's inability to reliably interact with web form elements or specific web page controls, even when the planner provided targeted instructions. This often led to the planner getting stuck in re-planning loops, trying to reissue instructions the worker could not execute.
- Hard tasks:
  - The baseline system showed capability here, succeeding on 'install extension' in 2/6 attempts (33.3% TSR). Its direct, although sometimes variable, approach proved effective when its internal model aligned with the UI flow for this complex task.
  - The hierarchical system failed all 6 attempts on hard tasks (0% TSR).
  - For 'install extension,' failures were often due to planner confusion regarding the 'developer mode' state, which led to the worker incorrectly toggling

the switch multiple times, or other subtle errors despite the worker sometimes performing the core file navigation steps correctly. The coordination overhead seemed detrimental.

5 Discussion

This chapter interprets the findings presented in the results section, placing them within the context of the initial research question and the broader field of LLM-powered GUI automation. We will discuss the implications of the observed performance differences between the hierarchical and baseline systems, analyze the dominant failure modes in relation to existing literature, acknowledge the limitations of this study, and propose directions for future research.

5.1 Interpretation of Key Findings

The central research question asked whether a hierarchical manager-worker architecture could improve task automation performance and influence operational behavior compared to a single-agent baseline. Our results provide a nuanced answer: while the hierarchical system achieved a higher overall Task Success Rate (TSR) (23.3% vs. 16.7%), its effectiveness was highly dependent on task complexity and type.

The overall 6.6% improvement in TSR suggests that introducing a dedicated high-level planner (GPT-4o) can offer tangible benefits. This was most evident in easy tasks, where the hierarchical system significantly outperformed the baseline (46.7% vs. 20.0%). This aligns with the premise that explicit task decomposition, as proposed in agentic frameworks like Agent S2 [3], can effectively guide agents through structured, multi-step processes where the UI is relatively predictable. The planner's ability to generate a logical sequence provided a clear roadmap that the UI-TARS worker could, in these simpler cases, execute more reliably than the baseline UI-TARS operating end-to-end, which sometimes exhibited catastrophic planning failures or lacked the specific procedural knowledge.

Counterintuitively, the baseline system outperformed the hierarchical system on hard tasks (33.3% vs. 0%), driven entirely by its partial success on 'install extension'. This suggests that for certain complex tasks, the overhead, potential for planner misinterpretation, or the rigidity of the initial plan in the hierarchical system might hinder performance. The baseline system, while generally less reliable, occasionally demonstrated a more direct, albeit potentially less 'reasoned,' path to success, possibly leveraging implicit knowledge learned during its pre-training [18]. This finding implies that a hierarchical structure is not universally superior and can introduce its own set of challenges, particularly when planner-worker coordination or planner state understanding becomes critical.

The complete failure (0% TSR) of both systems on medium tasks is perhaps the most notable finding. It strongly indicates that low-level interaction and grounding,

particularly within dynamic web environments, remains a fundamental bottleneck, as highlighted by multiple researchers [17, 31, 18]. Even when the hierarchical system's planner provided arguably correct, high-level instructions, the UI-TARS worker consistently failed to execute these actions reliably. This underscores a critical point: sophisticated planning is rendered ineffective if the execution module cannot reliably interact with the target UI elements. This suggests that improvements in native agent capabilities are paramount, regardless of the overarching architectural design.

Qualitatively, the hierarchical system demonstrated a more structured, albeit sometimes flawed, approach. Its failures were often due to ineffective re-planning or worker execution issues, rather than the catastrophic context loss seen in the baseline. This indicates that the manager-worker structure does influence behavior, promoting a more decomposable and potentially more interpretable, though not necessarily more successful process. The re-planning mechanism, while intended to add adaptiveness, often devolved into loops when faced with persistent worker failures, highlighting the need for better feedback and error-handling mechanisms between components.

5.2 Connection to Existing Work

This study provides a direct, albeit limited, comparison. While the agentic (hierarchical) approach showed a slight overall edge, its struggles, particularly on medium/hard tasks, reinforce the idea that the performance of such systems is deeply intertwined with the capabilities of their constituent native components (the worker) [3]. It supports the notion that modularity offers benefits in planning [24], but doesn't inherently solve core perception and interaction challenges [18].

While our planner (GPT-4o) could often generate logical plans, its 'open-loop' nature (planning many steps ahead, with only periodic checks) proved brittle. This echoes the need for more adaptive planning cycles, perhaps closer to the ReAct framework [28], or the proactive hierarchical planning seen in Agent S2 [3], where plans can be adjusted more dynamically based on real-time feedback.

Our results illustrate the persistence of interaction and grounding failures, especially on web GUIs, a challenge noted across numerous studies [17, 31]. The medium task failures strongly suggest that even with a powerful VLM like UI-TARS, reliably clicking, typing, and selecting elements on dynamic web pages remains a significant hurdle. This reinforces the importance of research into robust visual grounding.

The overall low TSRs for both systems (below 25%) align with the initial OSWorld findings [26], confirming that navigating real-world operating systems and applications remains a formidable challenge for current AI agents, far exceeding performance on more constrained or web-only benchmarks.

5.3 Contextualizing Results with Evolving Agent Capabilities

To place the primary findings of this thesis in the context of the rapidly evolving field of GUI agents, a single, exploratory test run was conducted. This initial test re-evaluated the two architectures on the same 10-task set, but with the worker and baseline agent updated to the more capable UI-TARS-1.5-7B [1]. This newer version represents a leap from the foundational model, incorporating advanced reasoning and planning capabilities that lead to state-of-the-art performance. It must be emphasized that the following observations are drawn from this single run and are therefore anecdotal, serving only to suggest potential trends rather than establish conclusive results.

The outcome of this single run was indicative and suggested a potential performance inversion. In this run, the new baseline using UI-TARS-1.5-7B as a single end-to-end agent achieved an overall Task Success Rate of 40%, successfully completing three easy tasks and one hard task. In contrast, the hierarchical system, in this instance, saw its performance stagnate to a 20% TSR, managing only two easy tasks.

This observed performance is an interesting, though anecdotal, finding. It raises the possibility that as a specialized 'worker' agent becomes more proficient, the high-level instructions from a generalist 'manager' could introduce what might be termed 'architectural friction' or 'coordination overhead'. In this specific run, the planner's instructions appeared to conflict with the worker's more refined execution strategy, and the simpler approach of the standalone agent seemed superior.

In summary, while drawn from a single exploratory run and thus highly preliminary, these observations lend tentative support to the hypothesis that for many standard GUI tasks, the rapid evolution of specialized, end-to-end models is making them a more efficient solution. The role for hierarchical systems may be shifting towards exceptionally complex problems that lie beyond the scope of even these improved native agents, a direction that requires much more extensive testing to verify.

5.4 Limitations

It is important to acknowledge the limitations of this study, which temper the generalizability of its findings:

- Limited task scope: the evaluation used only 10 tasks, all within the Chrome domain. Performance could differ significantly across other applications and a wider range of task types available in OSWorld.
- Small number of runs: executing each task only three times per system provides indicative results but lacks statistical robustness. The observed variability in

success (e.g., 1/3 vs. 3/3 successes on the same task) suggests that more runs are needed for definitive conclusions.

- Maximum step constraint: all experimental runs were terminated after a maximum of 15 steps. This constraint, while common in agent benchmarking for tractability, may have influenced the outcome. It could have prematurely ended solution paths that were correct but required more than 15 actions to complete. Consequently, this study does not evaluate the agents' performance or reasoning abilities on longer-horizon tasks, and the reported success rates might not be representative of scenarios that demand more extensive interaction.
- Model versioning and technological snapshot: the main study's quantitative results are based on UI-TARS-7B-DPO. As demonstrated in the preliminary exploratory run discussed in Section 5.3, newer agent versions like UI-TARS-1.5-7B possess significantly different, and in some cases superior, capabilities. This makes the primary findings of this thesis a technological snapshot, valid for the specific component versions tested at a particular point in time. The conclusions about the hierarchical system's advantages, therefore, cannot be generalized to newer or future agent generations without extensive further testing.
- Prompt engineering: the performance of both systems, especially the planner, is sensitive to prompt design, which was not a focus of systematic variation in this study.
- Fixed re-planning strategy: the periodic re-planning might not be optimal, event-triggered or confidence-based re-planning could be more effective.
- Qualitative analysis subjectivity: while structured, the categorization of failures involves a degree of subjective interpretation.

5.5 Future Research Directions

Based on the findings of this study, its limitations, and the current research landscape, we propose two primary and interconnected directions for future research:

Enhancing native agent interaction and robustness: this direction focuses on strengthening the core capabilities of the low-level worker agent, addressing the critical execution bottleneck observed in the medium difficulty tasks where both systems failed. Future work must prioritize improving the model's ability to reliably interact with complex and dynamic UI elements. This involves developing more robust visual grounding and interaction techniques, creating larger and more diverse training datasets focused on interaction traces, and designing specialized modules or strategies for handling

common failure points like pop-up dismissal or intricate form filling. Successfully advancing the capabilities of these native GUI models is a prerequisite for overcoming the fundamental execution challenges that currently limit all higher-level architectural designs.

Advancing hierarchical architectures for complex reasoning and recovery: building on this study and existing work like Agent S2 [3], this direction aims to improve the intelligence and resilience of the overall agentic framework. This includes exploring more adaptive planning and recovery mechanisms that move beyond the rigid re-planning used here. A key aspect is enhancing planner-worker communication, enabling richer feedback so the planner can better understand why an instruction failed.

Furthermore, the provocative result of our exploratory test, where a newer monolithic agent outperformed the hierarchical, calls for a deeper investigation of the long-term value of such architectures. Future work must conduct larger-scale studies that directly address the limitations of this thesis by expanding beyond a small task subset and, crucially, removing the restrictive 15-step horizon to properly evaluate long-horizon reasoning. A systematic evaluation is required to discover the limit at which hierarchical decomposition offers a definitive and enduring advantage over increasingly capable monolithic agents.

6 Conclusion

This thesis demonstrated that a hierarchical manager-worker architecture can offer a modest improvement in overall GUI automation performance compared to a single-agent baseline, primarily by imposing a logical structure on simpler, multi-step tasks. However, it also revealed that this architectural advantage is fragile and easily negated by fundamental limitations in low-level UI interaction, particularly within dynamic web environments where both systems failed. Our findings underscore that achieving robust GUI automation requires a dual focus on high-level planning and core interaction capabilities. The optimal balance between these elements is a moving target, especially as the rapid evolution of single, end-to-end models continually redefines the trade-offs between architectural complexity and performance. The path forward, therefore, lies not merely in stacking components, but in co-designing synergistic systems where strategic foresight and adaptive, reliable execution are tightly integrated.

7 Appendix

7.1 Github Repository

All source code, implementation scripts, and related materials developed for this thesis are publicly available in a GitHub repository. The repository can be accessed at the following URL:

`https://github.com/Fredibau/agenticLlmFrameworksForAutomatingGuiTasks`

7.2 Taskset

Easy Tasks

- **ID:** bb5e4c0d-f964-439c-97b6-bdb9747de3f4
  Description: Can you make Bing the main search thingy when I look stuff up on the internet?
- **ID:** 030eeff7-b492-4218-b312-701ec99ee0cc
  Description: Can you enable the 'Do Not Track' feature in Chrome to enhance my online privacy?
- **ID:** 2ad9387a-65d8-4e33-ad5b-7580065a27ca
  Description: Can you make a new folder for me on the bookmarks bar in my internet browser? Let's call it 'Favorites.'
- **ID:** 7a5a7856-f1b6-42a4-ade9-1ca81ca0f263
  Description: Can you save this webpage I'm looking at to bookmarks bar so I can come back to it later?
- **ID:** 9656a811-9b5b-4ddf-99c7-5117bcef0626
  Description: I want Chrome to warn me whenever I visit a potentially harmful or unsafe website. Can you enable this safety feature?

Medium Tasks

- **ID:** 1704f00f-79e6-43a7-961b-cedd3724d5fd
  Description: Find a large car with lowest price from next Monday to next Friday in Zurich.
- **ID:** 6c4c23a1-42a4-43cc-9db1-2f86ff3738cc
  Description: Find flights from Seattle to New York on 5th next month and only show those that can be purchased with miles.

- **ID:** cabb3bae-cccb-41bd-9f5d-0f3a9fecd825
  Description: Browse spider-man toys for kids and sort by lowest price.

Hard Tasks

- **ID:** 121ba48f-9e17-48ce-9bc6-a4fb17a7ebba
  Description: Find Dota 2 game and add all DLC to cart.
- **ID:** 6766f2b8-8a72-417f-a9e5-56fcaa735837
  Description: Could you help me install the unpacked extension at /home-/user/Desktop/ in Chrome?

7.3 UI-TARS-7B Server Setup

This section details the setup procedure for hosting the UI-TARS-7B model on a self-configured server, which was the method utilized for this thesis. This option provides greater control over the deployment environment, whether using a cloud virtual machine or a local machine.

Hardware Configuration For the 7B model, an `RTX 6000 Ada` or a GPU with similar capabilities is recommended to ensure sufficient performance for inference.

Environment Setup A CUDA-enabled environment is required. While any appropriately configured CUDA environment should suffice, the setup used in this work was modeled after the following RunPod environment:

- **RunPod PyTorch 2.4.0 Environment (Example):**
  - **Image:** `runpod/pytorch:2.4.0-py3.11-cuda12.4.1-devel-ubuntu22.04`
  - **Includes:**
    - CUDA 12.4.1
    - PyTorch 2.4.0
    - Python 3.11

Setup Steps The following steps outline the process for setting up the server:

1. **Install Hugging Face CLI tools:**
   Install the `huggingface_hub` package, which is used to download the model.

```
pip install -U huggingface_hub
```

2. **Download the 7B model:**
   Use the Hugging Face CLI to download the `ByteDance-Seed/UI-TARS-7B-DPO` model. Replace `/path/to/your/model` with the desired local directory on your server.

```
huggingface-cli download ByteDance-Seed/UI-TARS-7B-DPO --
   local-dir /path/to/your/model
```

3. **Create the `preprocessor_config.json` file:**
   Create or modify this configuration file within the model directory specified above (`/path/to/your/model`). The content should be as follows:

```
cat > /path/to/your/model/preprocessor_config.json << 'EOL'
{
  "do_convert_rgb": true,
  "do_normalize": true,
  "do_rescale": true,
  "do_resize": true,
  "image_mean": [
    0.48145466,
    0.4578275,
    0.40821073
  ],
  "image_processor_type": "Qwen2VLImageProcessor",
  "image_std": [
    0.26862954,
    0.26130258,
    0.27577711
  ],
  "max_pixels": 2116800,
  "merge_size": 2,
  "min_pixels": 3136,
  "patch_size": 14,
  "processor_class": "Qwen2VLProcessor",
  "resample": 3,
  "rescale_factor": 0.00392156862745098,
  "size": {
    "shortest_edge": 1080,
    "longest_edge": 1920
  },
  "temporal_patch_size": 2
}
EOL
```

Note: The ‘cat ... EOL‘ lines are a shell command to create the JSON file. The content between ‘‘ and ‘‘ is the JSON itself.

4. **Install `vllm`:**
   Install the `vllm` library (version 0.6.6 was used in this work) for efficient inference.

   ```
   pip install vllm==0.6.6
   ```

5. **Start the server:**
   Run the `vllm` OpenAI-compatible API server. Ensure the `-model` argument points to your model directory. A port (e.g., `4000`) must also be specified.

   ```
   python -m vllm.entrypoints.openai.api_server \
       --served-model-name ui-tars \
       --model /path/to/your/model \
       --trust-remote-code \
       --port 4000 \
       --limit-mm-per-prompt image=5
   ```

Upon successful execution of the final step, an API server will be active, allowing interaction with the UI-TARS-7B model for inference tasks.

References

[1] GitHub - bytedance/UI-TARS — github.com. `https://github.com/bytedance/UI-TARS`. [Accessed 14-04-2025].

[2] Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S: An open agentic framework that uses computers like a human. *arXiv preprint arXiv:2410.08164*, 2024.

[3] Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent S2: A compositional generalist-specialist framework for computer use agents. *arXiv preprint arXiv:2504.00906*, 2025.

[4] Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, et al. Windows agent arena: Evaluating multi-modal os agents at scale. *arXiv preprint arXiv:2409.08264*, 2024.

[5] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1535–1544, 2010.

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

[7] Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents. *arXiv preprint arXiv:2410.05243*, 2024.

[8] Theodore D Hellmann and Frank Maurer. Rule-based exploratory testing of graphical user interfaces. In *2011 Agile Conference*, pages 107–116. IEEE, 2011.

[9] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290, 2024.

[10] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J Brostow. Help, it looks confusing: Gui task automation through demonstration and follow-up questions. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, pages 233–243, 2017.

[11] Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36:39648–39677, 2023.

[12] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. Deeply reinforcing android gui testing with deep reinforcement learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[13] Wei Li, William E Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on ui control agents. *Advances in Neural Information Processing Systems*, 37:92130–92154, 2024.

[14] Sahisnu Mazumder and Oriana Riva. Flin: A flexible natural language interface for web navigation. *arXiv preprint arXiv:2010.12844*, 2020.

[15] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. Dart: a framework for regression testing "nightly/daily builds" of gui applications. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 410–419, 2003.

[16] A.M. Memon, M.E. Pollack, and M.L. Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001.

[17] Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, et al. Gui agents: A survey. *arXiv preprint arXiv:2412.13501*, 2024.

[18] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.

[19] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.

[20] Pascal J Sager, Benjamin Meyer, Peng Yan, Rebekka von Wartburg-Kottler, Layan Etaiwi, Aref Enayati, Gabriel Nobel, Ahmed Abdulkadir, Benjamin F Grewe, and Thilo Stadelmann. Ai agents for computer use: A review of

instruction-based computer control, gui automation, and operator assistants. *arXiv preprint arXiv:2501.16150*, 2025.

[21] Yucheng Shi, Wenhao Yu, Wenlin Yao, Wenhu Chen, and Ninghao Liu. Towards trustworthy gui agents: A survey. *arXiv preprint arXiv:2503.23434*, 2025.

[22] Jianqiang Wan, Sibo Song, Wenwen Yu, Yuliang Liu, Wenqing Cheng, Fei Huang, Xiang Bai, Cong Yao, and Zhibo Yang. Omniparser: A unified framework for text spotting key information extraction and table recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15641–15653, 2024.

[23] Bryan Wang, Gang Li, and Yang Li. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2023.

[24] Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, et al. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*, 2024.

[25] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[26] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

[27] Yiheng Xu, Zekun Wang, Junli Wang, Dunjie Lu, Tianbao Xie, Amrita Saha, Doyen Sahoo, Tao Yu, and Caiming Xiong. Aguvis: Unified pure vision agents for autonomous gui interaction. *arXiv preprint arXiv:2412.04454*, 2024.

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

[29] Wenwen Yu, Zhibo Yang, Jianqiang Wan, Sibo Song, Jun Tang, Wenqing Cheng, Yuliang Liu, and Xiang Bai. Omniparser v2: Structured-points-of-thought for

unified visual text parsing and its generality to multimodal large language models. *arXiv preprint arXiv:2502.16161*, 2025.

[30] Chaoyun Zhang, Shilin He, Jiaxu Qian, Bowen Li, Liqun Li, Si Qin, Yu Kang, Minghua Ma, Guyue Liu, Qingwei Lin, et al. Large language model-brained gui agents: A survey. *arXiv preprint arXiv:2411.18279*, 2024.

[31] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.

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