UFO^2: The Desktop AgentOS

Anonymous authors
Paper under double-blind review

Abstract

Recent *Computer-Using Agents* (CUAs), powered by multimodal large language models (LLMs), offer a promising direction for automating complex desktop workflows through natural language. However, most existing CUAs remain conceptual prototypes, hindered by shallow OS integration, fragile screenshot-based interaction, and disruptive execution.

We present **UFO^2**, a multiagent *AgentOS* for Windows desktops that elevates CUAs into practical, system-level automation. UFO^2 features a centralized HostAgent for task decomposition and coordination, alongside a collection of application-specialized AppAgents equipped with native APIs, domain-specific knowledge, and a unified GUI–API action layer. This architecture enables robust task execution while preserving modularity and extensibility. A hybrid control detection pipeline fuses Windows UI Automation (UIA) with vision-based parsing to support diverse interface styles. Runtime efficiency is further enhanced through speculative multi-action planning, reducing per-step LLM overhead. Finally, a *Picture-in-Picture* (PiP) interface enables automation within an isolated virtual desktop, allowing agents and users to operate concurrently without interference.

We evaluate UFO^2 across over 20 real-world Windows applications, demonstrating substantial improvements in robustness and execution accuracy over prior CUAs. Our results show that deep OS integration unlocks a scalable path toward reliable, user-aligned desktop automation.

1 Introduction

Automation of desktop applications has long been central to improving workforce productivity. Commercial Robotic Process Automation (RPA) platforms such as UiPath UiPath (2025), Automation Anywhere Anywhere (2025), and Microsoft Power Automate Microsoft (2025) exemplify this trend, using predefined scripts to replicate repetitive user interactions through the graphical user interface (GUI) Hofmann et al. (2020); Madakam et al. (2019). However, these script-based approaches often prove fragile in dynamic, continuously evolving environments Pramod (2022). Minor interface changes can break the underlying automation scripts, requiring manual updates and extensive maintenance effort. As software ecosystems grow increasingly complex and heterogeneous, the brittleness of script-based automation severely limits scalability, adaptability, and practicality.

Recently, *Computer-Using Agents* (CUAs) Zhang et al. (2024a) have emerged as a promising alternative. These systems leverage advanced multimodal large language models (LLMs) Zhao et al. (2023); Zhang et al. (2024d) to interpret diverse user instructions, perceive GUI interfaces, and generate adaptive actions (*e.g.*, mouse clicks, keyboard inputs) without fixed scripting Zhang et al. (2024a). Early prototypes such as UFO Zhang et al. (2024b), Anthropic Claude Anthropic (2024), and OpenAI Operator OpenAI (2025) demonstrate that such LLM-driven agents can robustly automate tasks too complex or ambiguous for conventional RPA pipelines. Yet, despite these advances, current CUA implementations remain largely *conceptual*: they mainly optimize visual grounding or language reasoning Wang et al. (2024b); Qin et al. (2025); OpenAI (2025); Zheng et al. (2025); Niu et al. (2024), but give little attention to *system-level* integration with desktop operating systems (OSs) and application internals (Figure 1 (a)).

Reliance on raw GUI screenshots and simulated input events has several drawbacks. First, visual-only inputs can be noisy and redundant, increasing cognitive overhead for LLMs and reducing execution efficiency Zhang

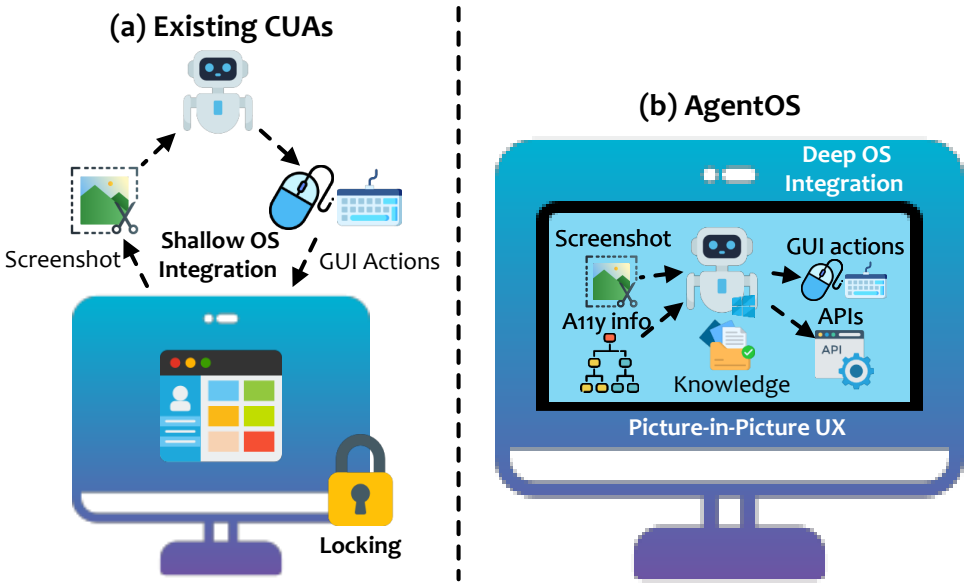

Figure 1: A comparison of (a) existing CUAs and (b) desktop AgentOS UFO^2.

et al. (2025). Second, existing CUAs rarely leverage the OS's native accessibility interfaces, application-level APIs, or detailed process states—a missed opportunity to significantly enhance decision accuracy, reduce latency, and enable more reliable execution. Finally, simulating mouse and keyboard events on the primary user desktop locks users out during automation, imposing a poor user experience (UX). These limitations must be resolved before CUAs can mature from intriguing prototypes into robust, scalable solutions for real-world desktop automation, and motivates our central research question:

> *How can we build a robust, deeply integrated system for desktop automation that flexibly adapts to evolving interfaces, reliably orchestrates diverse applications, and minimizes disruption to user workflows?*

In response, we present **UFO^2**, a new *AgentOS* for Windows that reimagines desktop automation as a first-class operating system abstraction. Unlike prior CUAs that treat automation as a layer atop screenshots and simulated input events, UFO^2 is architected as a deeply integrated, multiagent execution environment—embedding OS capabilities, application-specific introspection, and domain-aware planning into the core automation loop, as illustrated in Figure 1(b).

At its foundation, UFO^2 provides a modular, system-level substrate for natural language-driven automation. A centralized coordinator, the HostAgent, interprets user instructions, decomposes them into semantically meaningful subtasks, and dynamically dispatches execution to specialized AppAgents —expert modules tailored for specific Windows applications. Each AppAgent is equipped with an extensible toolbox of application-specific APIs, a hybrid GUI–API action interface, and integrated knowledge about the application's capabilities and semantics. This architecture enables robust orchestration across multiple concurrent applications, supporting workflows that span Excel, Outlook, Edge, and beyond.

To enable reliable execution across the full spectrum of application UIs, UFO^2 introduces a hybrid control detection pipeline, combining Windows UI Automation (UIA) APIs with advanced visual grounding models Lu et al. (2024). This allows agents to introspect and act on both standard and custom UI components, bridging the gap between structured accessibility trees and pixel-level perception. Moreover, UFO^2 continuously incorporates external documentation, patch notes, and past execution traces into a unified vectorized memory layer, enabling each AppAgent to incrementally refine its behavior without retraining.

At the interaction layer, UFO^2 exposes a unified GUI–API execution model, where agents seamlessly combine traditional GUI actions (*e.g.*, clicks, keystrokes) with native Windows or application-specific APIs via configurable Model Context Protocol (MCP) servers Hou et al. (2025). This hybrid approach improves execution efficiency, reduces brittleness to UI layout changes, and enables more expressive, higher-level operations. To further minimize the latency associated with LLM-based action planning, UFO^2 incorporates a speculative multi-action execution engine that proactively infers and validates action sequences using lightweight control-state checks at a single inference step—substantially reducing inference overhead without compromising correctness.

Finally, to ensure a practical and non-intrusive user experience, UFO^2 introduces a novel Picture-in-Picture (PiP) interface: a secure, nested desktop environment where agents can execute independently of the user's main session. Built atop Windows' native remote desktop loopback infrastructure, PiP enables seamless, side-by-side user–agent multitasking without disruption on the user's main desktop, addressing one of the most persistent UX limitations of existing CUAs.

Together, these design principles position UFO^2 not just as a smarter agent, but as a new OS-level abstraction for automation—transforming desktop workflows into programmable, composable, and robust entities. In summary, this paper makes the following contributions:

- **Deep OS Integration:** We design and implement UFO^2, a multiagent AgentOS that deeply embeds within the Windows OS, orchestrating desktop applications through introspection, API access, and fine-grained execution control.
- **Unified GUI–API Action Layer:** We propose a hybrid action interface that bridges traditional GUI interactions with application-native API calls via configurable MCP servers, enabling flexible, efficient, and robust automation.
- **Hybrid Control Detection:** We introduce a fusion pipeline combining UIA metadata with vision-based detection to achieve reliable control grounding even in non-standard interfaces.
- **Continuous Knowledge Integration:** We build a retrieval-augmented memory that integrates documentation and historical execution logs, allowing agents to improve autonomously over time without retraining.
- **Speculative Multi-Action Execution:** We reduce LLM invocation overhead by predicting and validating action sequences ahead of time using UI state signals.
- **Non-Disruptive UX:** We develop a nested virtual desktop environment that allows automation to proceed in parallel with user activity, avoiding interference and improving adoptability.
- **Comprehensive Evaluation:** We evaluate UFO^2 across 20+ real-world Windows applications, showing consistent improvements in success rate, execution efficiency, and usability over state-of-the-art CUAs like Operator.

Overall, UFO^2 advances the vision of OS-native automation by shifting the paradigm from GUI scripting to structured, programmable application control. Even when paired with general-purpose models like GPT-4o, UFO^2 outperforms dedicated CUAs by over 10%, highlighting the transformative impact of system-level integration and architectural design.

2 Background

2.1 The Fragility of Traditional Desktop Automation

For decades, desktop automation has relied on brittle techniques to replicate human interactions with GUI-based applications. Commercial RPA (Robotic Process Automation) tools—such as UiPath UiPath (2025), Automation Anywhere Anywhere (2025), and Microsoft Power Automate Microsoft (2025)—operate by recording and replaying mouse movements, keystrokes, or rule-based scripts. These systems rely heavily

on surface-level GUI cues (*e.g.*, pixel regions, window titles), offering little introspection into application state.

While widely deployed in enterprise settings, traditional RPA systems exhibit poor robustness and scalability Siderska et al. (2023). Even minor UI updates—such as reordering buttons or relabeling menus—can silently break automation scripts. Maintaining correctness requires frequent human intervention. Furthermore, these tools lack semantic understanding of application workflows and cannot reason about or adapt to novel tasks. As a result, RPA tools remain constrained to narrow, repetitive workflows in stable environments, far from general-purpose automation.

2.2 Rise of Computer-Using Agents

Recent advances in large language models (LLMs) and multimodal perception have enabled a new class of automation systems, referred to as *Computer-Using Agents* (CUAs) Zhang et al. (2024a); Zhao et al. (2023); Zhang et al. (2024d). CUAs aim to generalize across applications and tasks by leveraging LLMs to interpret user instructions, perceive GUI layouts, and synthesize actions such as clicks and keystrokes. Early CUAs like UFO Zhang et al. (2024b) demonstrated that multimodal models (*e.g.*, GPT-4V Yang et al. (2023b)) could map natural language requests to sequences of GUI actions with no hand-crafted scripts. More recent industry prototypes, including Claude-3.5 (Computer Use) Anthropic (2024) and OpenAI Operator OpenAI (2025), have pushed the envelope further, performing realistic desktop workflows across multiple applications.

These CUAs represent a promising evolution from static RPA scripts to adaptive, general-purpose automation. However, despite their sophistication, current CUAs largely remain research prototypes, constrained by architectural and systems-level limitations that impede real-world deployment.

2.3 Systems Challenges in CUAs

Current CUAs fall short in three fundamental ways, which we argue stem from missing operating system abstractions:

(1) Lack of OS-Level Integration. Most CUAs interact with the system through screenshots and low-level input emulation (mouse and keyboard events). They ignore rich system interfaces such as accessibility APIs, application process state, and native inter-process communication mechanisms (*e.g.*, shell commands, COM interfaces Gray et al. (1998)). This superficial interaction model limits reliability and efficiency—every action must be inferred from pixels rather than structured state.

(2) Absence of Application Introspection. CUAs typically operate as generalists with limited awareness of application-specific capabilities. They treat all interfaces uniformly, lacking the ability to leverage built-in APIs or vendor documentation. As a result, they cannot reason over high-level concepts unless such flows are explicitly embedded in the model. This rigidity limits their generalization and makes maintenance expensive.

(3) Disruptive and Unsafe Execution Model. Most CUAs drive automation directly on the user's desktop session, hijacking the real mouse and keyboard. This design prevents users from interacting with their system during execution, introduces interference risk, and violates isolation principles fundamental to safe system design. Long-running tasks—especially those involving multiple LLM queries—can monopolize the session for minutes at a time.

2.4 Missing Abstraction: OS Support for Automation

Despite growing demand for intelligent, language-driven automation, existing operating systems offer no first-class abstraction for exposing GUI application control to external agents. In contrast to system calls, files, or sockets, GUI workflows remain opaque and non-programmable. As a result, both RPA and CUA systems are forced to operate as ad-hoc layers atop the GUI, with no unified substrate for execution, coordination, or introspection.

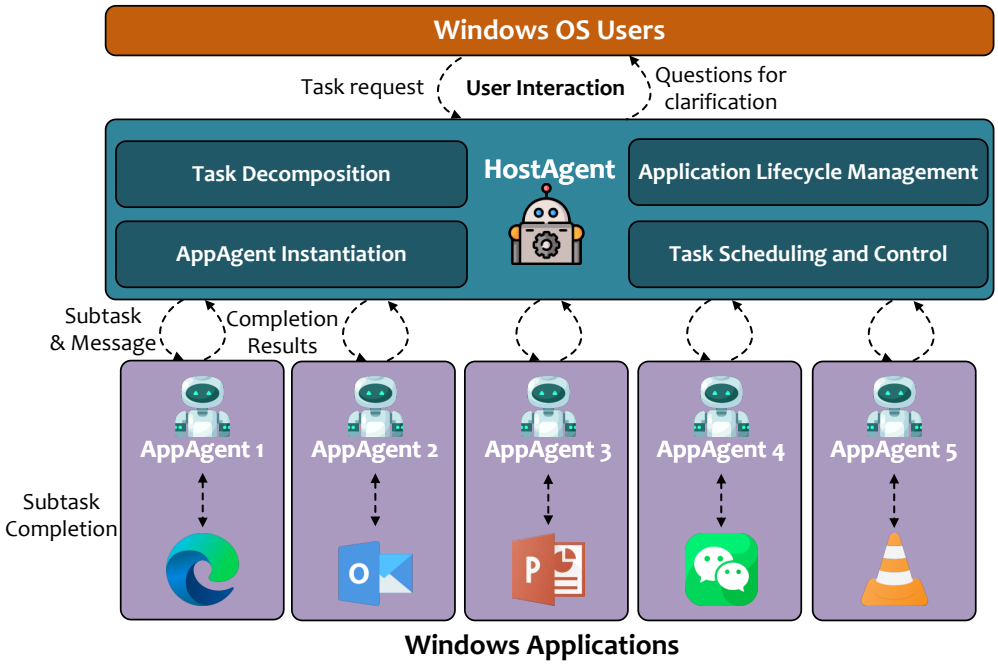

Figure 2: An overview of the architecture of UFO^2.

This paper argues that automation should be elevated to a system primitive. We present **UFO^2**, a new *AgentOS* for Windows that addresses these limitations by embedding automation as a deeply integrated OS abstraction—exposing GUI controls, application APIs, and task orchestration as programmable, inspectable, and composable system services.

3 System Design of UFO^2

Motivated by the challenges highlighted in Section 2, UFO^2 is designed to seamlessly interpret natural language user requests and reliably automate tasks across a wide range of Windows applications. This section provides an architectural overview of UFO^2 (Section 3.1) and explains how each component is deeply integrated with the underlying OS to overcomes the pitfalls of current CUAs, ultimately enabling a practical, robust AgentOS for desktop automation.

3.1 UFO^2 as a System Substrate for Automation

Figure 2 presents the high-level architecture of UFO^2, which provides a structured runtime environment for task-oriented automation on Windows desktops. Deployed as a local daemon, UFO^2 enables users to issue natural language requests that are translated into coordinated workflows spanning multiple GUI applications. The system provides core abstractions for orchestration, introspection, control execution, and agent collaboration—exposing these as system-level services analogous to those in traditional OSes.

At the heart of UFO^2 is a central control plane, the HostAgent, responsible for parsing user intent, managing system state, and dispatching subtasks to a collection of specialized runtime modules called AppAgents. Each AppAgent is dedicated to a particular application (*e.g.*, Excel, Outlook, File Explorer) and encapsulates all logic needed to observe and control that application, including API bindings, UI detectors, and knowledge bases. These modules act as isolated execution contexts with application-specific semantics.

Upon receiving a user request, HostAgent decomposes it into a series of subtasks, each mapped to the application best suited to fulfill it. If the corresponding application is not already running, HostAgent launches it using native Windows APIs and instantiates the corresponding AppAgent. Execution proceeds through a structured loop: each AppAgent continuously observes the application state (via accessibility

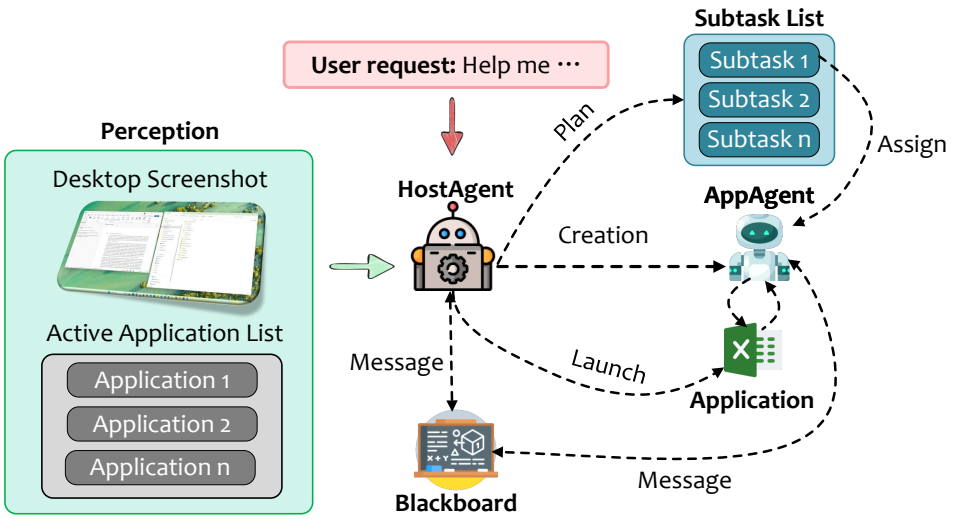

Figure 3: High-level architecture of HostAgent as a control-plane orchestrator.

APIs and vision-based detectors), reasons about the next operation using a ReAct-style planning cycle Yao et al. (2023), and invokes the appropriate action—either a GUI event or a native API call. This loop continues until the subtask terminates, either successfully or due to an unrecoverable error.

UFO^2 implements shared memory and control flow via a global blackboard interface, allowing HostAgent and AppAgents to exchange intermediate results, dependency states, and execution metadata. This architecture supports complex workflows across application boundaries—for instance, extracting data from a spreadsheet and using it to populate fields in a web form—without requiring hand-crafted scripts or coordination logic. Crucially, all interactions occur within a virtualized, PiP-based desktop environment, ensuring process-level isolation and safe multi-application concurrency.

Design Rationale: Centralized Multiagent Runtime. UFO^2 adopts a centralized multiagent Zhang et al. (2024b); Qiao et al. (2023); Han et al. (2024); Wang et al. (2024a) runtime to support both reliability and extensibility. The centralized HostAgent acts as a control plane, simplifying task-level orchestration, error handling, and lifecycle management. Meanwhile, each AppAgent is architected as a loosely coupled executor that encapsulates deep, application-specific functionality.

Modularity at the AppAgent level allows developers and third-party contributors to incrementally expand UFO^2's capabilities by authoring new application interfaces and API bindings. These agents are discoverable, self-contained, and dynamically instantiated by the runtime as needed. From a security and evolvability perspective, this separation of concerns ensures that application logic can evolve independently of the core task orchestration engine.

Together, the HostAgent –AppAgent model allows UFO^2 to function as a scalable, pluggable runtime substrate for GUI automation—abstracting away the complexity of heterogeneous interfaces and providing a unified system interface to structured application behavior.

3.2 HostAgent: System-Level Orchestration and Execution Control

The HostAgent serves as the centralized control plane of UFO^2. It is responsible for interpreting user-specified goals, decomposing them into structured subtasks, instantiating and dispatching AppAgent modules, and coordinating their progress across the system. HostAgent provides system-level services for introspection, planning, application lifecycle management, and multi-agent synchronization.

Figure 3 outlines the architecture of HostAgent. Operating atop the native Windows substrate, HostAgent monitors active applications, issues shell commands to spawn new processes as needed, and manages

the creation and teardown of application-specific AppAgent instances. All coordination occurs through a persistent state machine, which governs the transitions across execution phases.

Responsibilities and Interfaces. HostAgent exposes the following system services:

- **Task Decomposition.** Given a user's natural language input, HostAgent identifies the underlying task goal and decomposes it into a dependency-ordered subtask graph.
- **Application Lifecycle Management.** For each subtask, HostAgent inspects system process metadata (via UIA APIs) to determine whether the target application is running. If not, it launches the program and registers it with the runtime.
- **AppAgent Instantiation.** HostAgent spawns the corresponding AppAgent for each active application, providing it with task context, memory references, and relevant toolchains (*e.g.*, APIs, documentation).
- **Task Scheduling and Control.** The global execution plan is serialized into a finite state machine (FSM), allowing HostAgent to enforce execution order, detect failures, and resolve dependencies across agents.
- **Shared State Communication.** HostAgent reads from and writes to a global blackboard, enabling inter-agent communication and system-level observability for debugging and replay.

System Perception and Introspection. To perform its control functions, HostAgent fuses two layers of system introspection:

1. **Visual Layer.** Captures pixel-level screenshots of the desktop workspace, enabling coarse-grained layout understanding.
2. **Semantic Layer.** Queries Windows UIA APIs to extract structural metadata about applications, windows, and control hierarchies.

This dual perception enables HostAgent to resolve ambiguities, detect runtime inconsistencies, and guide agents with context-aware decisions.

Structured Output Interface. HostAgent produces structured outputs to drive downstream execution:

- *Subtask Plan:* A high-level execution plan detailing decomposed subtasks.
- *Shell Command:* A sequence of shell-level invocations for managing application lifecycles.
- *Assigned Application:* The process name and index of the application selected for instantiating the AppAgent, which will be used to execute the next subtask.
- *Agent Messages:* Context-specific instructions passed to AppAgent instances for localized execution.
- *User Prompts:* Interactive clarification requests in cases of ambiguity or failure.
- *HostAgent State:* The current state within the HostAgent's internal FSM.

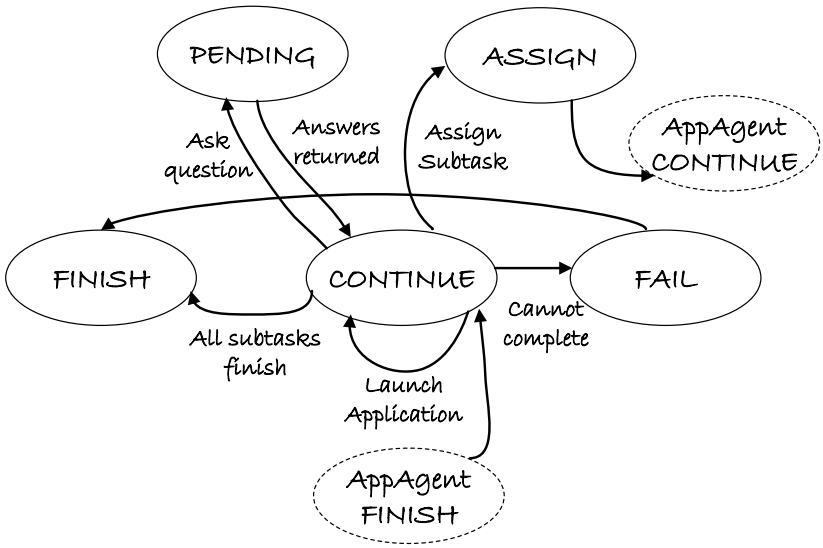

Figure 4: Control-state transitions managed by HostAgent.

Execution via Finite-State Controller. The core logic of HostAgent is modeled as a finite-state controller (Figure 4), with the following states:

- `CONTINUE:` Main execution loop; evaluates which subtasks are ready to launch or resume.
- `ASSIGN:` Selects an available application process and spawns the corresponding AppAgent agent.
- `PENDING:` Waits for user input to resolve ambiguity or gather additional task parameters.
- `FINISH:` All subtasks complete; cleans up agent instances and finalizes session state.
- `FAIL:` Enters recovery or abort mode upon irrecoverable failure.

This explicit FSM structure enables HostAgent to robustly orchestrate dynamic workflows while maintaining high-level guarantees over task completion and fault isolation.

Memory and State Management. HostAgent maintains two classes of persistent state:

- **Private State:** Tracks user intent, plan progress, and the control flow of the current session.
- **Shared Blackboard:** A concurrent, append-only memory space that facilitates transparent agent communication by recording key observations, intermediate results, and execution metadata accessible to all AppAgent instances.

This separation ensures that local context remains encapsulated, while global coordination is visible and consistent across the system.This separation ensures that each agent retains clean, scoped state while benefiting from a globally consistent view for collaborative task execution.

Overall, HostAgent abstracts away the complexity of managing concurrent, stateful, cross-application workflows in desktop environments Zhang et al. (2024e). Its control-plane role enables modular execution, coordinated progress, and robust task lifecycle management—all critical features in scaling desktop automation to real-world deployments.

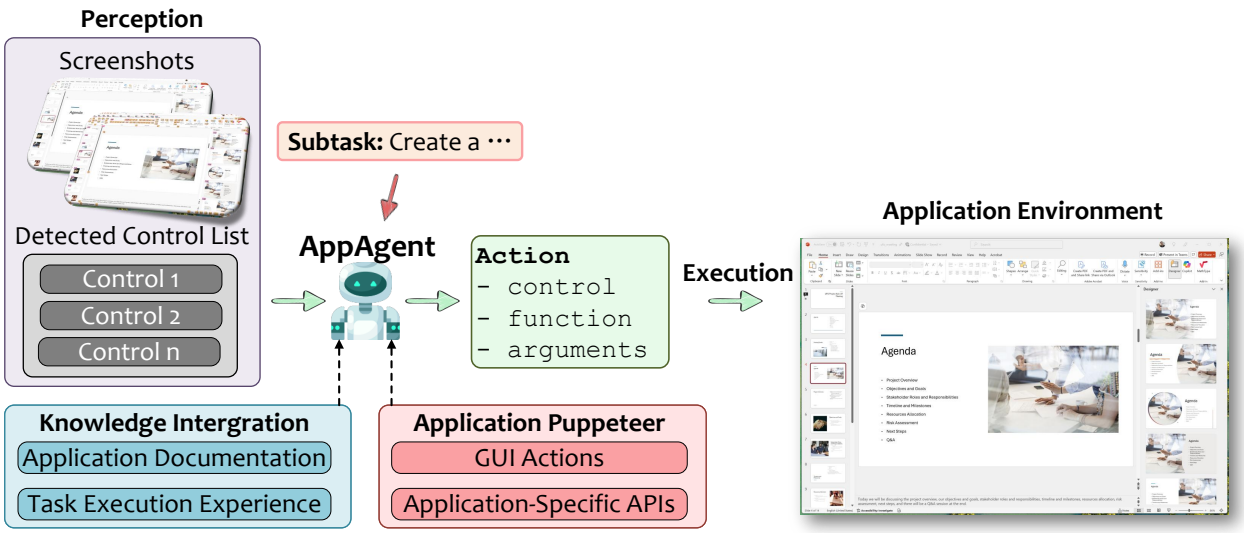

Figure 5: Architecture of an AppAgent, the per-application execution runtime in UFO^2.

3.3 AppAgent: Application-Specialized Execution Runtime

The AppAgent is the core execution runtime in UFO^2, responsible for carrying out individual subtasks within a specific Windows application. Each AppAgent functions as an isolated, application-specialized worker process launched and orchestrated by the central HostAgent (Section 3.2). Unlike monolithic CUAs that treat all GUI contexts uniformly, each AppAgent is tailored to a single application and operates with deep knowledge of its API surface, control semantics, and domain logic.

Figure 5 outlines the architecture of an AppAgent. Upon receiving a subtask and execution context from the HostAgent, the AppAgent initializes a ReAct-style control loop Yao et al. (2023), where it iteratively senses the current application state, reasons about the next step, and executes either a GUI or API-based action. This hybrid execution layer—implemented via a `Puppeteer` interface—enables reliable control over dynamic and complex UIs by favoring structured APIs whenever available, while retaining fallback to GUI-based interaction when necessary.

Perception Layer. Each AppAgent fuses multiple streams of perception:

- **Visual Input:** Captures GUI screenshots for layout understanding and control grounding.
- **Semantic Metadata:** Extracted from Windows UIA APIs, including control type, label, hierarchy, and enabled state.
- **Symbolic Annotation:** Uses Set-of-Mark (SoM) techniques Yang et al. (2023a) to annotate the control on screenshots.

These fused signals are converted into a structured observation object containing both the GUI screenshot and the set of candidate control elements. This multi-modal representation enables a more comprehensive understanding of the application state, going well beyond raw visual input alone.

Structured Output. Based on this state, the AppAgent produces a structured output:

- Target control (if applicable)
- Action type (*e.g.*, click, type, call API)
- Arguments or payload

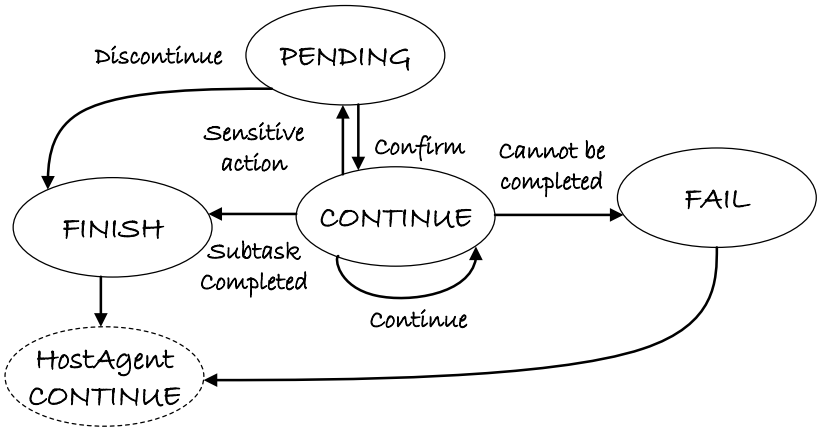

Figure 6: Control-state transitions for an AppAgent runtime.

- Reasoning trace and planning with Chain-of-Thought (CoT) Wei et al. (2022); Ding et al. (2024)
- Current state in local FSM

This design decouples perception from actuation, enabling deterministic replay, offline debugging, and fine-grained observability.

Execution via Finite-State Controller. Each AppAgent maintains a local finite-state machine (Figure 6) that governs its behavior within the assigned application context:

- `CONTINUE:` Default state for action planning and execution.
- `PENDING:` Invoked for safety-critical actions (*e.g.*, destructive operations); requires user confirmation.
- `FINISH:` Task completed; execution ends.
- `FAIL:` Irrecoverable failure detected (*e.g.*, application crash, permission error).

This bounded execution model isolates failures to the current task and enables safe preemption, retry, or delegation. The FSM also supports interruptible workflows, which can resume from intermediate checkpoints.

Memory and State Coordination. To enable stateful execution and maintain contextual awareness, each AppAgent maintains:

- **Private State:** A local log of all executed actions, control decisions, and CoT traces.
- **Shared State:** Updates to the system-wide blackboard, including intermediate outputs, encountered errors, and application-level insights.

This dual-memory design enables AppAgents to act autonomously on behalf of HostAgent while remaining synchronized with the broader system. It also supports composability: one AppAgent's output may become another's input in downstream subtasks.

Application-Aware SDK and Extensibility. To support rapid onboarding of new applications, UFO^2 exposes an SDK that encapsulates the development and maintenance of AppAgents. Developers can register application-specific APIs via a declarative interface, including function metadata, argument schemas, and prompt bindings. Domain-specific help documents and patch notes can be ingested into a searchable knowledge base that agents query at runtime.

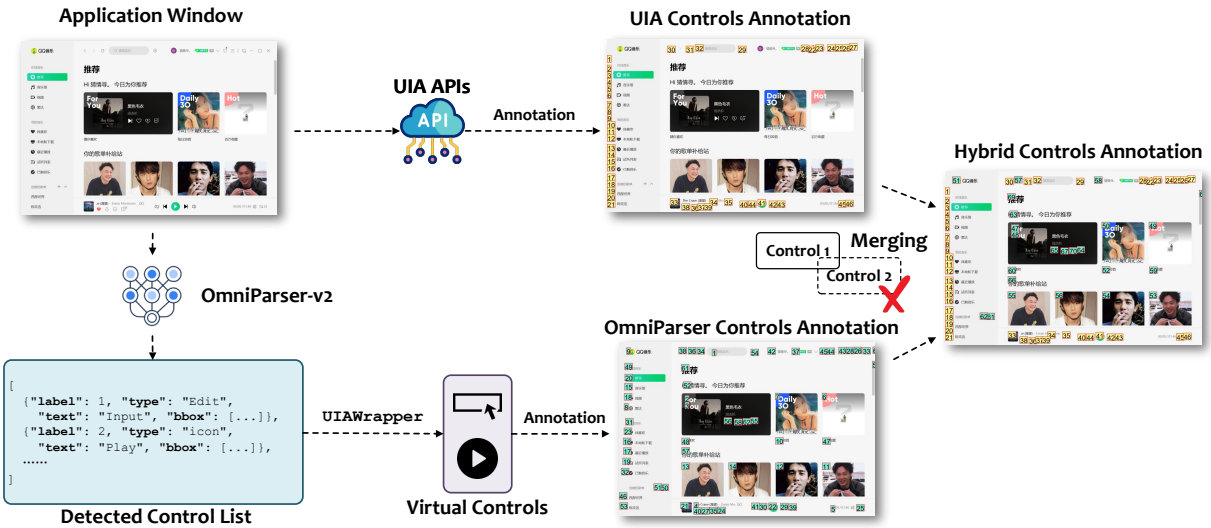

Figure 7: The hybrid control detection approach employed in UFO^2.

This modular abstraction allows third-party vendors or power users to extend UFO^2's capabilities without retraining any models. New functionality can be integrated by updating the application's AppAgent module, isolating changes from the rest of the system and minimizing regression risk.

Summary. As a per-application execution runtime, each AppAgent provides modular, domain-aware control that surpasses generic GUI agents in both efficiency and robustness. Its hybrid perception-action loop, plugin-based extensibility, and local fault containment enable UFO^2 to scale to large application ecosystems with minimal system-wide disruption.

3.4 Hybrid Control Detection

Reliable perception of GUI elements is fundamental to enabling AppAgents to interact with application interfaces in a deterministic and safe manner. However, real-world GUI environments exhibit substantial heterogeneity: some applications expose well-structured accessibility data via Windows UI Automation (UIA) APIs, while others—especially legacy or custom applications—render critical controls using non-standard toolkits that bypass UIA entirely.

To address this disparity, UFO^2 introduces a *hybrid control detection* subsystem that fuses UIA-based metadata with vision-based grounding Lu et al. (2024); Cheng et al. (2024); Gou et al. (2024) to construct a unified and comprehensive control graph for each application window (Figure 7). This design ensures both coverage and reliability, forming a resilient perceptual foundation for downstream action planning and execution.

UIA-Layer Detection. When available, UIA offers a semantically rich and high-precision interface for enumerating on-screen controls. The detection pipeline first queries the accessibility tree to extract controls satisfying a set of runtime predicates (*e.g.*, `is_visible()`, `is_enabled()`). These controls are annotated with their attributes (type, label, bounding box) and assigned stable identifiers, forming the initial control graph.

Vision-Layer Augmentation. To augment the perception pipeline for UI-invisible or custom-rendered controls, we integrate OmniParser-v2 Lu et al. (2024), a vision-based grounding model designed for fast and accurate GUI parsing. OmniParser-v2 combines a lightweight YOLO-v8 Reis et al. (2023) detector with a fine-tuned Florence-2 (0.23B) Xiao et al. (2024) encoder to process raw application screenshots and

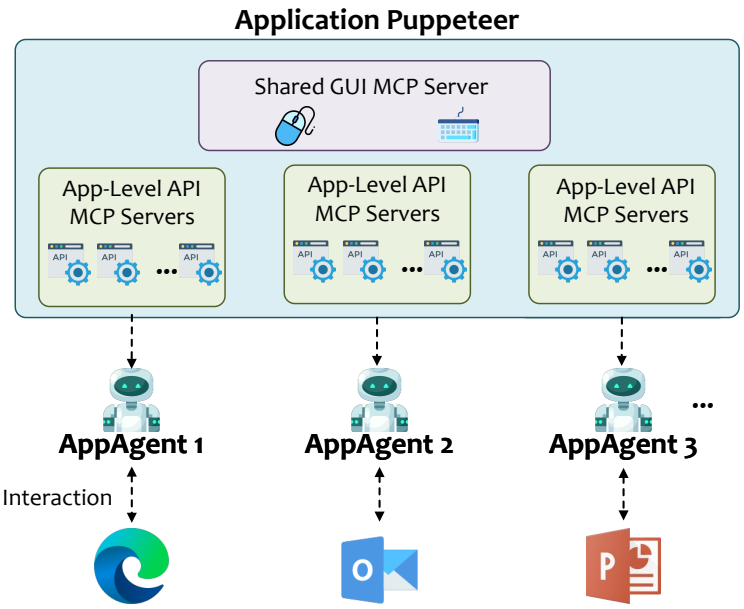

Figure 8: `Puppeteer` serves as a unified execution engine that harmonizes GUI actions and native API calls with MCP servers.

identify additional interactive elements. Each detection includes the control type, confidence score, and spatial bounding box.

Fusion and Deduplication. We unify these two streams by performing deduplication based on bounding-box overlap. Visual detections with Intersection-over-Union (IoU) greater than 10% against any UIA-derived control are discarded. Remaining visual-only detections are converted into pseudo-UIA objects using a lightweight `UIAWrapper` abstraction, allowing them to seamlessly integrate into the rest of the AppAgent pipeline. This fused control set is passed downstream to the SoM-based annotation module Yang et al. (2023a). Figure 7 shows a typical scenario involving a hybrid-rendered GUI. Yellow bounding boxes denote standard UIA-detected elements, while blue bounding boxes represent visual-only detections. Both are integrated into a single actionable control graph consumed by the AppAgent.

3.5 Unified GUI–API Action Orchestrator

AppAgents interact with application environments that typically expose two complementary classes of interfaces: *(i) GUI frontends*, which are universally available but inherently brittle; and *(oi) native APIs*, which offer high-fidelity control but require explicit integration and maintenance Zhang et al. (2025).

To unify these heterogeneous execution backends under a single abstraction, UFO^2 introduces the `Puppeteer`, a modular action orchestrator that manages a collection of *Model Context Protocol* (MCP) servers exposing both GUI and API operations. We choose MCP servers for GUI and API integration due to their ease of use, ability to establish a standardized action interface, and strong ecosystem support.

Concretely, `Puppeteer` maintains a shared *GUI MCP server* that simulates generic user inputs such as mouse clicks and keyboard events. This server is accessible to all applications, ensuring coverage for tasks where no native APIs are available. In addition, each application can define its own *API MCP servers* Hou et al. (2025), exposing application-specific capabilities (*e.g.*, Excel cell operations, Outlook email handling). Multiple servers can be registered for a single application, allowing fine-grained decomposition of functionality.

At runtime, `Puppeteer` dynamically selects between GUI-level automation and API-level execution for each action step (Figure 8). This design improves *robustness* (by falling back to GUI actions when APIs are missing), *latency* (by collapsing long GUI sequences into single API calls), and *maintainability* (by modular-

```python
@mcp.tool(tags={"ExcelAgent"})
def summarize_table(
    sheet_name: Annotated[str, Field(description="Name or index of the target worksheet")],
    table_range: Annotated[str, Field(description="Cell range in Excel A1 notation (e.g., 'A1:D10')")]
) -> Annotated[Dict, Field(description="Summary statistics of the selected table")]:
    """
    Extracts the content of a table from Excel and returns basic
    statistics (row count, column count, column headers) along with
    a Markdown preview of the data.
    """
    ...
```

Figure 9: An example API registration for Excel. Developers expose high-level operations (here, table summarization) by attaching an `@mcp.tool` decorator. The tool is automatically integrated into the AppAgent's runtime action space for Excel.

izing app-specific APIs behind a stable interface). For example, formatting a block of cells in Excel can be executed as one API call instead of a series of GUI interactions, substantially reducing both execution time and the surface area for error Song et al. (2024); Zhang et al. (2025).

To support extensibility, `Puppeteer` provides a lightweight API registration mechanism. Developers can expose high-level operations by attaching simple Python decorators to functions (Figure 9). Each registered function is automatically wrapped with metadata, including its name, argument schema, and associated AppAgent binding and incorporated into the AppAgent's runtime action space. This declarative mechanism lowers the barrier for integrating new capabilities, enabling continuous evolution of application-specific automation while preserving architectural modularity and reuse.

At runtime, UFO^2 prompts AppAgent to employs a decision-making policy to choose the most appropriate execution path for each operation. If a semantically equivalent API is available, `Puppeteer` prefers it over GUI automation for reliability and atomicity.If an API fails or is unavailable (*e.g.*, missing bindings or runtime permission errors), the system gracefully falls back to GUI-based control via simulated clicks or keystrokes. This runtime flexibility allows AppAgent to preserve robustness across heterogeneous environments without sacrificing generality.

`Puppeteer` transforms action execution in UFO^2 from a monolithic GUI-centric model into a flexible, OS-integrated control layer that mixes perceptual agility with semantic precision. This hybrid execution model not only improves system performance and stability but also lays the groundwork for sustainable integration of application-specific capabilities in future desktop agents.

3.6 Continuous Knowledge Integration Substrate

Unlike traditional CUAs, which rely heavily on static training corpora, UFO^2 introduces a persistent and extensible *knowledge substrate* that supports runtime augmentation of application-specific understanding. As illustrated in Figure 10, this substrate enables each AppAgent to retrieve, interpret, and apply external documentation and prior execution traces without requiring retraining of the underlying models. This hybrid memory design functions analogously to an OS-level metadata manager, abstracting over two key knowledge flows: static references (*e.g.*, user manuals) and dynamic experience (*e.g.*, execution logs).

Bootstrapping from Documentation. Most real-world desktop applications expose substantial task-level documentation via user guides, help menus, or online tutorials. UFO^2 capitalizes on this resource by offering a one-click interface to parse and ingest such documents into an application-specific vector store. Documents are structured as `json` records with a natural language description in the `request` field and detailed execution guidance in the `guidance` field.

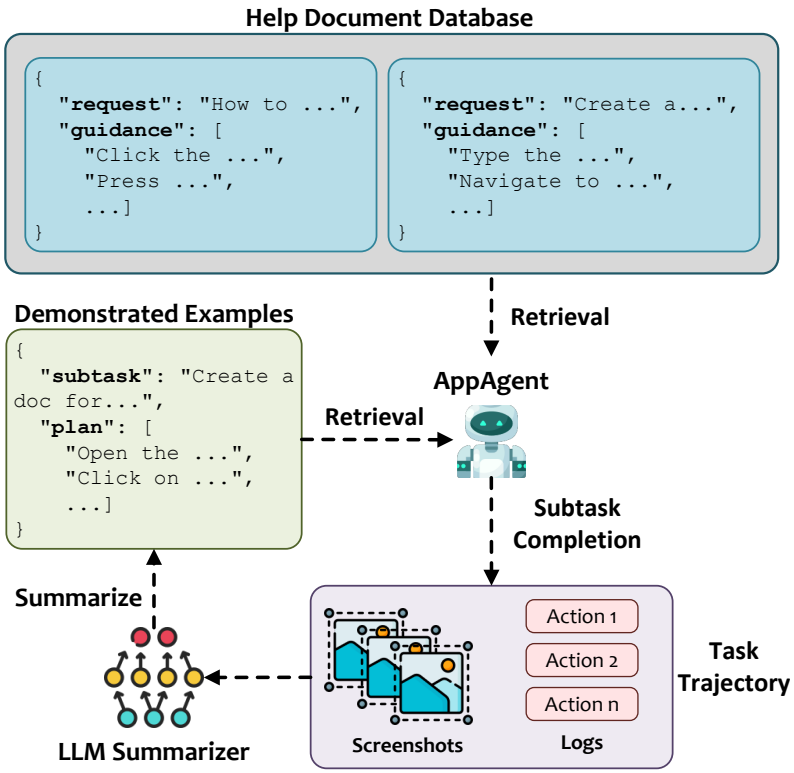

Figure 10: Overview of the knowledge substrate in UFO^2, combining static documentation with dynamic execution history.

At runtime, when an AppAgent receives a subtask, it queries this indexed store to retrieve relevant guidance, which is then injected into the agent's prompt. This mechanism effectively mitigates cold-start issues—especially when handling novel applications or infrequent operations—by enriching the agent's reasoning context with domain-grounded procedural knowledge.

Reinforcing from Experience. Beyond static knowledge, UFO^2 continuously learns from its own execution history. Each automation run produces structured logs—including natural language task descriptions, executed action sequences, application screenshots, and final outcomes. Periodically, these logs are mined offline by a summarization module that distills successful trajectories into reusable `Example` records.

Each record contains a task signature and an associated step-by-step plan, stored in an application-specific example database. When a similar task is encountered in the future, AppAgent uses In-Context Learning (ICL) Dong et al. (2024); Min et al. (2022); Zhang et al. (2024c); Luo et al. to retrieve relevant demonstrations and improve execution fidelity. This dynamic reinforcement pipeline transforms the system into a long-lived agent that improves with use, without introducing the brittleness or operational cost of fine-tuning Wang et al. (2025).

Runtime RAG Integration. At the system level, the knowledge substrate acts as a Retrieval-Augmented Generation (RAG) layer Lewis et al. (2020); Liu et al. (2024); Gao et al. (2023); Jiang et al. (2024) that bridges the gap between pre-trained language models and application-specific requirements. Because both help documents and examples are indexed with semantic embeddings, the retrieval pipeline is fast, interpretable, and cache-friendly. Additionally, versioned indexing ensures that knowledge artifacts can evolve alongside software updates, preventing model obsolescence and supporting robust execution across long deployment cycles.

Algorithm 1 Speculative Multi-Action Execution in UFO^2

Require: Initial UI context C_0, batch size k
Ensure: List `Executed` of actions completed so far

Stage 1: Batch Prediction

$A \leftarrow$ LLM_PREDICT(C_0, k) ▷ $A = \left[(ctrl_i, op_i\right]_{i=1}^{k}$
`Executed` $\leftarrow$ [ ]; $C \leftarrow C_0$

Stage 2 & 3: Sequential Validate-Execute Loop

for $i \leftarrow 1$ **to** k **do**
  $(ctrl, op, _) \leftarrow A[i]$
  // Validate in the current *context*
  if not UIA_ISENABLED$(ctrl, C)$ $\vee$ **not** UIA_ISVISIBLE$(ctrl, C)$ **then**
    break ▷ validation failed $\rightarrow$ early stop
  end if
  //Execute and refresh context
  EXECUTE$(ctrl, op)$
  `append` $(ctrl, op)$ to `Executed`
  $C \leftarrow$ UIA_GETCONTEXT() ▷ UI changed
end for
if $|$`Executed`$| < k$ **then**
  REPORTPARTIAL(`Executed`)
  REPLAN(C)
end if

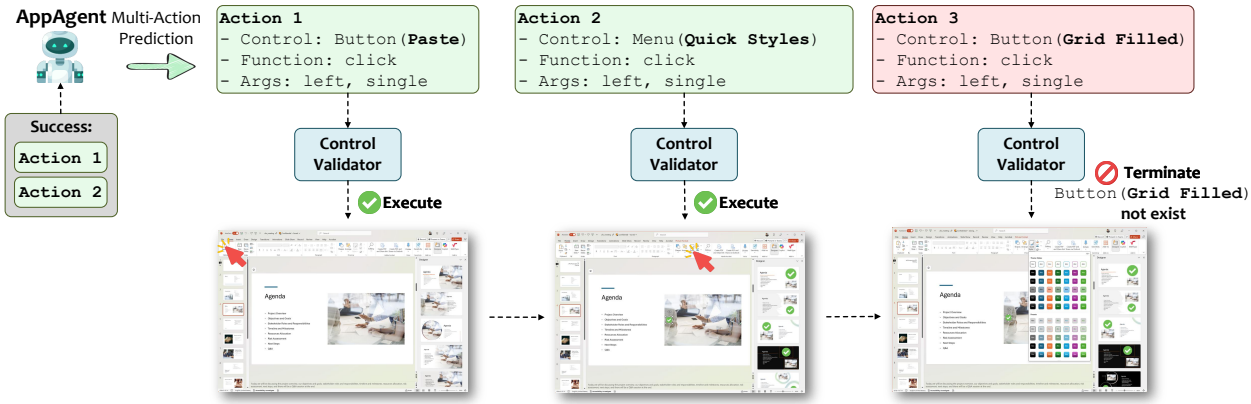

Figure 11: Speculative multi-action execution in UFO^2: batched inference with online validation.

By integrating static and experiential knowledge into a unified RAG pipeline, UFO^2 transforms CUAs from brittle, training-time constructs into dynamic, evolving agents. This substrate plays a foundational role in enabling sustainable automation across complex, heterogeneous application ecosystems.

3.7 Speculative Multi-Action Execution

Conventional CUAs suffer from a fundamental execution bottleneck: each automation step is executed in isolation, requiring a full LLM inference for every single GUI action. This *step-wise inference loop* introduces excessive latency, inflates system resource usage, and increases cumulative error rates—especially when interacting with complex or multi-phase workflows Zhang et al. (2025). The root cause is the dynamic and uncertain nature of GUI environments, where any single action may alter the interface and invalidate future plans.

To overcome these limitations, UFO^2 introduces a system-level optimization called *speculative multi-action execution*, inspired by classical ideas from speculative execution in processor design and instruction pipelining.

Rather than issuing one action per LLM call, UFO^2 speculatively generates a *batch* of likely next steps using a single inference pass and validates their applicability at runtime through tight OS integration. We present an algorithm in 1.

The speculative executor operates in three stages:

1. **Action Prediction:** The AppAgent issues a single LLM query to predict multiple plausible actions under its current context. Each predicted step includes a target control, intended operation, and rationale.

2. **Runtime Validation:** For each action, the system consults the Windows UIA API to verify the action's preconditions (*e.g.*, `is_enabled()`, `is_visible()`). This check ensures that each target control is still valid and interactive.

3. **Sequential Execution and Early Exit:** Actions are executed in order, halting immediately if any validation fails due to interface change (*e.g.*, control no longer exists or is disabled). The executor then reports a partial result set and prompts the agent to replan.

We show an illustrative example of speculative multi-action execution in Figure 11. In this case, the AppAgent initially plans to execute three actions in a single step: clicking `Paste`, then `Quick Style`, and finally `Grid Filled`. However, after the second action, the control validator detects that the control required for the third action (`Grid Filled`) is no longer present—likely because the GUI layout changed as a result of the previous step. The `Puppeteer` then terminates execution at that point and returns the partial results. This example highlights how UFO^2 safely handles speculative execution by validating each control before acting, ensuring robustness even in the face of dynamic interface changes.

Overall, this strategy drastically reduces LLM invocation frequency and amortizes the cost of action planning across multiple steps, while preserving the correctness guarantees of per-step validation. Critically, validation is performed by trusted OS-level APIs instead of vision models, ensuring high reliability and eliminating spurious interactions.

4 Picture-in-Picture Interface

A key design objective of UFO^2 is to deliver high-throughput automation while preserving the responsiveness and usability of the primary desktop environment. Existing CUAs often monopolize the user's workspace, seizing mouse and keyboard control for extended periods and making the system effectively unusable during task execution. To overcome this, UFO^2 introduces a *Picture-in-Picture* (PiP) interface: a lightweight, virtualized desktop window powered by Remote Desktop loopback, enabling fully isolated agent execution in parallel with active user workflows, as illustrated in Figure 12.

4.1 Virtualized User Environment with Minimal Disruption

Unlike conventional CUAs that operate in the main desktop session, the PiP interface presents a resizable, movable window containing a fully functional replica of the user's desktop. Internally, this is implemented via Windows' native Remote Desktop Protocol (RDP) loopback Miller & Pegah (2007), creating a distinct virtual session hosted on the same machine. Applications launched within the PiP session inherit the user's identity, credentials, settings, and network context, ensuring consistency with foreground operations.

From the user's perspective, the PiP window behaves like a sandboxed workspace: the automation executes in the background, visible but unobtrusive. The user retains full control of the primary desktop and can minimize or reposition the PiP window at will. This enables UFO^2 to perform long-running or repetitive workflows (*e.g.*, data entry, batch file processing) without blocking user interaction or degrading responsiveness.

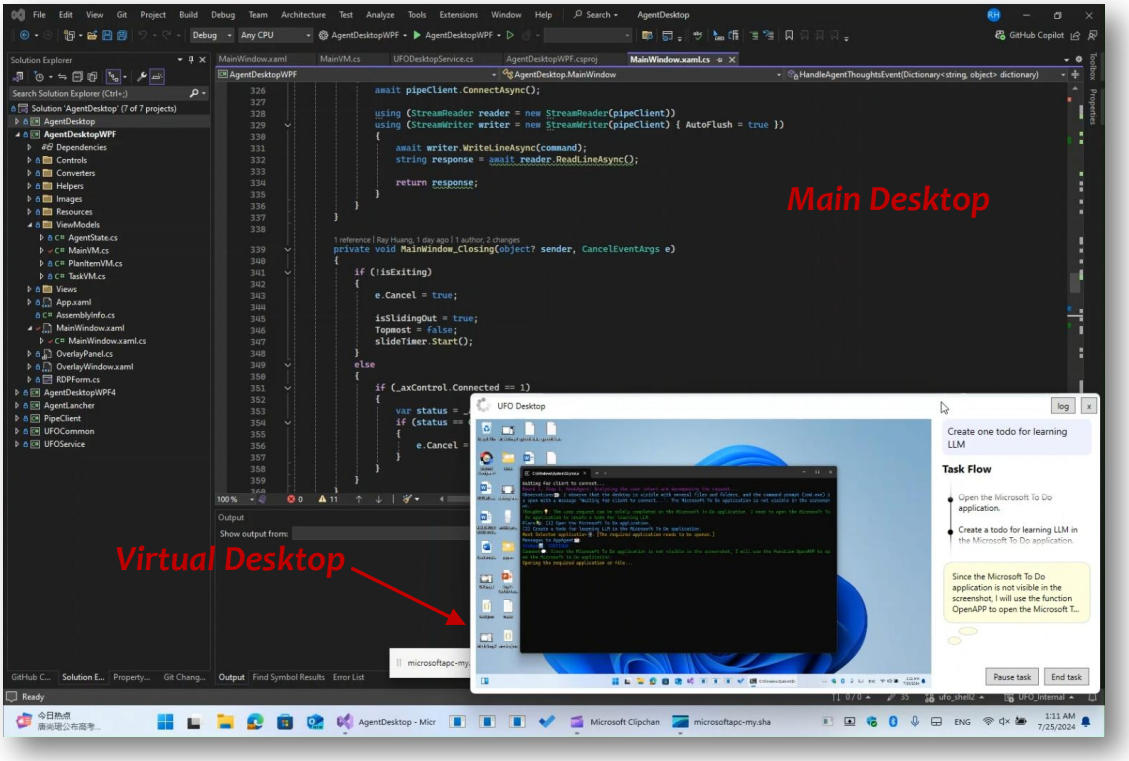

Figure 12: The Picture-in-Picture interface: a virtual desktop window for non-disruptive automation.

4.2 Robust Input and State Isolation

To ensure robust separation between agent actions and user activities, UFO^2 leverages the RDP subsystem to maintain distinct input queues and device contexts across sessions. Mouse and keyboard events generated within the PiP desktop are fully scoped to that session and cannot interfere with the primary desktop. Similarly, GUI changes and focus transitions are restricted to the virtual environment.

This level of input isolation is critical for preventing accidental interference—either by the user or the agent—and ensures that automation sequences remain stable, even during simultaneous foreground activity. The architecture also supports controlled error recovery: failures or unexpected UI states within the PiP session do not propagate to the primary desktop, preserving the integrity of the user's environment.

4.3 Secure Cross-Session Coordination

Although visually and operationally distinct, the PiP session must remain logically connected to the host environment. To enable this, UFO^2 establishes a secure inter-process communication (IPC) channel between the PiP agent runtime and a host-side coordinator. We implement this using Windows Named Pipes, authenticated and encrypted using per-session credentials Venkataraman & Jagadeesha (2015).

This IPC layer supports two-way messaging:

- From the host to the PiP: task assignment, progress polling, cancellation, and user clarifications.
- From the PiP to the host: status updates, completion reports, and exception notifications.

Users interact with the automation pipeline through a lightweight frontend panel on the host desktop, enabling real-time visibility and partial control without needing to directly access the PiP window. This

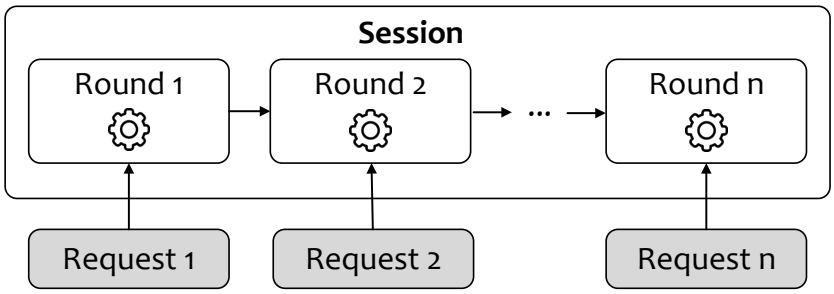

Figure 13: The interactive `Session` model in UFO^2 supports multi-round refinement.

transparent yet secure communication channel ensures trust and usability, particularly in long-running or partially supervised workflows.

4.4 System-Level Implications

The PiP interface represents more than a UX refinement—it is a system-level abstraction that reconciles concurrency, usability, and safety. It decouples automation execution from foreground interactivity, introduces a new isolation primitive for GUI-based agents, and simplifies failure recovery by sandboxing side effects. By exploiting existing RDP capabilities with minimal system overhead, the PiP interface offers a practical and backwards-compatible approach to scalable desktop automation.

5 Implementation and Specialized Engineering Design

We implement UFO^2 as a full-stack desktop automation framework spanning over 30,000 lines of `Python` and `C#` code. Python serves as the core runtime environment for agent orchestration, control logic, and API integration, while `C#` supports GUI development, debugging interfaces, and Windows-specific operations such as the Picture-in-Picture desktop. To support retrieval-augmented reasoning, UFO^2 leverages Sentence Transformers Reimers & Gurevych (2019) for embedding-based document and experience retrieval.

Beyond its core functionality, UFO^2 incorporates multiple specialized engineering components that target critical systems goals: composability, interactivity, debuggability, and scalable deployment. We highlight several key mechanisms below.

5.1 Multi-Round Task Execution

Unlike stateless one-shot agents, UFO^2 adopts a session-based execution model to support iterative, interactive workflows (Figure 13). Each `Session` maintains persistent contextual memory—including intermediate results, task progress, and application state—across multiple `Rounds` of execution. Users can refine prior instructions, launch follow-up tasks, or intervene when agents encounter ambiguous or unsafe operations.

This multi-round interaction paradigm facilitates progressive convergence on complex tasks while preserving transparency and human oversight. It enables UFO^2 to support human-in-the-loop refinement strategies, bridging static LLM workflows with dynamic user guidance.

5.2 Safeguard Mechanism

While automation substantially boosts productivity, any CUA carries inherent risks of executing unsafe actions that may adversely affect user data or system stability Levy et al. (2024); Zhang et al. (2024b). Examples include deleting critical files, terminating applications prematurely (resulting in unsaved data loss), or activating sensitive devices such as webcams without explicit consent. These actions pose severe risks, potentially causing irrecoverable damage or security breaches.

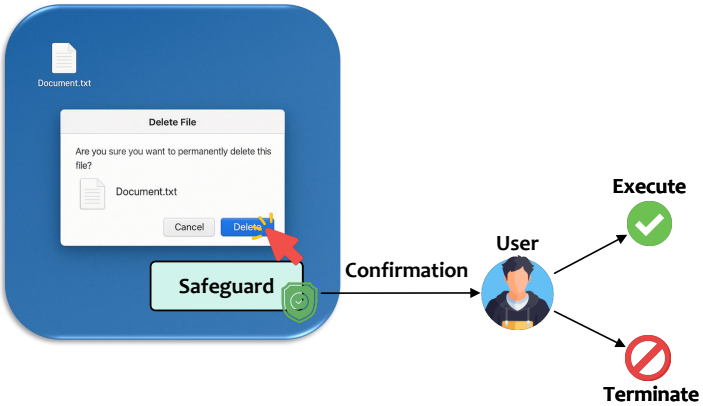

Figure 14: The safeguard mechanism employed in UFO^2.

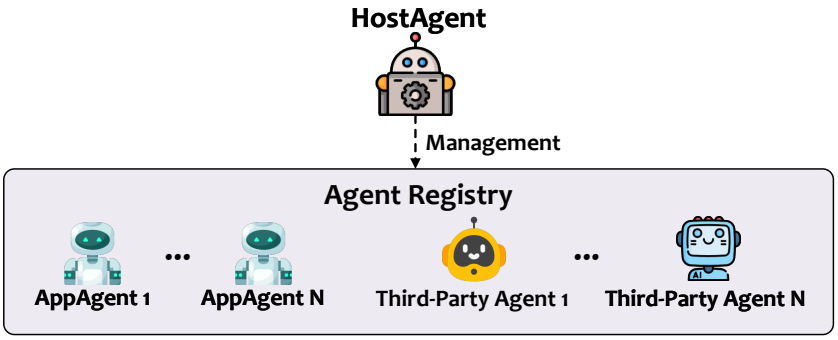

Figure 15: The agent registry supports seamless wrapping of third-party components into the AppAgent framework.

To mitigate such risks, UFO^2 incorporates an explicit *safeguard mechanism*, designed to actively detect potentially dangerous actions, as shown in Figure 14. Specifically, whenever an AppAgent identifies an action matching predefined risk criteria, it transitions into a dedicated `PENDING` state, pausing execution and actively prompting the user for confirmation. Only upon receiving explicit user consent does the agent proceed; otherwise, the action is aborted to prevent harm. The definition and scope of what constitutes a risky action are fully customizable through a straightforward prompt-based interface, enabling users and system administrators to precisely tailor safeguard behavior according to their organization's specific risk policies. This flexibility allows the safeguard system to be dynamically adapted as automation requirements evolve.

Through this proactive safety-checking framework, UFO^2 significantly reduces the likelihood of executing harmful operations, thus enhancing overall system safety, user trust, and robustness in real-world deployments.

5.3 Everything-as-an-AppAgent

To support ecosystem extensibility, UFO^2 introduces an agent registry mechanism that encapsulates arbitrary third-party components as pluggable AppAgents (Figure 15). Through a simple registration API, external automation solutions—such as domain-specific copilots or proprietary tools—can be wrapped with lightweight compatibility shims that expose a unified interface to the HostAgent.

This design enables HostAgent to treat native and external AppAgents interchangeably, dispatching subtasks based on capability and specialization. We find that even minimal wrappers (*e.g.*, for OpenAI

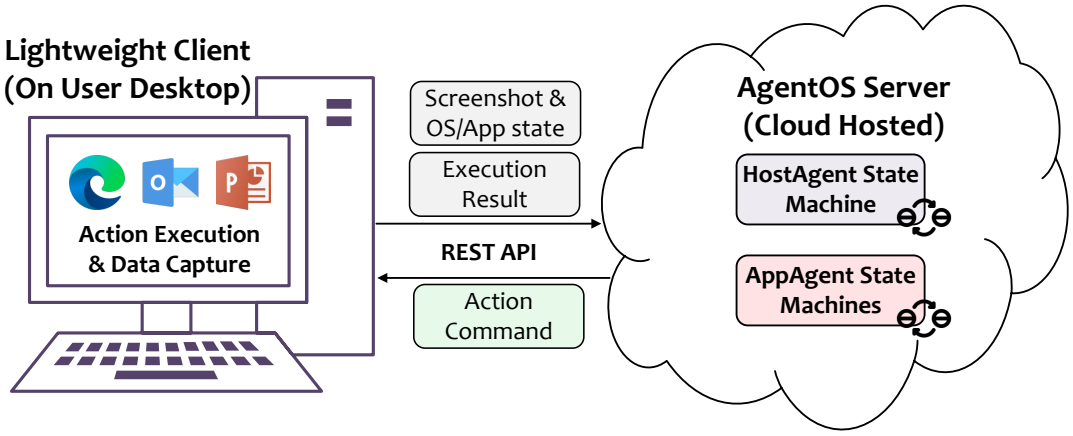

Figure 16: The client-server deployment model used in AgentOS-as-a-Service.

Operator OpenAI (2025)) lead to tangible performance gains, highlighting the system's modularity and its ability to incorporate diverse execution backends with minimal engineering overhead.

5.4 AgentOS-as-a-Service

UFO^2 adopts a client-server architecture to support practical deployment at scale (Figure 16). A lightweight client resides on the user's machine and is responsible for GUI operations and application-side sensing. Meanwhile, a centralized server (running on-premises or in the cloud) hosts the HostAgent/AppAgent logic, orchestrates workflows, and handles LLM queries.

This separation of control and execution offers several systems-level benefits:

- **Security**: Sensitive orchestration and model execution are isolated from user devices.
- **Maintainability**: Server-side updates propagate without modifying the client.
- **Scalability**: The system can support multiple concurrent clients with centralized scheduling and load management.

The client-server boundary enforces a clean service abstraction, promoting modularity and simplifying rollout in enterprise environments.

5.5 Comprehensive Logging and Debugging Infrastructure

Robust observability is essential for diagnosing failures and supporting ongoing system improvement. To this end, UFO^2 implements a comprehensive logging and debugging framework. Each session captures fine-grained traces of execution: prompts, LLM outputs, control metadata, UI state snapshots, and error events.

At the end of each session, UFO^2 compiles these artifacts into a structured, Markdown-formatted execution log. Developers can inspect action-by-action agent decisions, visualize interface state transitions, and replay behavior for debugging. The framework also supports prompt editing and selective replay for targeted hypothesis testing, significantly accelerating the debugging cycle. We show an example of these tools in Figure 17.

This observability layer functions as a lightweight provenance system for agent behavior, fostering transparency, accountability, and rapid iteration during deployment.

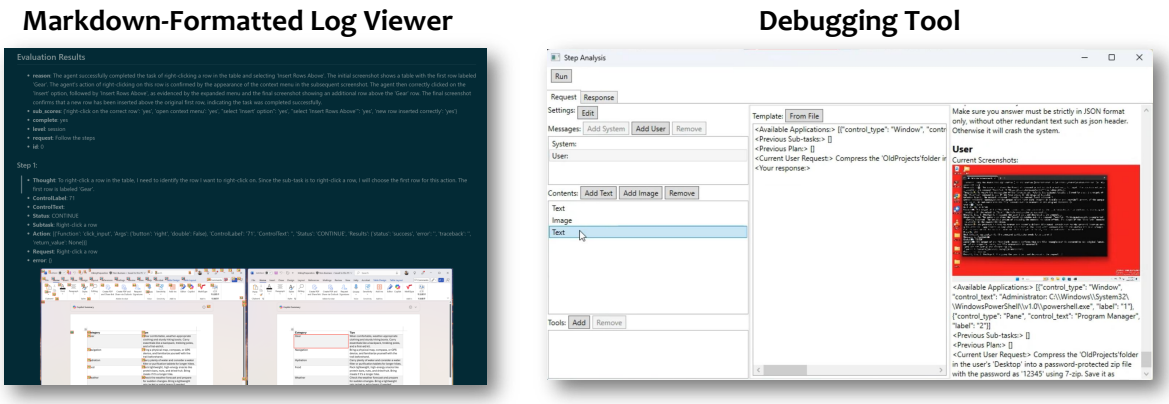

Figure 17: An illustration of the markdown-formatted log viewer and debugging tool in UFO^2.

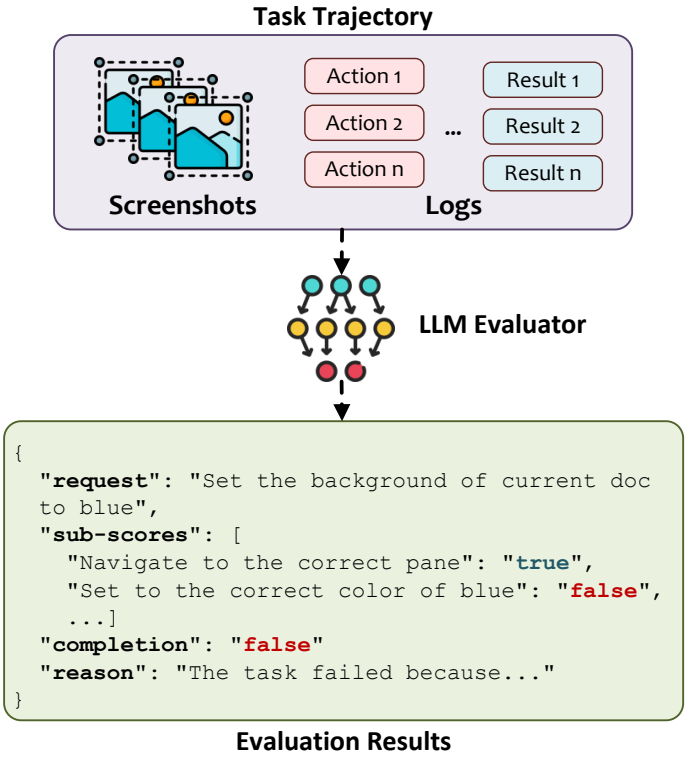

Figure 18: The LLM-based task evaluator applies CoT reasoning to structured session logs.

5.6 Automated Task Evaluator

To provide structured feedback and facilitate continuous improvement, UFO^2 includes an automated task evaluation engine based on LLM-as-a-judge Chen et al. (2024a). As shown in Figure 18, the evaluator parses session traces—including actions, rationales, and screenshots—and applies CoT reasoning to decompose tasks into evaluation criteria.

It assigns partial scores and synthesizes an overall result: *success*, *partial*, or *failure*. This structured outcome feeds into downstream dashboards and debugging tools. It also supports self-monitoring and offline analysis of failure cases, closing the loop between execution, diagnosis, and improvement.

Summary. These engineering components demonstrate UFO^2's commitment to operational robustness and extensibility. From session-based execution and pluggable agents to service-oriented deployment and observability infrastructure, each module reflects a design focused on bridging conceptual LLM agent architectures with the systems realities of deployment at scale.

6 Evaluation

We tested UFO^2 rigorously across more than 20 Windows applications, including office suites, file explorers, and custom enterprise tools to assess performance, efficiency, and robustness. Our experiments show:

1. UFO^2 achieves a *10%* higher task completion rate, a 50% relative improvement—over the best-performing current CUA Operator, enabled by deeper OS-level integration.
2. The hybrid UIA–vision approach identifies custom or nonstandard GUI elements missed by UIA alone, boosting success in interfaces with proprietary widgets.
3. Allowing AppAgents to invoke native APIs or GUI interactions to improve completion rate by over 8%, cuts latency and reduces the fragility seen in purely click-based workflows.
4. Leveraging external documents and execution logs increases UFO^2's ability to handle unfamiliar features without retraining.
5. Speculative multi-action execution consolidates multiple steps into a single LLM call, lowering inference cost by up to *51.5%* without compromising reliability.
6. By enabling Everything-as-an-AppAgent (*e.g.*, Operator), UFO^2 both boosts overall performance and uncovers the full potential of each individual agent.

Overall, these results confirm that UFO^2's deeper integration with Windows and application-level APIs yields both higher performance and reduced overhead, making a compelling case for an OS-native approach to desktop automation.

6.1 Experimental Setup

Deployment Environment. The benchmark environments are hosted on isolated VMs with *8* AMD Ryzen 7 CPU cores and *8* GB of memory, matching typical deployment conditions. All GPT-family models (GPT-4V, GPT-4o, o1, and Operator) are accessed via Azure OpenAI services, while the OmniParser-v2 vision model operates on a separate virtual machine provisioned with an NVIDIA A100 80GB GPU to support efficient and high-throughput visual grounding.

Benchmarks. We evaluate UFO^2 using two established Windows-centric automation benchmarks:

- **Windows Agent Arena (WAA) Bonatti et al.:** Consists of 154 live automation tasks across 15 commonly used Windows applications, including office productivity tools, web browsers, system utilities, development environments, and multimedia apps. Each task includes a custom verification script for automated correctness checking.
- **OSWorld-W Xie et al. (2024):** A targeted subset of the OSWorld benchmark specifically tailored for Windows, comprising 49 live tasks across office applications, browser interactions, and file-system operations. Tasks are similarly equipped with handcrafted verification scripts for reliable outcome validation.

Each task runs independently, and verification strictly follows the original scripts provided by each benchmark.[1]

[1]Reported baseline scores in OSWorld differ slightly from prior results focused on Ubuntu due to corrections in verification scripts and alignment with Windows-specific tasks (OSWorld-W).

Baselines. We compare UFO^2 with five representative state-of-the-art CUAs, each leveraging GPT-4o as the inference engine:

- **UFO** Zhang et al. (2024b): A pioneering multiagent, GUI-focused automation system designed explicitly for Windows, integrating UIA and visual perception.
- **NAVI** Bonatti et al.: A single-agent baseline from WAA, utilizing screenshots and accessibility data for GUI understanding.
- **OmniAgent** Lu et al. (2024): Employs OmniParser for visual grounding combined with GPT-based action planning.
- **Agent S** Agashe et al.: Features a multiagent architecture with experience-driven hierarchical planning, optimized for complex, multi-step tasks.
- **Operator** OpenAI (2025): A recent, high-performance CUA from OpenAI, simulating human-like mouse and keyboard interactions via screenshots.

These baselines were selected for their representativeness of diverse architectural and design paradigms (*e.g.*, single-agent vs. multiagent, GUI-only vs. hybrid approaches). To ensure fairness, each agent is restricted to a maximum of *30 execution steps* per task, reflecting practical user expectations and preventing excessively long task executions. Additionally, we evaluate a base version of UFO^2 (termed UFO^2-base) using only UIA detection, GUI-based interactions, and without dynamic knowledge integration, alongside the full implementation of UFO^2 featuring hybrid control detection, combined GUI-API interactions, and continuous knowledge augmentation. API integrations were selectively implemented for three office applications within OSWorld-W as illustrative examples; no APIs were introduced for the WAA tasks. Further implementation details are available in Section 6.4.

Evaluation Metrics. We utilize two primary metrics for performance evaluation:

- **Success Rate (SR):** Defined as the percentage of tasks successfully completed, validated via the benchmarks' own verification scripts.
- **Average Completion Steps (ACS):** Measures the average number of LLM-involved action inference steps required per task. Fewer steps correspond to higher efficiency, directly correlating with lower inference latency and reduced computational overhead.

These metrics effectively reflect both functional effectiveness and practical efficiency, providing clear indicators of real-world automation performance.

6.2 Success Rate Comparison

Table 1 summarizes the success rates (SR) of all evaluated agents across the WAA and OSWorld-W benchmarks, as verified by each benchmark's automated validation scripts. Notably, even the basic configuration (UFO^2-base)—which relies solely on standard UI Automation and GUI-driven actions—consistently surpasses prior state-of-the-art CUAs. Specifically, with GPT-4o, UFO^2-base achieves an SR of 23.4% on WAA, outperforming the best existing baseline, Operator (20.8%), by 2.6%. This margin widens significantly when employing the stronger o1 LLM, lifting UFO^2-base's performance to 25.3%.

Moreover, the complete version of UFO^2, incorporating hybrid GUI–API action execution, advanced visual grounding, and continuous knowledge integration, further amplifies these performance gains. With GPT-4o, UFO^2 achieves a 27.9% SR on WAA, exceeding Operator by a substantial 7.1%. The performance gap becomes even more pronounced on OSWorld-W, where UFO^2 achieves a 28.6% SR compared to Operator's 14.3%, effectively doubling its success rate. Utilizing the stronger o1 model further improves UFO^2's performance to 30.5% (WAA) and 32.7% (OSWorld-W), solidifying its leading position.

Table 1: Comparison of success rates (SR) across agents on WAA and OSWorld-W benchmarks.

Agent	**Model**	**WAA**	**OSWorld-W**
UFO	GPT-4o	19.5%	12.2%
NAVI	GPT-4o	13.3%	10.2%
OmniAgent	GPT-4o	19.5%	8.2%
Agent S	GPT-4o	18.2%	12.2%
Operator	computer-use	20.8%	14.3%
UFO^2-base	GPT-4o	23.4%	16.3%
UFO^2-base	o1	25.3%	16.3%
UFO^2	**GPT-4o**	**27.9%**	**28.6%**
UFO^2	**o1**	**30.5%**	**32.7%**

Table 2: SR breakdown by application type on WAA and OSWorld-W.

Agent	**Model**	**WAA**						**OSWorld-W**	
		Office	Web Browser	Windows System	Coding	Media & Video	Windows Utils	Office	Cross-App
UFO	GPT-4o	0.0%	23.3%	33.3%	29.2%	33.3%	8.3%	18.5%	4.5%
NAVI	GPT-4o	0.0%	20.0%	29.2%	9.1%	25.3%	0.0%	18.5%	0.0%
OmniAgent	GPT-4o	0.0%	27.3%	33.3%	27.3%	30.3%	8.3%	14.8%	0.0%
Agent S	GPT-4o	0.0%	13.3%	45.8%	29.2%	19.1%	**22.2%**	22.2%	0.0%
Operator	computer-use	**7%**	26.7%	29.2%	29.2%	28.6%	8.3%	22.2%	4.5%
UFO^2-base	GPT-4o	2.3%	36.7%	29.2%	41.7%	33.3%	0.0%	22.2%	**9.1%**
UFO^2-base	o1	2.3%	30.0%	37.5%	50.0%	33.3%	8.3%	22.2%	**9.1%**
UFO^2	GPT-4o	4.7%	30.0%	41.7%	**58.3%**	33.3%	8.3%	44.4%	**9.1%**
UFO^2	o1	4.7%	**40.0%**	**45.8%**	50.0%	**38.1%**	16.7%	**51.9%**	**9.1%**

These significant performance improvements clearly underscore the advantages of UFO^2's deep integration with OS-level mechanisms and its unified system architecture. While prior CUAs primarily emphasize model-level optimization or singular reliance on visual interfaces, our results demonstrate that robust, system-level orchestration—combining structured OS APIs, specialized application knowledge, and hybrid GUI–API interaction—is instrumental in achieving higher task reliability and broader automation coverage. Crucially, even a general-purpose, less-specialized model like GPT-4o can surpass highly specialized CUAs (such as Operator) when integrated within the comprehensive UFO^2 framework. This insight reinforces the value of architectural design and OS integration as key drivers of practical, deployable desktop automation solutions.

Performance Breakdown. Table 2 presents a detailed breakdown of success rates (SR) by application type on the WAA and OSWorld-W benchmarks, enabling deeper understanding of where UFO^2 achieves particularly strong results and identifying areas for further system-level improvements. Across multiple categories, UFO^2 consistently demonstrates superior performance compared to baseline CUAs, particularly in application scenarios demanding deeper OS integration or sophisticated multi-step task execution.

Notably, UFO^2 excels in tasks involving web browsers and coding environments. For instance, the strongest configuration (UFO^2 with o1) attains an impressive 40.0% SR for web browser tasks—markedly outperforming the next-best baseline (OmniAgent) by over 12%. Similarly, in coding-related workflows, UFO^2 (GPT-4o) achieves the highest SR of 58.3%, significantly exceeding all competing CUAs. These results underscore the effectiveness of UFO^2's hybrid GUI-API approach and continuous knowledge integration, which enable more precise action inference, reduce brittleness due to GUI changes, and substantially elevate reliability in multi-step workflows.

The breakdown further reveals a clear correlation between application complexity, popularity, and system-level support. Tasks involving LibreOffice (in the Office category of WAA) uniformly yield lower SRs across all evaluated CUAs, largely due to inadequate adherence to accessibility standards and incomplete UIA support. Conversely, OSWorld-W tasks predominantly utilize Microsoft 365 Office applications, which offer

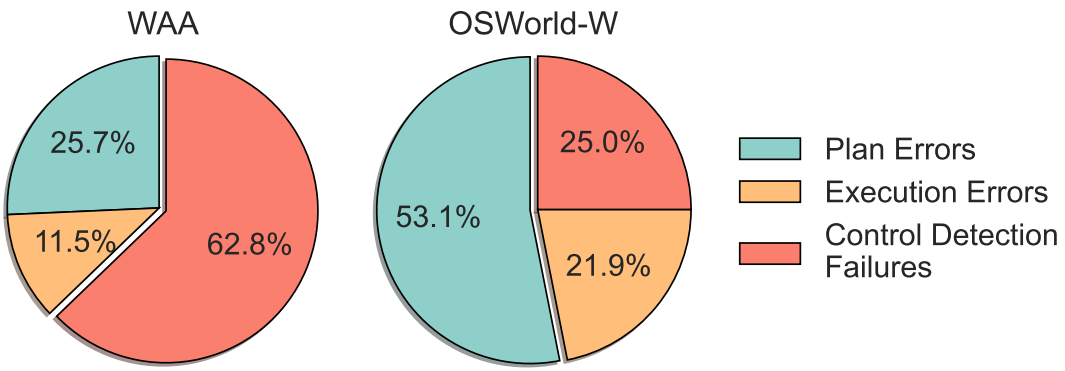

Figure 19: Error analysis of UFO^2-base (GPT-4o) on the two benchmarks.

richer OS-native APIs and structured accessibility data, resulting in improved SRs (up to 51.9% for UFO^2-o1). This discrepancy highlights the critical role that robust OS-level integration and API availability play in achieving high-quality desktop automation.

Cross-application tasks, especially prominent in OSWorld-W, present an even greater challenge. Such tasks inherently require sophisticated task decomposition and robust inter-agent coordination, pushing CUAs—and even human users—to their limits. Here, UFO^2's multiagent architecture, led by the centralized HostAgent and specialized AppAgents, demonstrates notable promise, outperforming other baselines with a 9.1% SR. Although performance remains relatively modest, it clearly illustrates the strength of systematic multiagent collaboration and centralized orchestration in addressing complex scenarios that cross traditional application boundaries.

Overall, these detailed breakdown results validate the system-level design principles of UFO^2, particularly its emphasis on deep OS and application-specific integration, multiagent coordination, and flexible action orchestration. While there remains significant potential for further enhancements in niche or less-supported application domains (*e.g.*, custom or legacy software with limited API availability), UFO^2's current architecture already provides a substantial, measurable improvement in practical, real-world desktop automation tasks.

Error Analysis. To systematically understand the limitations of UFO^2 and identify opportunities for further improvement, we conducted a detailed manual review of all failure cases for UFO^2-base (GPT-4o) on both benchmarks. Following a classification framework similar to Agashe *et al.*, Agashe et al., each failure was categorized into one of three distinct system-level categories:

- **Plan Errors:** Failures arising from inadequate high-level task understanding, typically reflected by incomplete or incorrect action plans. These errors indicate gaps in the agent's task comprehension or insufficient grounding in application-specific workflows.
- **Execution Errors:** Cases where the high-level plan is reasonable but the execution is flawed (*e.g.*, selecting an incorrect control, performing unintended actions). Execution errors often stem from inaccurate visual reasoning, incorrect associations between GUI elements and actions, or erroneous inference by the LLM.
- **Control Detection Failures:** Instances where the agent fails to detect or identify critical GUI controls required to complete a task, usually due to non-standard or custom-rendered UI elements that are not fully accessible via standard OS APIs.

Figure 19 summarizes our findings for UFO^2-base. On the WAA benchmark, more than 62% of failures were attributed to *Control Detection Failures*, highlighting significant gaps in standard UIA API coverage—especially for third-party applications (*e.g.*, LibreOffice) that do not strictly adhere to accessibility standards. Conversely, the OSWorld-W benchmark exhibited a higher incidence of *Plan Errors*, underscor-

Table 3: Comparison of SR and CRR across control detection mechanisms.

Control Detector	**Model**	**WAA**		**OSWorld-W**	
		SR	CRR	SR	CRR
UIA	GPT-4o	23.4%	-	22.4%	-
OmniParser-v2	GPT-4o	26.6%	7.0%	14.3%	0%
Hybrid	**GPT-4o**	**26.6%**	**9.9%**	**22.4%**	**12.5%**
UIA	o1	25.3%	-	24.5%	-
OmniParser-v2	o1	20.8%	7.0%	14.3%	0%
Hybrid	**o1**	**27.9%**	**9.9%**	**28.6%**	**25.0%**

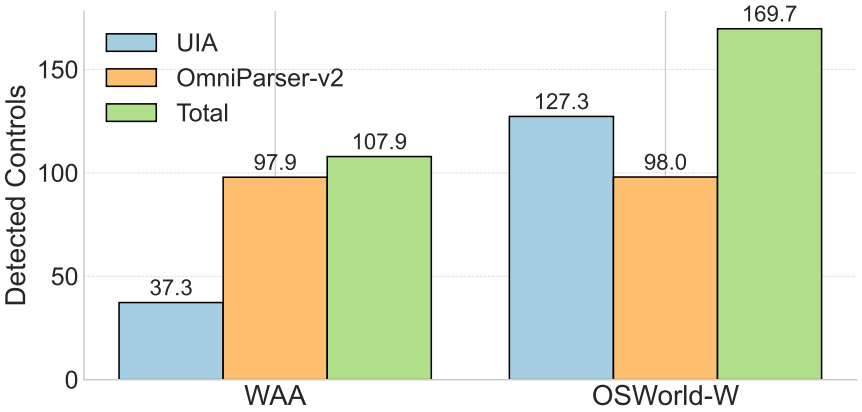

Figure 20: The number of detected controls of different approaches.

ing that tasks in this set frequently involve more complex workflows, necessitating deeper domain knowledge or advanced contextual reasoning capabilities beyond simple visual recognition.

These observations provide concrete evidence of specific system-level shortcomings, directly motivating the enhancements incorporated into the complete version of UFO^2. The high frequency of *Control Detection Failures* validates our choice of adopting a hybrid GUI detection pipeline that supplements standard UIA data with advanced visual grounding techniques. Similarly, the prevalence of *Plan Errors* underscores the critical role of integrating richer external documentation, domain-specific knowledge bases, and application-level APIs to strengthen task understanding and action inference. In the subsequent sections, we explicitly demonstrate how these incremental system-level improvements progressively mitigate each identified category of errors, thereby substantially boosting UFO^2's overall task completion effectiveness.

6.3 Evaluation on Hybrid Control Detection

As shown in Figure 19, a considerable fraction of failures arise from *Control Detection Failures*, where non-standard UI elements do not comply with UIA guidelines. To quantify the effectiveness of different detection strategies, we compare UIA-only, OmniParser-v2–only, and our hybrid method (Section 3.4). We introduce a *Control Recovery Ratio (CRR)* to measure how many UIA-only failures are "recovered" (*i.e.*, become successful completions) under OmniParser or the hybrid approach.

Table 3 presents the results on both benchmarks, across multiple model configurations. The hybrid method consistently outperforms either UIA-only or OmniParser-only settings, raising the overall success rate and converting up to 9.86% of previously irrecoverable cases into completions. This gain highlights the complementary strengths of the two detection pipelines, as the hybrid approach bridges coverage gaps in UIA while avoiding OmniParser's limitations in more standardized GUIs.

In Figure 20, we report the average number of controls detected from each source (UIA, OmniParser-v2, and the merged set) under the hybrid approach. Owing to differences in application coverage, the total number

Table 4: APIs supported across Office applications.

API	Application	Description
`select_text`	Word	Select matched text in the document.
`select_paragraph`	Word	Select a paragraph in the document.
`set_font`	Word	Set the font size and style of selected text.
`save_as`	Word	Save the current document to a desired format.
`insert_excel_table`	Excel	Insert a table at the desired position.
`select_table_range`	Excel	Select a range within a table.
`reorder_column`	Excel	Reorder columns of a table.
`save_as`	Excel	Save the current sheet to a desired format.
`set_background_color`	PowerPoint	Set the background color of slide(s).
`save_as`	PowerPoint	Save the current presentation to a desired format.

Table 5: Performance comparison of GUI-only vs. GUI + API actions.

Action	Model	SR	PRR	ERR	CRR	ACS
GUI-only	GPT-4o	16.3%	-	-	-	13.8
GUI+API	**GPT-4o**	**22.4%**	**5.9%**	**14.3%**	**25.0%**	**12.9**
GUI-only	o1	16.3%	-	-	-	16.0
GUI+API	**o1**	**24.5%**	**17.7%**	**0.0%**	**12.5%**	**6.6**

of detected controls is generally higher in OSWorld-W than in WAA. Notably, both UIA and OmniParser-v2 identify substantial subsets of controls, and after merging, 27.9% and 56.7% of OmniParser-v2 detections are discarded due to overlap with UIA. These observations indicate that OmniParser-v2 provides a valuable complement to UIA by recovering nonstandard or custom elements. At the same time, the merging step removes redundancies and prevents double-counting, ultimately reducing control detection failures in the hybrid scheme.

6.4 Effectiveness of GUI + API Integration

We now evaluate how unifying API-based actions with standard GUI interactions in the `Puppeteer` impacts performance (Section 3.5). To do so, we focus on the *27* office-related tasks in OSWorld and manually develop *12* APIs for Word, Excel, and PowerPoint. These applications provide COM interfaces that facilitate the creation of custom functions, making them ideal exemplars for deeper OS- and application-level integration. Importantly, many of these operations would require cumbersome multi-step GUI procedures but become straightforward single calls via these APIs (*e.g.*, select paragraphs). Table 4 details the implemented APIs.

Table 5 compares *(i)* overall Success Rate (SR), *(ii)* Plan Error Recovery rate (PRR), *(iii)* Execution Error Recovery rate (ERR), *(iv)* Control Detection Failure Recovery rate (CRR), and *(v)* Average Completion Steps (ACS) for two configurations: GUI-only versus GUI + API. We calculate ACS on the subset of tasks that both configurations successfully complete, ensuring a fair comparison.

Figure 21: A case study comparing the completion of the same task using GUI-only actions vs. GUI + API actions.

The results show that integrating API actions boosts SR for both GPT-4o (+6.1%) and o1 (+8.2%), underscoring the effectiveness of mixing GUI and API interactions. Notably, GPT-4o benefits most from APIs in recovering from *Control Detection Failures* by circumventing unannotated GUI elements. In contrast, o1 more frequently addresses *Plan Errors* through API "shortcuts", reflecting the model's stronger reasoning capabilities and preference for concise solutions.

Beyond higher success rates, GUI + API also reduces the effort required to complete tasks. UFO^2 achieves a 6.5% step savings with GPT-4o and an impressive 58.5% reduction for o1 on identical tasks. The latter improvement stems from o1's ability to strategically call API functions, bypassing multiple GUI-based steps. Overall, these findings confirm the advantages of mixing GUI automation with API calls, both in terms of robustness and efficiency, and showcase the importance of deep system integration for desktop automation.

Case Study. To illustrate how the GUI + API approach streamlines task execution, Figure 21 shows the completion trajectory for exporting an Excel file to CSV format in a case of OSWorld-W, using either GUI-only or GUI + API interactions. Although both configurations eventually succeed, the GUI-only setting requires five steps to open the Save dialog, select the file format, and confirm the action. In contrast, a single call to the `save_as` API completes the task immediately. Beyond improving efficiency, this one-step solution also reduces the risk of compounding errors across multiple GUI interactions—a clear demonstration of the advantages of deeper OS and application-level integration.

6.5 Continuous Knowledge Integration Evaluation

We next evaluate the impact of continuous knowledge integration (Section 3.6) on UFO^2's performance. Specifically, we augment UFO^2 with external documentation and execution-derived insights to dynamically improve its domain understanding without retraining. We create *34* help documents tailored to benchmark tasks, each containing precise step-by-step instructions, enabling UFO^2 to retrieve the most relevant guidance (maximum of one per task) at runtime. Additionally, we implement an automated pipeline that summarizes successful execution trajectories—validated by our Task Evaluator and archives them into a retrievable knowledge database. For subsequent tasks, UFO^2 dynamically retrieves up to three relevant past execution

Table 6: Performance comparison with and without knowledge integration.

Knowledge Enhancement	**Model**	**WAA**		**OSWorld-W**	
		SR	PRR	SR	PRR
None	GPT-4o	23.4%	-	22.4%	-
Help Document	GPT-4o	26.6%	10.34%	26.5%	11.8%
Self-Experience	GPT-4o	26.6%	13.79%	24.5%	11.8%
None	o1	25.3%	-	24.5%	-
Help Document	o1	27.9%	3.5%	28.5%	17.7%
Self-Experience	o1	20.8%	13.79%	26.5%	17.7%

Table 7: The SR and ACS comparison between single action and speculative multi-action mode.

Action Execution	**Model**	**WAA**			**OSWorld-W**		
		SR	**ACS**	**Success Subset**	**SR**	**ACS**	**Success Subset**
Single	GPT-4o	23.4%	10.00	30	22.4%	13.30	10
Speculative	GPT-4o	23.4%	**8.78**		24.5%	**7.40**	
Single	o1	25.3%	9.95	32	24.5%	6.80	10
Speculative	o1	24.7%	**8.85**		26.5%	**3.30**	

logs to guide task planning and execution. Given that knowledge integration primarily addresses failures arising from insufficient planning (*Plan Errors*), we employ the *Plan Recovery Ratio (PRR)* to measure the proportion of previously failed planning cases successfully resolved by integrating new knowledge.

Table 6 compares the overall SR and PRRs across two benchmarks, highlighting significant performance improvements attributable to knowledge integration. Both live help-document retrieval and self-experience summarization yield noticeable gains, reducing planning failures by up to 17.7%. Notably, self-experience enhancements using the stronger model (o1) achieve consistent improvements across both benchmarks, underscoring the efficacy of leveraging prior successes for adaptive improvement. While help documents occasionally result in modest gains, their effectiveness depends on task complexity and document specificity.

These findings underscore the value of systematic knowledge integration, demonstrating that continuous augmentation of the agent's knowledge base can substantially enhance its robustness, scalability, and adaptability in real-world deployments. Moreover, as UFO^2 continues to accumulate execution experience and documentation over time, it inherently evolves toward higher reliability and improved autonomy, marking a clear path for ongoing enhancement in desktop automation.

6.6 Effectiveness of Speculative Multi-Action Execution

Next, we evaluate how speculative multi-action execution (Section 3.7) affects task completion rates and efficiency. Table 7 compares two modes for UFO^2: generating and executing one action per inference (*single-action*) versus inferring multiple consecutive actions in one step (*speculative multi-action*). To ensure a fair comparison, we compute the Average Completion Steps (ACS) only on the subset of tasks that both modes complete successfully.

The results show that speculative multi-action execution retains a comparable Success Rate (SR) to single-action mode while notably cutting the average steps—by up to 10% on WAA and an impressive 51.5% on OSWorld-W. Because each step requires an LLM call, reducing the number of steps significantly lowers both latency and cost. This finding confirms that speculative multi-action planning enhances efficiency without compromising reliability, further highlighting UFO^2's ability to optimize resource utilization in practical desktop automation.

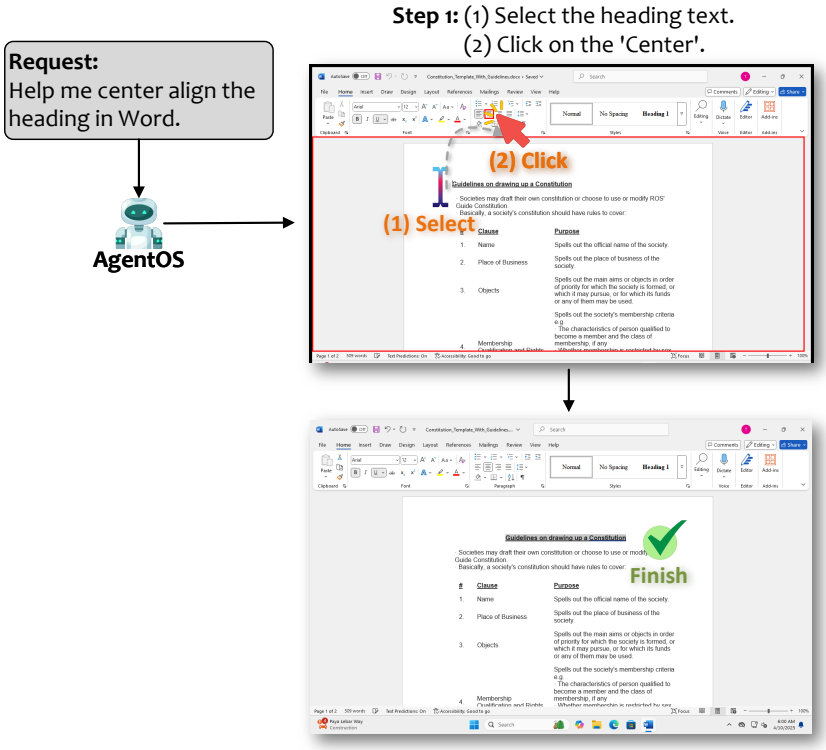

Figure 22: A case study of the successful speculative multi-action execution.

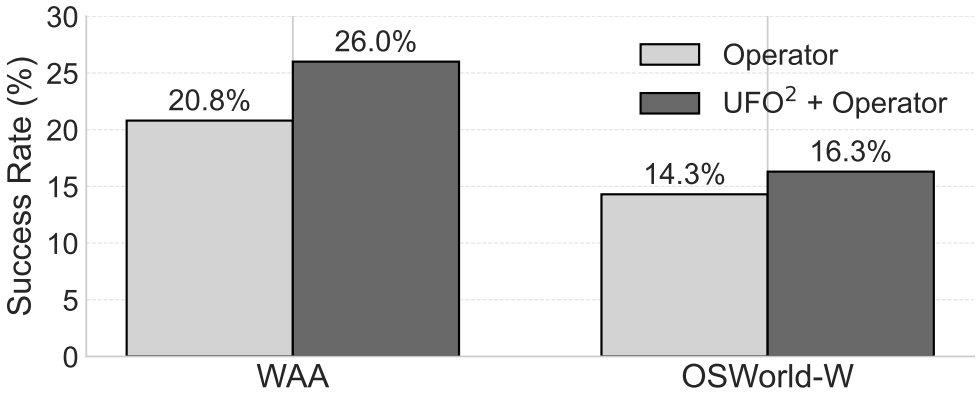

Figure 23: Comparison of Operator vs. UFO^2 + Operator on WAA and OSWorld-W.

Case Study. Figure 22 illustrates how speculative multi-action execution operates in practice. When the user requests that UFO^2 center-align a heading in a Word document, the sequence of steps would typically require selecting the heading text and then clicking the `Center` icon. These actions that are sequentially dependent but do not interfere with each other. Instead of treating each action as a separate LLM inference, UFO^2 predicts both actions in a single step, leveraging speculative multi-action planning. Consequently, it completes the task with just one LLM call, significantly enhancing efficiency while maintaining accuracy.

6.7 Operator as an AppAgent

To demonstrate the "Everything-as-an-AppAgent" capability (Section 5.3), we conducted an experiment where UFO^2's HostAgent orchestrator uses *only* Operator as the AppAgent. In other words, all native AppAgents were disabled, leaving Operator to accept subtasks and communicate via the standard UFO^2

Table 8: Step count statistic for UFO^2.

Agent	Model	WAA				OSWorld-W			
		HostAgent	AppAgent	Total	Success Subset	HostAgent	AppAgent	Total	Success Subset
UFO^2-base	GPT-4o	**2.21**	8.11	10.32	31	**1.80**	10.80	12.60	7
UFO^2	GPT-4o	2.32	**7.89**	**10.21**		2.80	**7.20**	**10.00**	
UFO^2-base	o1	2.14	7.00	9.14	34	2.50	8.83	11.33	8
UFO^2	o1	**2.00**	**4.05**	**6.05**		**2.00**	**3.50**	**5.50**	

messaging protocol. The only adjustment to Operator's perception layer was restricting it to screenshots of the selected application window, rather than the full desktop.

Figure 23 shows that UFO^2 + Operator achieves higher success rates than running Operator alone, particularly on WAA (26.0% vs. 20.8%). We attribute these gains to three key factors. First, the HostAgent breaks down complex user instructions into clearer, single-application subtasks, reducing ambiguity. Second, HostAgent messages include additional tips that improve Operator's decision making. Finally, limiting Operator's view to a single, active application window reduces visual noise and simplifies control detection. Taken together, these results underscore the benefits of UFO^2's multiagent design, while demonstrating how "Everything-as-an-AppAgent" can elevate the performance of an existing CUA.

6.8 Efficiency Analysis

To comprehensively understand the performance characteristics of UFO^2, we conducted detailed profiling of task execution efficiency, focusing specifically on step count and latency.

Step Count Profiling. Table 8 summarizes the average number of execution steps performed by the HostAgent and AppAgents across both benchmark suites. The steps reported are computed on tasks that were successfully completed across all model configurations, ensuring fair comparisons. Two key insights emerge:

First, the fully integrated UFO^2 configuration consistently reduces the average number of steps required compared to the baseline (UFO^2-base), achieving reductions of up to 50%. This substantial efficiency gain demonstrates how deep OS integration, specifically the hybrid GUI–API action orchestration and advanced control detection strategies, significantly streamline execution paths.

Second, utilizing a more powerful reasoning model (*e.g.*,, *o1* versus GPT-4o) further reduces step counts, indicating that enhanced reasoning capability enables the agent to identify and exploit more efficient action sequences. For instance, stronger models can better leverage direct API interactions or avoid unnecessary intermediate GUI interactions. This underscores the complementary role of both robust system integration and advanced LLM reasoning in minimizing execution overhead.

Latency Breakdown. Figure 24 provides a detailed breakdown of average latency per execution step in UFO^2, separated into five key phases: *(i) Screenshot Capture*, *(ii) Control Detection* via UIA APIs (and) OmniParser-v2, *(iii) Prompt Preparation*, including retrieval of relevant help documents and historical execution experiences, *(iv) LLM Inference*, and *(v) Action Execution* on target applications.

Across all configurations, the LLM inference phase dominates total latency, averaging around 10 seconds per inference. This bottleneck highlights a clear opportunity for optimization by deploying smaller, specialized models or employing more powerful inference hardware—strategies that remain viable but are beyond our current evaluation scope.

Excluding LLM inference overhead, the baseline system (UFO^2-base) achieves highly efficient execution, incurring around 10 seconds per step on average. In contrast, the fully integrated UFO^2 incurs only an additional 1 second per step for its hybrid control detection pipeline, largely due to OmniParser-v2 visual parsing. This added overhead represents a deliberate trade-off, significantly enhancing the robustness and accuracy of GUI control detection at a modest latency cost.

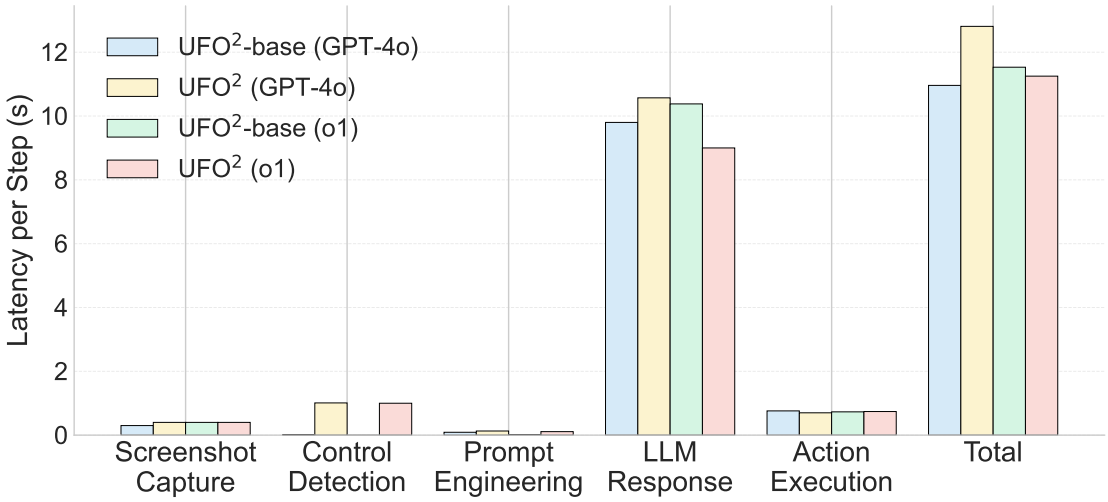

Figure 24: Average time cost per-stage of a single execution step.

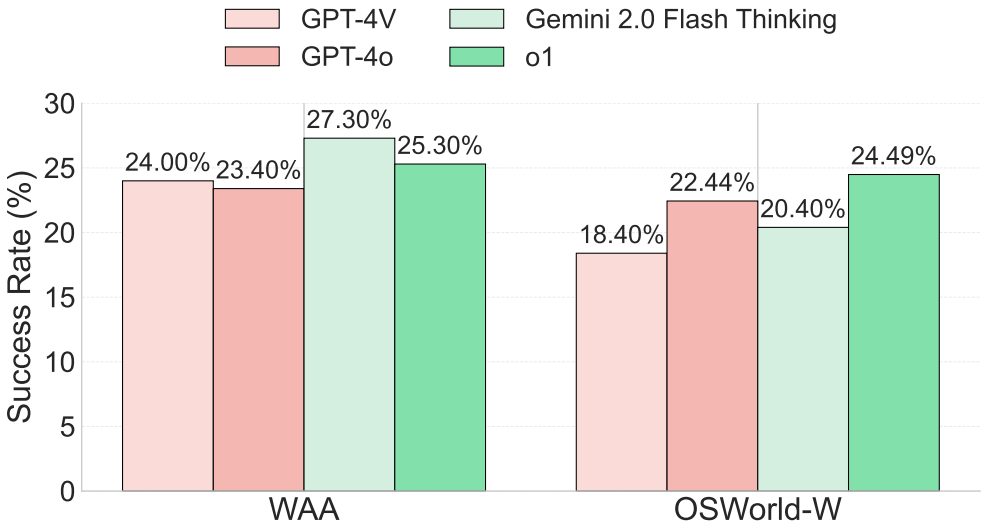

Figure 25: Comparison of different LLMs used in UFO^2 and UFO^2-base on WAA and OSWorld-W.

Taken together, these results indicate that the substantial reduction in total steps required per task ensures overall task completion times remain practical (approximately 1 minute per task). These profiling insights reinforce that UFO^2's comprehensive system-level integration balances latency, accuracy, and efficiency, offering a scalable and performant solution for real-world desktop automation.

6.9 Model Ablation

Table 25 compares the performance of UFO^2 and UFO^2-base across four large language models. GPT-4V and GPT-4o generate direct answers without exposing an explicit CoT, whereas Gemini 2.0 (Flash Thinking) and o1 embed reasoning steps internally before producing final outputs. Overall, models with built-in reasoning typically achieve higher success rates (SR), highlighting the value of more deliberative or CoT-driven processes in desktop automation Lu et al. (2025).

This result underscores a promising direction for CUAs: fine-tuning advanced reasoning models specifically for desktop automation tasks. By allowing agents to formulate and refine multi-step plans—especially when integrated with deeper OS-level signals—UFO^2 can address complex or ambiguous situations more reliably.

As LLM-based reasoning continues to mature, we expect further gains in both accuracy and generality from UFO^2's model-agnostic design.

7 Discussion & Future Work

Latency and Responsiveness. UFO^2 currently invokes LLM inference at each decision step, incurring a latency typically ranging from several seconds to tens of seconds per action. Despite various engineering optimizations, complex tasks comprising multiple sequential actions can accumulate to execution times of 1–2 minutes, which remains acceptable but still inferior to skilled human performance. To alleviate user-perceived delay, we introduced the Picture-in-Picture (PiP) interface, enabling UFO^2 to execute tasks unobtrusively within an isolated virtual desktop, thus substantially reducing user inconvenience during longer-running automations. In future work, we aim to further lower latency by investigating the deployment of specialized, lightweight Large Action Models (LAMs) Wang et al. (2024b), optimized for task-specific inference to enhance both responsiveness and scalability.

Closing the Gap with Human-Level Performance. Our comprehensive evaluations indicate that UFO^2, while robust and effective, does not yet consistently achieve human-level performance across all Windows applications. Bridging this gap will necessitate advances primarily along two critical dimensions. First, enhancing foundational visual-language models through fine-tuning on extensive, diverse GUI interaction datasets will significantly improve agents' capabilities and generalization across varied applications. Second, tighter integration with OS-level APIs, native application interfaces, and comprehensive, structured documentation sources will deepen contextual understanding and bolster execution reliability. Given UFO^2's modular architecture, these enhancements can be incrementally adopted, continuously refining performance towards human-equivalent proficiency across diverse application scenarios.

Generalization Across Operating Systems. While UFO^2 targets Windows OS due to its widespread market adoption (over 70% market share[2]), the modular, layered system design facilitates straightforward adaptation to other desktop platforms, such as Linux and macOS. The core approach—leveraging accessibility frameworks like Windows UI Automation (UIA)—has direct analogs on Linux (AT-SPI Linux From Scratch) and macOS (Accessibility API Apple Inc. (2025)). Consequently, the existing design principles, agent decomposition strategy, and unified GUI-API orchestration model generalize naturally, enabling rapid, platform-specific customizations. Exploring cross-platform deployments will be an important area of future work, potentially laying the groundwork for a unified ecosystem of desktop automation solutions spanning diverse operating environments.

8 Related Work

Integrating LLMs into OS represents a growing, yet nascent area of research. In this section, we discuss prior work that intersects with our research on system-level integration of multimodal LLM-based desktop automation agents.

8.1 Computer-Using Agents (CUAs)

Recent advancements in multimodal LLMs have significantly accelerated the development of *Computer-Using Agents* (CUAs), which automate desktop workflows by simulating GUI interactions at the OS level. Early pioneering systems, such as UFO Zhang et al. (2024b), leveraged multimodal models (*e.g.*, GPT-4V Yang et al. (2023b)) alongside UIA APIs to interpret graphical interfaces and execute complex tasks via natural language instructions. UFO notably introduced a multi-agent architecture, enhancing the reliability and capability of CUAs to handle cross-application and long-term workflows.

Subsequent efforts have focused primarily on refining underlying multimodal models and extending platform capabilities. For example, CogAgent Hong et al. (2024), built upon the vision-language model CogVLM Wang

[2] `https://gs.statcounter.com/os-market-share/desktop/worldwide/\#monthly-202301-202301-bar`

et al. (2024c), specialized in GUI understanding across multiple platforms (PC, web, Android), representing one of the earliest dedicated multimodal CUAs. Industry interest has similarly accelerated with Anthropic's Claude-3.5 (Computer Use) Anthropic (2024), an agent relying entirely on screenshot-based GUI interactions, and OpenAI's Operator OpenAI (2025), which significantly improved desktop automation performance through advanced multimodal reasoning.

However, these existing CUAs remain largely prototype demonstrations, often lacking deep integration with the OS and native application capabilities. In contrast, our work in UFO^2 directly addresses these fundamental system-level limitations through a modular AgentOS architecture, deep OS and API integration, hybrid GUI detection, and a non-disruptive execution model, bridging the gap between conceptual CUAs and practical desktop automation.

8.2 LLMs for Operating Systems

Another promising research direction involves embedding LLMs directly within OS architectures, aiming to substantially enhance automation, adaptability, and usability. Ge *et al.*, first proposed the conceptual framework AIOS Ge et al. (2023), positioning an LLM at the center of OS design to orchestrate high-level user interactions and automated decision-making. In their vision, agents resemble OS applications, each exposing specialized capabilities accessible via natural language, effectively enabling users to "program" their OS intuitively.

Building on this conceptual foundation, Mei *et al.*, Mei et al. (2024) realized AIOS as a concrete prototype, encapsulating LLM interactions and tool APIs within a privileged OS kernel. This design provides core OS functionalities such as process scheduling, memory management, I/O handling, and access control, leveraging LLMs to simplify agent development through a dedicated SDK. Rama *et al.*, Rama et al. (2025) extended this paradigm, introducing semantic file management capabilities directly within traditional OS environments through AIOS-based agents, further demonstrating practical system-level integration.

Complementing these high-level OS integrations, AutoOS Chen et al. (2024b) applied LLMs for automatic tuning of kernel-level parameters in Linux, achieving substantial efficiency gains through autonomous exploration and optimization. This highlights another dimension where LLM integration can directly enhance core system performance and management.

Collectively, these research efforts illustrate an emerging paradigm shift where LLMs become integral components of operating systems, enabling powerful automation, enhanced user interaction, and adaptive system behavior. Our work with UFO^2 extends this line of research specifically to desktop automation, offering a deeply integrated, scalable, and practical AgentOS that leverages multimodal LLMs in conjunction with robust OS-level mechanisms.

9 Conclusion

We introduced **UFO^2**, a practical, OS-integrated Windows desktop automation AgentOS that transforms CUAs from conceptual prototypes into robust, user-oriented solutions. Unlike prior CUAs, UFO^2 leverages deep system-level integration through a modular, multiagent architecture consisting of a centralized HostAgent and application-specialized AppAgents. Each AppAgent seamlessly combines GUI interactions with native APIs and continually integrates application-specific knowledge, substantially improving reliability and execution efficiency. UFO^2 can operate on a PiP virtual desktop interface further enhances usability, enabling concurrent user-agent workflows without interference.

Our comprehensive evaluation across over 20 real-world Windows applications demonstrated that UFO^2 achieves significant improvements in robustness, accuracy, and scalability compared to state-of-the-art CUAs. Notably, by coupling our integrated framework with robust OS-level features, even less specialized foundation models (*e.g.*, GPT-4o) surpass specialized CUAs such as Operator.

References

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. In *The Thirteenth International Conference on Learning Representations*.

Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku, 2024. URL `https://www.anthropic.com/news/3-5-models-and-computer-use`. Accessed: 2024-10-26.

Automation Anywhere. Automation anywhere: Automation 360 platform, 2025. URL `https://www.automationanywhere.com/products/automation-360`. Accessed: 2025-03-18.

Apple Inc. Accessibility api. `https://developer.apple.com/documentation/accessibility/accessibility-api`, 2025. Accessed: 2025-03-31.

Rogerio Bonatti, Dan Zhao, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Keunho Jang, et al. Windows agent arena: Evaluating multi-modal os agents at scale. In *NeurIPS 2024 Workshop on Open-World Agents*.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yaochen Wang, Yinuo Liu, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. In *Forty-first International Conference on Machine Learning*, 2024a.

Huilai Chen, Yuanbo Wen, Limin Cheng, Shouxu Kuang, Yumeng Liu, Weijia Li, Ling Li, Rui Zhang, Xinkai Song, Wei Li, et al. Autoos: make your os more powerful by exploiting large language models. In *Forty-first International Conference on Machine Learning*, 2024b.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9313–9332, 2024.

Ruomeng Ding, Chaoyun Zhang, Lu Wang, Yong Xu, Minghua Ma, Wei Zhang, Si Qin, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. Everything of thoughts: Defying the law of penrose triangle for thought generation. In *Findings of the Association for Computational Linguistics ACL 2024*, pp. 1638–1662, 2024.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, et al. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 1107–1128, 2024.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2, 2023.

Yingqiang Ge, Yujie Ren, Wenyue Hua, Shuyuan Xu, Juntao Tan, and Yongfeng Zhang. Llm as os, agents as apps: Envisioning aios, agents and the aios-agent ecosystem. *arXiv preprint arXiv:2312.03815*, 2023.

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.

David N Gray, John Hotchkiss, Seth LaForge, Andrew Shalit, and Toby Weinberg. Modern languages and microsoft's component object model. *Communications of the ACM*, 41(5):55–65, 1998.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu, and Chaoyang He. Llm multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024.

Peter Hofmann, Caroline Samp, and Nils Urbach. Robotic process automation. *Electronic markets*, 30(1): 99–106, 2020.

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*, pp. 14281–14290, 2024.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278*, 2025.

Yuxuan Jiang, Chaoyun Zhang, Shilin He, Zhihao Yang, Minghua Ma, Si Qin, Yu Kang, Yingnong Dang, Saravan Rajmohan, Qingwei Lin, et al. Xpert: Empowering incident management with query recommendations via large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Ido Levy, Ben Wiesel, Sami Marreed, Alon Oved, Avi Yaeli, and Segev Shlomov. St-webagentbench: A benchmark for evaluating safety and trustworthiness in web agents. *arXiv preprint arXiv:2410.06703*, 2024.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020.

Linux From Scratch. At-spi - assistive technology service provider interface. `https://www.linuxfromscratch.org/blfs/view/5.1/gnome/at-spi.html`. Accessed: 2025-03-31.

Jun Liu, Chaoyun Zhang, Jiaxu Qian, Minghua Ma, Si Qin, Chetan Bansal, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Large language models can deliver accurate and interpretable time series anomaly detection. *arXiv preprint arXiv:2405.15370*, 2024.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent. *arXiv preprint arXiv:2408.00203*, 2024.

Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Guanjing Xiong, and Hongsheng Li. Ui-r1: Enhancing action prediction of gui agents by reinforcement learning, 2025. URL `https://arxiv.org/abs/2503.21620`.

Man Luo, Xin Xu, Yue Liu, Panupong Pasupat, and Mehran Kazemi. In-context learning with retrieved demonstrations for language models: A survey. *Transactions on Machine Learning Research*.

Somayya Madakam, Rajesh M Holmukhe, and Durgesh Kumar Jaiswal. The future digital work force: robotic process automation (rpa). *JISTEM-Journal of Information Systems and Technology Management*, 16:e201916001, 2019.

Kai Mei, Xi Zhu, Wujiang Xu, Wenyue Hua, Mingyu Jin, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. Aios: Llm agent operating system. *arXiv preprint arXiv:2403.16971*, 2024.

Microsoft. Microsoft power automate, 2025. URL `https://www.microsoft.com/en-us/power-platform/products/power-automate/`. Accessed: 2025-03-18.

Karissa Miller and Mahmoud Pegah. Virtualization: virtually at the desktop. In *Proceedings of the 35th annual ACM SIGUCCS fall conference*, pp. 255–260, 2007.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11048–11064, 2022.

Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. Screenagent: a vision language model-driven computer control agent. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pp. 6433–6441, 2024.

OpenAI. Computer-using agent: Introducing a universal interface for ai to interact with the digital world. 2025. URL `https://openai.com/index/computer-using-agent`.

Dhanya Pramod. Robotic process automation for industry: adoption status, benefits, challenges and research agenda. *Benchmarking: an international journal*, 29(5):1562–1586, 2022.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*, 2023.

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.

Balaji Rama, Kai Mei, and Yongfeng Zhang. Cerebrum (aios sdk): A platform for agent development, deployment, distribution, and discovery, 2025. URL `https://arxiv.org/abs/2503.11444`.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 3982–3992, 2019.

Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. Real-time flying object detection with yolov8. *arXiv preprint arXiv:2305.09972*, 2023.

Julia Siderska, Lili Aunimo, Thomas Süße, John von Stamm, Damian Kedziora, and Suraya Nabilah Binti Mohd Aini. Towards intelligent automation (ia): literature review on the evolution of robotic process automation (rpa), its challenges, and future trends. *Engineering Management in Production and Services*, 15(4), 2023.

Yueqi Song, Frank Xu, Shuyan Zhou, and Graham Neubig. Beyond browsing: Api-based web agents. *arXiv preprint arXiv:2410.16464*, 2024.

UiPath. Uipath: Automation platform, 2025. URL `https://www.uipath.com/`. Accessed: 2025-03-18.

Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of inter-process communication mechanisms. *Architecture*, 86(64), 2015.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.

Lu Wang, Fangkai Yang, Chaoyun Zhang, Junting Lu, Jiaxu Qian, Shilin He, Pu Zhao, Bo Qiao, Ray Huang, Si Qin, et al. Large action models: From inception to implementation. *arXiv preprint arXiv:2412.10047*, 2024b.

Siyuan Wang, Zhuohan Long, Zhihao Fan, Xuan-Jing Huang, and Zhongyu Wei. Benchmark self-evolving: A multi-agent framework for dynamic llm evaluation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 3310–3328, 2025.

Weihan Wang, Qingsong Lv, Wenmeng Yu, Wenyi Hong, Ji Qi, Yan Wang, Junhui Ji, Zhuoyi Yang, Lei Zhao, Song XiXuan, et al. Cogvlm: Visual expert for pretrained language models. *Advances in Neural Information Processing Systems*, 37:121475–121499, 2024c.

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.

Bin Xiao, Haiping Wu, Weijian Xu, Xiyang Dai, Houdong Hu, Yumao Lu, Michael Zeng, Ce Liu, and Lu Yuan. Florence-2: Advancing a unified representation for a variety of vision tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4818–4829, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Jing Hua Toh, 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.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023a.

Zhengyuan Yang, Linjie Li, Kevin Lin, Jianfeng Wang, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. The dawn of lmms: Preliminary explorations with gpt-4v (ision). *arXiv preprint arXiv:2309.17421*, 9(1): 1, 2023b.

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.

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*, 2024a.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. Ufo: A ui-focused agent for windows os interaction. *arXiv preprint arXiv:2402.07939*, 2024b.

Chaoyun Zhang, Zicheng Ma, Yuhao Wu, Shilin He, Si Qin, Minghua Ma, Xiaoting Qin, Yu Kang, Yuyi Liang, Xiaoyu Gou, et al. Allhands: Ask me anything on large-scale verbatim feedback via large language models. *arXiv preprint arXiv:2403.15157*, 2024c.

Chaoyun Zhang, Shilin He, Liqun Li, Si Qin, Yu Kang, Qingwei Lin, and Dongmei Zhang. Api agents vs. gui agents: Divergence and convergence. *arXiv preprint arXiv:2503.11069*, 2025.

Duzhen Zhang, Yahan Yu, Jiahua Dong, Chenxing Li, Dan Su, Chenhui Chu, and Dong Yu. Mm-llms: Recent advances in multimodal large language models. *arXiv preprint arXiv:2401.13601*, 2024d.

Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501*, 2024e.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Jiani Zheng, Lu Wang, Fangkai Yang, Chaoyun Zhang, Lingrui Mei, Wenjie Yin, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. Vem: Environment-free exploration for training gui agent with value environment model. *arXiv preprint arXiv:2502.18906*, 2025.