Enhancing Multimodal GUI AI-Agent Benchmarking

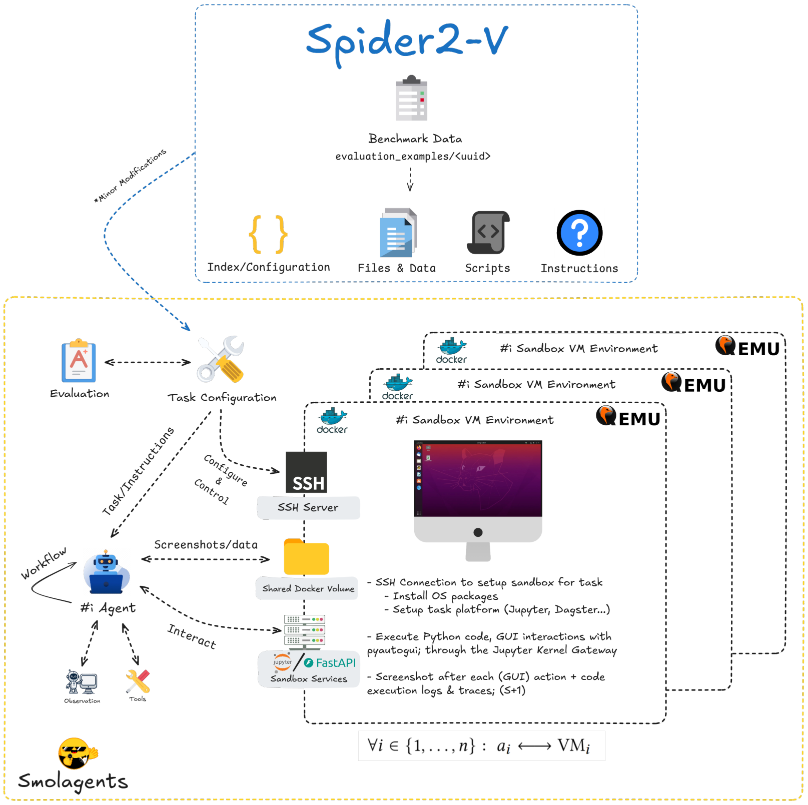
A Modular, Structured and Code Centric Framework for Spider2-V

submitted in partial fulfillment for the degree of master of science

Stijn Hering
12377422

Master Information Studies
data science
Faculty of Science
University of Amsterdam

Submitted on 27.06.2025



- SSH Connection to setup sandbox for task
  - Install OS packages
  - Setup task platform (Jupyter, Dagster...)
- Execute Python code, GUI interactions with pyautogui; through the Jupyter Kernel Gateway
- Screenshot after each (GUI) action + code execution logs & traces; (S+1)

	UvA Supervisor
Title, Name	Prof. Paul Groth
Affiliation	INDELab
Email	p.t.groth@uva.nl

Abstract

Recent advances in AI agents have created a need for robust benchmarks to evaluate their performance on complex, real-world workflows. However, existing multi-modal frameworks like Spider2-V often suffer from monolithic designs, poor developer experience (Dev-X), and limited observability, hindering reproducible research. This thesis introduces a modular, code-centric framework built on open-source standards (Docker, QEMU, SSH) to address these limitations. The proposed framework enhances a baseline agent's capabilities by providing a flexible action space that combines direct Python code execution with GUI automation.

A comparative evaluation on 44 Jupyter-based tasks from the Spider2-V benchmark was conducted. The results demonstrate a statistically significant improvement, with the proposed framework achieving a 39% success rate compared to the baseline's 6%. A quantitative analysis reveals the new framework is also significantly more maintainable, with a 66% reduction in core Python code. Qualitative analysis of agent execution traces shows that success is driven not by avoiding GUI automation, but by the agent's ability to dynamically switch between action types based on a rich, immediate feedback loop. The framework's modularity and enhanced observability demonstrably improve both agent performance and developer productivity.

Keywords

AI Agent Benchmarking, Developer Experience (Dev-X), Multimodal AI Agents, Code Generation, GUI Automation, Reproducible Research

Github Repository

https://github.com/melchiorhering/Automating-DS-DE-Workflows

1 Introduction

Recent advances in Artificial Intelligence (AI)—particularly through large language models (LLMs) and vision-language models (VLMs)—have accelerated the development of *AI agents*: autonomous systems capable of reasoning, navigating graphical interfaces, generating code, and orchestrating complex workflows in real time [14, 28, 30]. These agents are no longer limited to producing static outputs but operate as dynamic entities that interact with executable environments to complete multi-step tasks across various domains [22].

Existing AI-agent architectures have demonstrated *autonomous task execution, graphical user interface (GUI) navigation, code generation, and real-time decision-making*. For instance, SWE-Agent [30] bridges human-computer interaction and automated software engineering, while OS-Copilot [28] manages file operations and GUI applications in diverse settings. Despite these successes, many systems still lack *modularity, robust tracking mechanisms, and adaptability* [23].

As a result, recent efforts have focused on evaluating agents in more grounded, interactive settings. Multiple benchmarking frameworks have emerged to assess AI-agent performance in realistic tasks [7, 10, 12, 13, 29, 32]. However, achieving standardization and robust error resilience across these evaluations remains an open challenge.

Within this broader benchmarking landscape, a particularly promising application area lies in the automation of professional workflows in Data Science (DS) and Data Engineering (DE). These workflows involve complex multi-step tasks such as data ingestion, transformation, orchestration, and visualization—often carried out through graphical and command-line interfaces provided by tools like Jupyter Notebooks, BigQuery, dbt, Airbyte, Metabase, and Dagster [7].

Problem Statement. Despite recent progress, current AI agents struggle to automate such workflows in a consistent and robust way. Most systems lack the modularity, observability, and architecture required to generalize across tasks and tools. Benchmarks such as SWE-Bench [10] and WebArena [32] focus on narrow domains (e.g., software engineering or web navigation), limiting their applicability to DS/DE settings. Furthermore, many agent frameworks operate in loosely specified environments with limited standardization and weak guarantees on reproducibility.

Spider2-V: A Realistic Benchmark. To address these gaps, the *Spider2-V* benchmark was introduced [7]. It is the first multimodal benchmark targeting the end-to-end automation of DS/DE workflows. It features a real-time executable computer environment adapted from OSWorld [29], integrating GUI and CLI-based tasks across 20 professional tools and encompassing **494 realistic** scenarios. Each task is grounded in tutorial-derived instructions and evaluated using custom environment-based metrics. Figure 1 and Figure 2 illustrate the diversity of tools and empirical distributions across task complexity dimensions. A statistical overview of task types, interface modalities, and difficulty levels is presented in Table 1, highlighting the operational diversity and non-trivial nature of the benchmark suite.

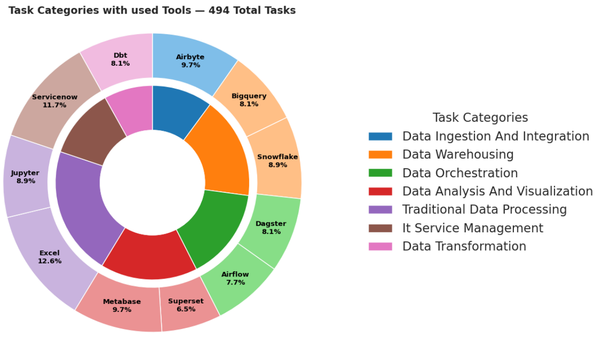


Figure 1: Task categories across tools in the Spider2-V benchmark

Nonetheless, as of the current date, adoption of Spider2-V has been limited. Since its release, only one new agent entry has been recorded [24]. This contrasts sharply with its text-only (text-to-SQL) predecessor, Spider2.0 [13], which has garnered over ten submissions to date. This disparity may be attributed to Spider2-V's

Table 1: Summary of task types and complexity in *Spider2-V*.

Statistic	Value
Total tasks	494 (100%)
Pure CLI	28 (5.7%)
Pure GUI	184 (37.2%)
CLI + GUI	282 (57.1%)
With authentic user account	170 (34.4%)
Without authentic user account	324 (65.6%)
Task complexity (steps)	
Easy (≤ 5 steps)	98 (19.8%)
Medium (6–15 steps)	310 (62.8%)
Hard (> 15 steps)	86 (17.4%)
Median action steps (P25 / P50 / P75)	7 / 9 / 13
Mean abstract instruction length	37 tokens
Mean verbose instruction length	246 tokens
Mean applications used per task	2

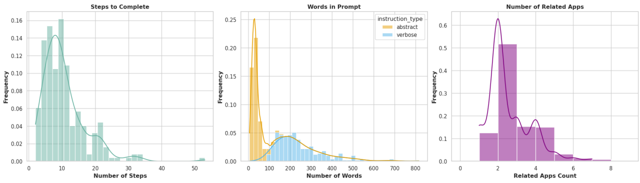


Figure 2: Empirical distributions of action steps, instruction length, and applications per task

more intricate setup, its GUI-intensive nature, and a current lack of streamlined tooling for experimentation and reproducibility.

Research Focus and Contributions. This research proposes a *lean, modular agent framework based on the AI agent framework Smolagents [20]*, grounded in standard OS-level protocols (e.g., SSH, QEMU, Docker), open-source software, and *enabling direct execution of Python code with immediate feedback* using Smolagents-*CodeAgent* abstraction. It can optionally be extended with tools from existing popular AI agent ecosystems such as *LangChain*, *LlamaIndex*, *HuggingFace*, and others—to serve as a usable, reproducible, and performant alternative for the current Spider2-V benchmarking framework. The primary contributions are:

(1) **Modular Agent Stack (Developer Experience and Developer Productivity).** A lightweight and pluggable agent framework was designed with the explicit goal of enhancing developer experience (Dev-X) and productivity (Dev-P), as defined by Razzaq et al. [19]. The architecture is built on declarative workflow graphs and swappable modules, which results in minimal, composable, and tool-agnostic interfaces. This modularity, grounded in open standards and established open-source tools (e.g., SSH, Docker, QEMU), facilitates rapid prototyping and ensures reproducible evaluations.

(2) **Direct Code Execution with Immediate Feedback.** Safe and efficient execution of agent-generated Python code directly inside containerized VMs, providing the AI agent (and developers) with immediate feedback on code actions.

(3) **Observability and Telemetry.** Built-in OpenTelemetry tracing and structured logging infrastructure capture fine-grained runtime data, enabling longitudinal analysis of agent behavior and benchmarking conditions.

(4) **Quantitative Benchmarking.** Task-level evaluations of success rate, execution time, resource usage, token-count, and monetary cost (e.g., AI inference API calls).

Research Questions. To assess the framework's effectiveness, the study is structured around the following research question and sub-questions:

RQ: *To what extent can a modular, code-centric agent framework, built on open-source standards, simultaneously enhance an AI agent's task performance and improve the usability and reproducibility of the* ***Spider2-V*** *benchmark?*

SRQ1: *What specific limitations in the existing* ***Spider2-V*** *pipeline—related to developer experience, modularity, and agent observability—can be addressed by a framework grounded in open standards and direct code execution?*

SRQ2: *How does enabling a flexible action space, where an agent can choose between direct code execution and GUI automation (using* `pyautogui`*), impact task performance metrics like success rate and efficiency, when compared to the restricted, GUI-primitive-only approach of the* ***Spider-2-V*** *baseline framework?*

SRQ3: *How does the modular design and enhanced observability of the proposed framework accelerate the development and evaluation cycle for new agent strategies on the Spider2-V benchmark?*

2 Related Work

Recent years have seen increasing efforts to benchmark AI agents on complex, real-world tasks. Despite promising progress, existing benchmarks remain limited in scope, reproducibility, and evaluation rigor. State-of-the-art (SOTA) benchmarks like *WebArena* [32], *SWE-Bench* [10], and *OSWorld* [29] each capture narrow aspects of agent performance—such as web navigation, software debugging, or OS interaction—but none encompass the full breadth of multimodal, GUI-intensive workflows typical of professional data science and engineering.

Spider2-V [7] aims to fill this gap by combining a suite of code- and GUI-based DS/DE benchmark tasks with its own evaluation scripts and a custom AI-agent framework. However, this built-in framework suffers from poor extensibility, limited flexibility, and a lack of modern tooling. These shortcomings—coupled with weak error tracking and inconsistent developer ergonomics—undermine its usability. As of this writing, the only new result submitted since the benchmark's release reports a task success rate of just 16.6% [24]. This limited uptake may reflect not only the inherent difficulty of the tasks, but also practical barriers to adoption—namely the benchmark's rigid agent framework, fragmented tooling, and limited developer support.

This section provides a thematic overview of prior work grouped into three main areas: (1) AI-agent benchmarks and modularity,

(2) developer experience and benchmarking efficiency and (3) task orchestration frameworks. All linked to the the Research Question (**RQ**) and Sub-Research Questions (**SRQ1–SRQ3**).

2.1 Benchmarks and Modularity for AI Agents

Early work on single-agent systems (e.g., *ReAct*, *RAISE*, *AutoGPT+P*, and *LATS*) demonstrated the importance of *reasoning, planning, and effective tool-calling* for autonomous task execution [16]. More recent AI-agent architectures, including both single-agent and multi-agent paradigms, have exhibited increasing capabilities in *long-horizon reasoning, structured decision-making, and adaptive tool integration*. However, as highlighted in[16], the effectiveness of an AI-agent system largely depends on its architectural choices, with the most successful implementations incorporating *well-defined system prompts, structured task delegation, iterative reasoning-execution-evaluation loops, dynamic team structures, and intelligent feedback mechanisms*. Architectures leveraging these design principles tend to perform more robustly across diverse benchmarks and problem domains [16].

2.1.1 Limitations in Existing Benchmarks. Despite recent advancements, AI-agent benchmarking faces persistent challenges in *standardization, reproducibility, and scalability*. As noted in [11], the lack of standardized evaluation protocols leads to *irreproducible results*, making it difficult to assess genuine improvements in agent performance. Without clear benchmark guidelines, developers often modify evaluation scripts, leading to inconsistent results across implementations.

Furthermore, repurposing *LLM benchmarks* for agent evaluations introduces inconsistencies, as these benchmarks were not designed to handle the dynamic interactions required by AI agents. High computational costs further complicate evaluation, as benchmarking an agent often requires thousands of API calls, making repeated evaluations impractical [11]. Additionally, dynamic environments, such as GUI-based interfaces and web interactions, introduce unpredictable external factors that affect benchmarking accuracy. Rate limits, UI changes, and server inconsistencies can significantly alter evaluation outcomes, making it difficult to maintain reliability across different test runs.

Finally, the absence of *error tracking mechanisms* leads to subtle evaluation flaws, including incorrect success rate reporting and failure to detect agent errors. Addressing these challenges requires a well-defined, *modular benchmarking framework* that improves *reproducibility, cost-efficiency, and error transparency*.

2.1.2 Limitations Specific to Spider2-V. Spider2-V extends OSWorld-style AI agent evaluation by introducing code- and GUI-based workflows for data engineering. It spans multiple tools and modalities across 494 real-world tasks in a full OS environment. However, its evaluation protocols lack standardization, and its agent interface restricts flexibility due to limited error tracking and constrained action spaces [11, 29].

Another key limitation is that *Spider2-V* currently imposes substantial guardrails on agent behavior, restricting agents to predefined and inflexible action spaces. The framework explicitly forbids direct and arbitrary Python execution, instead confining the agent to specific configurations like `COMPUTER13` or `PYAUTOGUI`. The `COMPUTER13` action space, for example, is limited to a predefined set of string-encoded primitives, such as structured commands for moving the cursor or typing text, rather than allowing executable code. Even when an action space appears to involve code, as with the `PYAUTOGUI` setup, it is a heavily sandboxed environment that only permits calls to the `pyautogui` [3] library, preventing any other type of programming logic or tool use.

This rigid design stands in stark contrast to a growing body of research demonstrating the power of fully executable action spaces. Studies by Wang et al. [26] and Nguyen et al. [17] show that permitting agents to dynamically generate and execute arbitrary code markedly enhances their flexibility and performance in open-ended tasks. By constraining agents to fixed primitives, *Spider2-V*'s environment can stifle agent performance and, critically, eliminates the opportunity for an agent to autonomously recover from errors by generating and debugging its own code based on runtime feedback. This inflexibility increases the overhead of integrating novel methods and underscores the need for a more modular and adaptable framework that embraces modern, code-based agent strategies.

Several lines of research offer insights for overcoming these challenges. *Containerized benchmarks* [10, 31] achieve reproducibility in code-oriented tasks but often lack full GUI support. *VM-based benchmarks* such as OSWorld [29] allow comprehensive OS-level evaluations yet suffer from high overhead, limiting parallelization. *Browser-based approaches* like WebArena, VisualWebArena, and WorkArena [6, 9, 12, 32] excel at parallel execution via Ray [2] but do not provide a full OS GUI environment.

Why Spider2-V Needs a Modular AI Agent Framework. By merging the strengths of OS-level, container-based, and browser-based benchmarks, *Spider2-V* can achieve both extensibility and maintainability. Existing work suggests that a *modular architecture* can offer several key advantages, including (1) employing and tuning different problem-solving strategies across sub-agents, (2) enabling sub-agents to gather information from distributed sources, and (3) reducing unnecessarily long trajectories that increase costs and add extraneous context [5]. These attributes are crucial for *Spider2-V* given its reliance on complex GUIs, code-intensive tasks, and multi-modal data.

In the context of *Spider2-V*, adopting a modular[1] code-centric AI agent framework directly addresses:

- **Dev-X & Dev-P Factors**: A more flexible design simplifies adding new agent methods.
- **Challenges & Limitations**: Containerization can reduce resource and complexity overhead and enhance reproducibility.
- **Performance**: A robust modular code-centric system can streamline tool usage, flexibility and execution paths.
- **Accelerated Experimentation**: Increased adaptability lowers the time and effort required to evaluate novel algorithms or workflows.

2.1.3 Developer-Centered Evaluation of Benchmarks. Beyond technical concerns like reproducibility, execution time and performance,

[1] In this context, *"modular"* refers to the agent system's ability to decompose into reusable components—e.g., execution engines, perception models, and tool plug-ins—that can be independently swapped or extended.

the usability of benchmarking frameworks from a developer's perspective has emerged as a vital success factor. According to a systematic literature review by Razzaq et al. [19], Developer Experience (Dev-X) directly influences Developer Productivity (Dev-P), particularly in multi-stage workflows involving tool orchestration, infrastructure, and iteration. Key findings emphasize the importance of minimizing cognitive load, standardizing workflows, preserving developer flow, and supporting modular mental models through transparency and observability.

The current *Spider2-V* implementation fails to meet these criteria. Its AI agent framework is tightly coupled to the benchmarking infrastructure: agent logic, environment setup, and evaluation procedures are co-located in a way that prevents isolated development or testing. Although *Spider2-V* resets the virtual environment before each evaluation, it supports only one runtime environment at a time and does not permit running agents independently from the sandbox or orchestrating across multiple containerized instances. This inflexibility complicates development workflows and inhibits scaling or experimentation.

Moreover, tools and agent workflow steps in *Spider2-V* are not pluggable or declarative. They are hardwired into the framework's structure, forcing developers to modify internal files even for simple extensions or tool injections. Telemetry, if present, is minimal and benchmark-specific. Developers cannot readily plug into external monitoring systems, and there is no unified interface for debugging or tracking execution failures across tasks.

These limitations map closely to the Dev-X antipatterns identified by Razzaq et al., including work fragmentation, high code complexity, poor flow management, and limited situational awareness. They discourage contribution and experimentation, and they make it unnecessarily difficult to adopt *Spider2-V* in new research or production-grade evaluation workflows.

The new modular framework developed in this research explicitly addresses these problems. The AI agent is decoupled from the runtime environment and communicates over standardized interfaces (e.g., SSH, QEMU, Docker). Agent steps and tools are registered declaratively and only integrated when needed, allowing microservice-like composition and reusability. The framework is compatible with OpenTelemetry [1] and can be paired with any open-source telemetry dashboard or service. As a result, developers can monitor performance, trace errors, and iterate quickly without being locked into a brittle evaluation stack.

Table 2 provides a comparative summary of Dev-X design factors across *Spider2-V* and the new framework.

2.2 AI-Agent Frameworks and Task Orchestration

This subsection discusses key frameworks for modular agent orchestration and automation pipelines, with direct relevance to **SRQ1** through **SRQ3**. Contemporary AI-agent frameworks employ modular pipelines to enable *automation, adaptability, and decision-making*. Established platforms like *LangChain* and *CrewAI* provide *structured feedback loops, error recovery, and multi-agent coordination* [8, 25], but can be relatively heavyweight. In contrast, emerging solutions such as *Pydantic-AI* [21] and *Smolagents* [20] emphasize simplicity and minimal overhead, offering streamlined ways to create or adapt tools within the agent framework.

As previously stated, recent studies [17, 26] highlight the advantages of having agents generate and execute code in situ. The *Smolagents* framework, for instance, implements this principle through its *CodeAgent*, which is designed to dynamically produce Python code, execute it, and directly revise actions based on observed outcomes. This approach not only aligns with *Spider2-V*'s DS & DE tasks but also integrates well with open-source AI model repositories like Hugging Face [4] or AI inference providers, further simplifying the experimental workflow for researchers.

Furthermore, a key advantage of direct code execution is the ability for an agent to perform autonomous error recovery. When generated code fails, the resulting output (e.g., a stack trace or an error message) provides structured, actionable feedback. This allows the agent to debug its own code and iteratively refine its approach, a capability shown to significantly improve robustness in complex problem-solving scenarios.

Smolagents as a Focal Agent Framework. Given the updated vision for *Spider2-V*, a lightweight and flexible agent framework such as *Smolagents* becomes particularly compelling. Its core design principles—*simplicity, code-driven action spaces, and modular tooling*—address key benchmarking needs:

- **Developer Experience (Dev-X & Dev-P):** Its minimal abstractions and code-first design reduce cognitive overhead and accelerate the development and testing cycle for new agent strategies.
- **Flexible Task and Tool Integration:** The code-centric architecture provides a unified approach for setting up experiments across diverse tasks (e.g., GUI automation and direct code execution), addressing the limitations of rigid, single-purpose frameworks.
- **Performance and Extensibility:** The ability to dynamically generate and execute code improves task efficiency and performance. The framework's modularity supports future extensibility by facilitating not only the rapid integration of new components, like alternative vision models (e.g. specialised GUI models), but also allowing the agent's core reasoning workflow to be easily adjusted, updated, or expanded.

While agent frameworks like *LangChain* and *CrewAI* may offer comprehensive structures for role-based delegation or multi-agent coordination, *Smolagents* provides a straightforward, minimalistic entry point for researchers and developers looking to extend *Spider2-V* without incurring excessive architectural overhead. Moreover, the design of a modular agent framework ensures that specific tools or integrations from larger agent frameworks can still be incorporated if needed. This flexibility allows researchers to selectively draw upon the strengths of existing AI-agent ecosystems (e.g., advanced text parsers, multi-agent orchestration modules) while preserving the lightweight and code-centric focus that characterizes Smolagents.

Dev-X Factor	Spider2-V	Smolagents Framework
Simplicity & Minimalism	High barrier to entry; setup requires unpacking entangled agent/evaluation logic.	Minimal abstractions and *CodeAgents* (≈1000 kLoC) enable rapid onboarding and reduce boilerplate.
Standardization & Consistency	Interfaces and scripts vary across tasks, hindering reproducibility.	Unified design based on common standards (SSH, Docker, QEMU, Model Context Protocol (MCP)); declarative workflow graphs enforce API consistency.
Access to Resources	Debugging in VM snapshots is ad-hoc and manual.	Containerized design supports parallel agents; telemetry can be visualized in any OpenTelemetry-compatible UI.
Mental Model Support	Limited visibility into agent reasoning and step outcomes.	Modular callbacks, rich logging, and traceable workflows make agent behavior transparent and inspectable.
Developer Flow Management	Frequent context switching due to manual resets and interdependencies.	Agents, tools, and sandbox environments are decoupled and hot-swappable; developers can iterate without touching unrelated components.
Managing Code Complexity	Agents confined to fixed primitives (e.g., `CLICK(x,y)`, sole *pyautogui* code), leading to brittle glue code.	CodeAgents generate and execute Python dynamically, reducing indirection and increasing expressivity.
Situational Awareness	No integrated metrics; results exposed post-hoc via static reports.	Real-time metrics exported via OpenTelemetry with step-level observability.
Technology Upgradability	Hardwired interfaces and custom abstractions impede tool integration.	Plugin-based architecture supports new tools and models as self-contained modules in the agent workflow (e.g., GUI-models, AI agent ecosystem tools & modules, MCP - services).

Table 2: Comparison of Dev-X factors between the original *Spider2-V* and the new modular AI-agent framework, adapted from [19].

3 Methodology & Experimental Setup

3.1 Research Design and Methodology

This research employs a comparative evaluation to assess the effectiveness of a newly developed modular agent framework against the original *Spider2-V* benchmark setup. The primary goal is to address the limitations in usability, developer experience (Dev-X), and agent performance identified in the existing framework. The new framework, built from scratch to be fully decoupled from the original implementation, replaces the VMware-based setup with QEMU and Kernel-based Virtual Machines (KVM), which are containerized and orchestrated via Docker. This new architecture aims to improve performance, reproducibility, and transparency.

Due to time and resource constraints, the scope of this evaluation is focused on the **44** tasks within the *Spider2-V* benchmark that specifically involve Jupyter Notebooks. This total of **44** tasks is composed of task pairs, where each pair shares an identical evaluation script and expected outcome but is presented to the agent with either a verbose or a less verbose natural language prompt. This subset was chosen because it represents a significant and complex portion of data-centric workflows. Jupyter is not only central to data science for tasks like exploration and modeling, but also serves as an interface in data engineering. The selected tasks reflect this duality, encompassing activities such as creating plots, managing the notebook environment (e.g., starting servers or deleting kernels), refactoring code from scripts into notebooks, and performing data engineering workflows like connecting to databases to extract, transform, and load data. This makes the subset a highly representative toolset for comparing agent performance and framework usability within the benchmark's broader DS/DE focus.

The core of the methodology is a direct comparison between two distinct configurations of the defined Jupyter tasks:

(1) **SpiderV-2 Baseline:** The official *Spider2-V* framework, which uses Jupyter Lab as its code (notebook) editor. In this configuration, the agent operates in a GUI-only modality, making decisions based solely on raw screenshots of the environment and executing actions via `pyautogui` primitives. Some adaptations were made to its source code to ensure compatibility with contemporary AI model inference providers.

(2) **Proposed Modular Framework:** The new, custom-built framework based on Smolagents. This configuration uses Visual Studio Code (VS Code) as the notebook editor. To ensure a direct and fair comparison, the agent's visual processing is identical to the baseline, relying only on the same raw screenshots for its GUI understanding. The only architectural difference is the introduction of a bimodal action space: alongside the standard GUI automation, the agent is equipped with a **direct code execution channel** (via a Jupyter Kernel Gateway). The agent's execution is driven by two components: the official, unmodified system prompt from the Smolagents source code, and a task-specific input prompt, detailed in Appendix A.

By comparing the results from these two setups on the same set of tasks, this study aims to provide quantitative and qualitative evidence to answer the research questions.

3.2 The Modular Framework: Architecture and Components

The proposed framework was built from the ground up to prioritize modularity, reproducibility, and a superior developer experience. Its architecture, illustrated in Figure 4, is founded on a clear separation between the agent's logic (host) and the task execution environment (guest).

Infrastructure, Sandboxing, and Communication. Each agent task runs within a completely isolated and safe sandbox environment, which is achieved by running a QEMU virtual machine inside a Docker container. To facilitate communication between the host system and the guest VM, specific network ports within the container are exposed. These ports connect to services that are pre-configured in the base OS image of the virtual machine, creating four primary channels built on open standards:

- **Direct Code Execution (via Jupyter Kernel Gateway):** A Jupyter Kernel Gateway exposes a WebSocket connection, allowing the agent to execute code directly within the sandbox. This provides an immediate feedback loop with rich output (e.g., print statements, data visualizations, and error traces) that is fundamental for agent performance and autonomous error recovery.
- **Visual Observation (via FastAPI Server):** A custom, lightweight FastAPI server provides API endpoints that allow the agent to programmatically capture the state of the GUI. Typically, a callback function is invoked after each agent action to request a screenshot, allowing the agent to *'see'* the result of its action, or to trigger a video recording for subsequent debugging.
- **Interactive GUI Access (via noVNC):** A noVNC server streams the VM's graphical desktop directly to any standard web browser. This channel is crucial for real-time, interactive debugging, allowing a human developer to observe the agent's actions as they happen or to manually inspect the environment's state without needing a separate VNC client.
- **Control and File Transfer (via SSH):** A standard SSH connection is used throughout the task lifecycle for initial setup, transferring task files, runtime control, and, crucially, for remotely executing evaluation scripts and retrieving the final results.

This containerized approach ensures each agent has its own dedicated sandbox, making the system scalable and allowing multiple agents to run in parallel without interference. Formally, each agent a_i is bound to a unique VM instance, ensuring strict isolation:

$$\forall i \in \{1, \ldots, n\} : \; a_i \longleftrightarrow \mathrm{VM}_i$$

Agent Orchestration with Smolagents. The AI agent itself is built using the *Smolagents* framework, chosen for its lightweight, code-first, and transparent design. While the framework's core modules for model connections, memory, and telemetry are used directly, several minor abstractions and improvements were implemented to better handle the multimodal feedback loop required for this benchmark. The resulting agent workflow (Figure 3) thus leverages many of Smolagents' built-in features, with the exception of the multi-step **Planning** module, which was not enabled for this study.

Built-in Framework Functionality. To improve developer insight and enable in-depth analysis, the framework includes several built-in capabilities: runtime introspection of the VM, per-task configuration overrides, automatic screenshot capture with a cursor overlay to show focus, logging of all executed code snippets, and a mechanism to replay failed task executions for easier debugging.

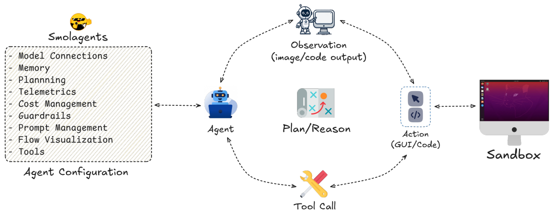


Figure 3: Overview of the Smolagents based framework workflow loop

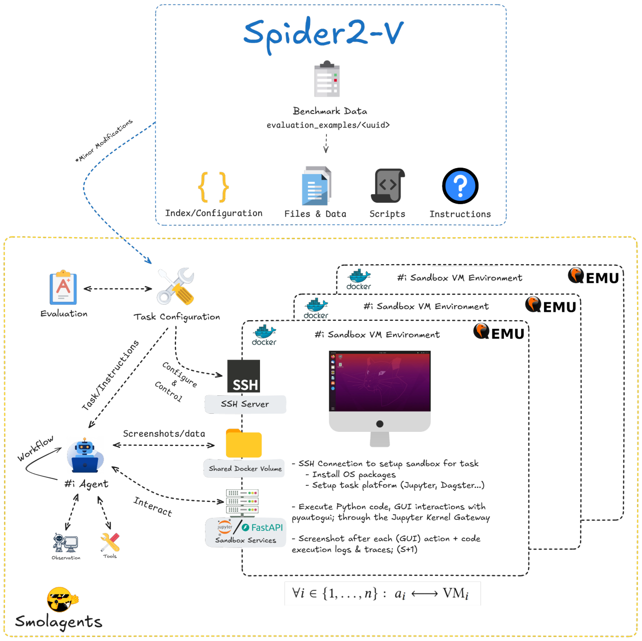

Spider2-V
Benchmark Data
evaluation_examples/<uuid>
Index/Configuration
Files & Data
Scripts
Instructions
*Minor Modifications
Evaluation
Task Configuration
Task/Instructions
Configure & Control
SSH
SSH Server
Workflow
#i Agent
Screenshots/data
Shared Docker Volume
Interact
jupyter
FastAPI
Sandbox Services
Observation
Tools
docker
#i Sandbox VM Environment
EMU
- SSH Connection to setup sandbox for task
- Install OS packages
- Setup task platform (Jupyter, Dagster...)
- Execute Python code, GUI interactions with pyautogui; through the Jupyter Kernel Gateway
- Screenshot after each (GUI) action + code execution logs & traces; (S+1)
∀i ∈ {1, . . . , n} : a_i ⟷ VM_i
Smolagents

Figure 4: High-level architecture. Each agent interacts with its own isolated VM via a dedicated Jupyter Kernel and observation server.

3.3 Experimental Infrastructure and Configuration

Compute Hardware. All experiments were executed on a single local workstation to facilitate rapid iteration and low-latency debugging. While the fully containerized stack is designed to be portable to larger compute environments for future work, a single high-performance machine is sufficient for this evaluation. The specifications of the workstation used are as follows:

Operating System Windows 11 Pro with WSL 2 (Ubuntu 24.04.2)
CPU AMD Ryzen 9 7950X3D
Memory 32 GB DDR5
GPU NVIDIA RTX 4080 SUPER

Model and Inference Configuration. The framework uses the *Smolagents*'s unified routing layer to abstract model backends, allowing for flexibility in choosing inference providers. However, for this evaluation, the model choice was constrained by hardware limitations for local hosting and limited access to alternative commercial APIs. To ensure the comparison focused on the framework's architecture rather than the model's capabilities, a single model—OpenAI's gpt-4o-mini-2024-04-16[18]—was exclusively used for all language reasoning, code generation, and visual analysis tasks in both the baseline *Spider2-V* and the proposed framework setups. This model was selected as it represents a strong balance of capability and efficiency within these constraints. According to OpenAI, it is a cost-effective, multimodal model optimized for fast reasoning that performs efficiently on both coding and visual tasks, making it a suitable single model for the diverse requirements of this experiment.

3.4 Evaluation Setup and Metrics

The evaluation directly compares the baseline and proposed frameworks on the **44** Jupyter tasks. The original evaluation-examples from *Spider2-V* were adapted for the new framework with additional metadata and configuration hooks to support modular execution.

To ensure a fair and controlled comparison, two key experimental parameters were standardized across both configurations. First, to maintain consistency with the baseline methodology, the maximum number of agent steps per task was set to **15**, a limit established by the original **Spider2-V** framework. If this limit was reached, the interaction was terminated, and the task was evaluated based on the environment's final state. This evaluation method correctly credits tasks that were accomplished before the step limit was reached, even if the agent did not explicitly terminate. Second, to account for potential non-determinism, each of the **44** tasks was executed three times for each setup. The final performance metrics are reported as the average over these runs.

The comparison between the two frameworks is performed across two primary dimensions: quantitative task performance and qualitative developer experience.

Quantitative Performance Comparison. Data for quantitative analysis is collected from both frameworks, though the richness of the data differs significantly.

- The baseline *Spider2-V* setup produces a minimal output: a text file containing the final binary score (1.0 or 0.0) and a .jsonl file logging the agent's action steps.
- The proposed modular framework generates a comprehensive JSON log for each task run. This log contains the full message history, the final score, any evaluation errors, the agent's terminal output, the reason for task termination (e.g., success, max_steps_error), the total tokens consumed, and the total execution time.

From this data, the primary metric used for direct comparison between the two frameworks is the **Task Success Rate** . Additionally, for the proposed modular framework, the number of **Reasoning and Tool Steps** is analyzed to provide deeper insight into the agent's efficiency and problem-solving path.

Beyond these primary metrics, a behavioral analysis is also conducted on the actions taken by the CodeAgent within the proposed modular framework. The execution logs from the task runs are analyzed to distinguish between two types of actions: (1) **Direct Code Execution**, where the agent uses standard Python libraries via the Jupyter kernel, and (2) **GUI Automation**, where the agent resorts to using pyautogui to simulate mouse and keyboard interactions. This analysis provides deeper insight into the agent's problem-solving strategies and its ability to choose the most efficient interaction modality.

Qualitative Experience Comparison (Dev-X & Dev-P). In addition to quantitative metrics, a qualitative comparison of the Developer Experience (Dev-X) and Developer Productivity (Dev-P) is conducted. This assessment is based on the experience of implementing, running, and debugging the experiments within both frameworks. Key factors include the ease of tracing agent behavior, the effort required to analyze failures, and the overall friction of the development cycle. The rich, structured logging from the proposed framework is a key feature that directly facilitates this analysis.

Framework Isolation Assumption. Because the original *Spider2-V* agent logic is not reused, exact replication of certain implicit behaviors may diverge. This potential divergence is mitigated by using an adapted version of the official *Spider2-V* evaluation scripts & functions, which ensures that outcome comparability is preserved at the task level.

4 Results

This section presents the results of the experimental analysis, comparing the proposed modular code-centric framework (Setup 2: CodeAgent) against the baseline (Setup 1: Spider2-V). The findings directly address the research questions concerning task performance and agent behavior.

4.1 Overall Task Performance

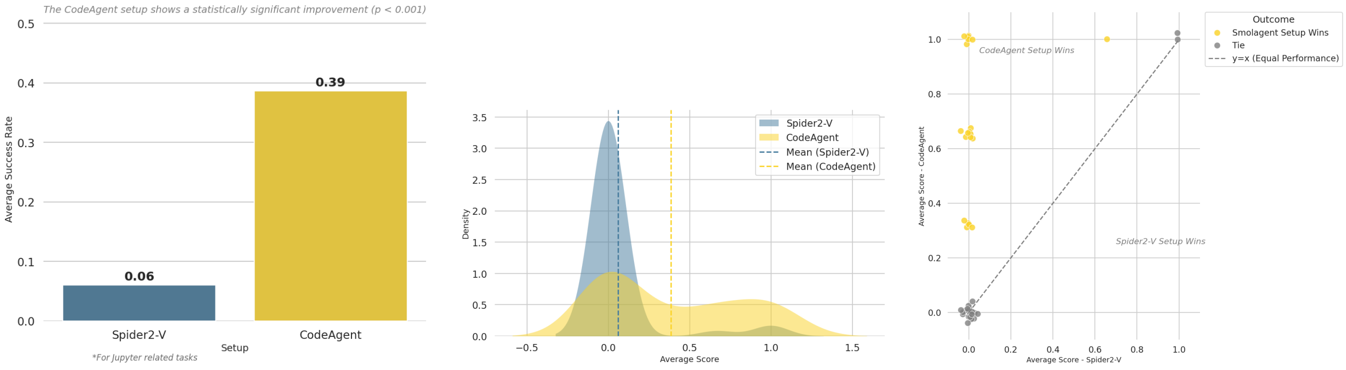
To address SRQ2, an initial analysis establishes a quantitative performance baseline between the two setups. The proposed framework, utilizing the CodeAgent, demonstrates an average success rate of **39%**, a substantial improvement over the **6%** from the baseline (Setup 1). As detailed in Table 3 and illustrated in Figure 5, this performance gap is confirmed to be statistically significant (Wilcoxon signed-rank test, $p < 0.001$).

The distribution of scores (Figure 5b) further clarifies this result. The baseline's scores are heavily concentrated at 0.0, whereas the proposed framework exhibits a bimodal distribution with a notable peak at 1.0, indicating a clear increase in solved tasks. The head-to-head comparison in Figure 5c reinforces this finding, illustrating that for every task where performance differed, the proposed framework outperformed the baseline.

4.2 Analysis of Agent Behavior

To investigate the drivers behind this performance gap, a key hypothesis was that the choice between direct code execution and fragile GUI automation would be a primary predictor of success. The visualizations in Figure 6 provide insight into this relationship.

Contrary to the initial hypothesis, the analysis reveals a more nuanced picture. The distribution of PyAutoGUI code-line ratios (Figure 6a) is remarkably similar for both successful and unsuccessful tasks, with median ratios of **0.148** and **0.143** respectively. The logistic regression in Figure 6b shows a flat, slightly negative correlation, confirming that the ratio of GUI-to-code lines is not a strong predictor of the final outcome. These results suggest that



(a) Overall average success rate.

(b) Distribution of scores per task.

(c) Head-to-head task outcomes.

Figure 5: Summary of task performance comparison between the baseline (Setup 1) and the proposed CodeAgent framework (Setup 2).

Table 3: Performance Comparison on Jupyter Tasks.

Metric	**Spider2-V**	**CodeAgent**
Observation Space	`screenshots`	`screenshots`, `logs` & `code-output`
Action Space	`pyautogui`	`pyautogui` & `direct-code-execution`
Average Success Rate	0.06	**0.39**
Standard Deviation	0.23	0.42
Runs per Task	3	3
Total Tasks (N)	44	
Wilcoxon p-value	$p < 0.001$	

for this benchmark, the agent's overall proficiency, rather than its choice of tool, is the primary limiting factor in its performance.

A qualitative analysis of the execution logs further reveals a dynamic problem-solving strategy. The agent regularly switches between GUI automation and direct code execution based on the feedback from each step. This iterative, multimodal reasoning process is exemplified in the traces found in Appendix B.

While the tool selection ratio does not determine the outcome, the agent's ability to self-evaluate its state is another critical factor in its reliability. An investigation into this was conducted by comparing its self-reported final state against the ground-truth score. The confusion matrix in Figure 7 reveals that while the agent correctly identified its state in **74** instances (*30 True Negatives* and *44 True Positives*), its self-evaluation is prone to a high rate of error. The analysis highlights two key error types:

- **False Positives (51 instances):** A significant number of cases where the agent incorrectly reported success on a failed task.
- **False Negatives (7 instances):** A smaller number of cases where the agent terminated due to the step limit (`max_steps_error`) on a task that was already solved.

This tendency towards a high number of False Positives suggests that while the agent is effective at attempting solutions, its internal criteria for verifying success remain an area for improvement.

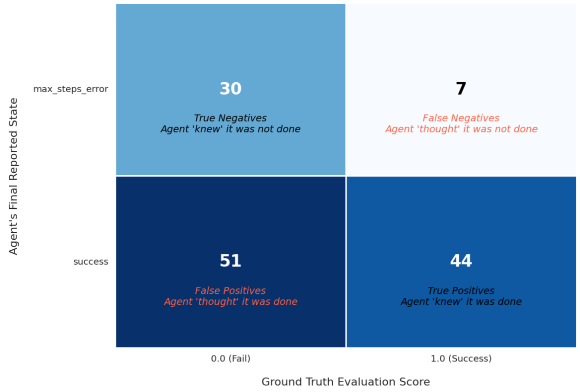
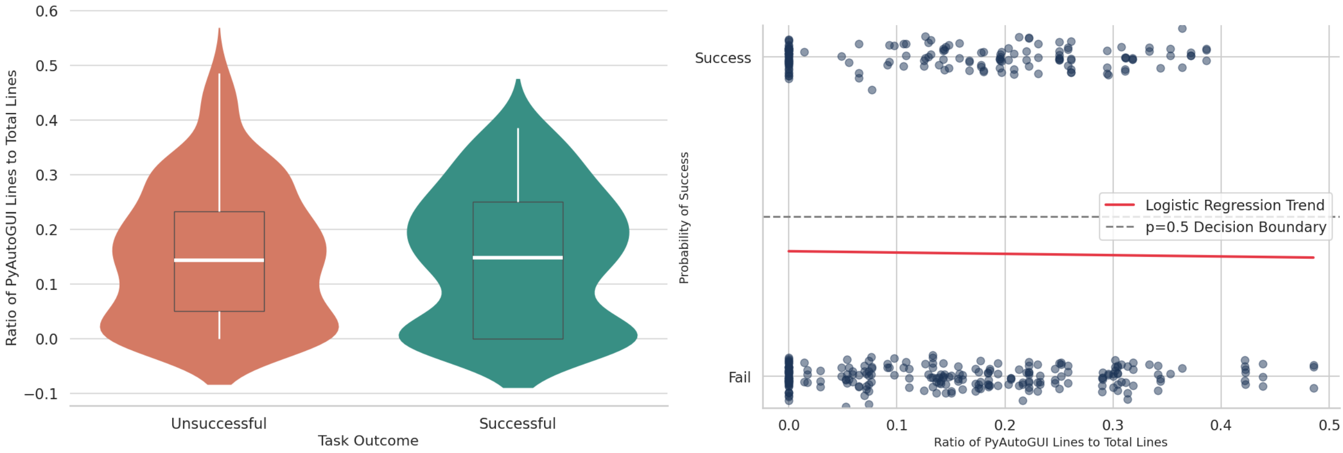


Figure 7: Confusion matrix of reported vs. actual state for the CodeAgent.

4.3 Framework Complexity and Developer Experience

A core objective of this research (SRQ1, SRQ3) was to address limitations in the baseline's developer experience by creating a more modular and maintainable framework. A quantitative analysis of the two codebases, presented in Table 4, reveals the impact of this new design. The proposed framework achieves a 62.1% reduction in total lines of code and, more importantly, a 66.0% reduction in the core Python logic required. This significant decrease in codebase size and complexity directly contributes to improved usability and a more rapid development cycle.

These quantitative improvements are rooted in specific design choices that align with established Developer Experience (Dev-X) themes from the literature. Table 5 (see Appendix for full table) provides a detailed comparative analysis across four key Dev-X themes.



(a) Distribution of PyAutoGUI line ratio by task outcome.

(b) Probability of success as a function of PyAutoGUI usage.

Figure 6: Analysis of agent tool usage and its impact on task success within the CodeAgent framework.

The monolithic architecture of the baseline negatively impacts factors like *Code Complexity* and *Technology Upgradability*. In contrast, the proposed framework's modular, containerized design improves the developer's ability to maintain focus and understand the system's state, positively influencing *Developer Flow* and *Information Needs*. By leveraging open standards and reducing boilerplate, the framework not only enhances agent performance but also demonstrably improves the key developer-centric metrics that are key for reproducible and efficient research.

Table 4: Core Codebase Complexity Comparison.

Metric	Spider2-V	CodeAgent	Reduction
Repository Metrics			
Total Files	87	45	48.3%
Total Lines of Code	14,313	5,423	62.1%
Lines of Code			
Python	13,388	**4,547**	**66.0%**

5 Discussion & Future Work

This section interprets the experimental results in the context of the primary research questions and the broader state-of-the-art, reflects on the study's limitations, and proposes concrete directions for future research.

5.1 Performance in the Context of State-of-the-Art

The results demonstrate a statistically significant performance improvement of the proposed modular framework over the *Spider2-V baseline*. The key architectural shift—providing the agent with a flexible action space that includes direct code execution—appears to be the primary driver of this improvement. This finding aligns with a growing consensus in the field, where recent work on agents with dynamic code generation capabilities has shown similar performance gains over more restricted, primitive-based approaches [17, 26]. These results contribute to this body of evidence, demonstrating that providing agents with a flexible, code-centric action space is an important factor for success in complex multi-modal tasks.

5.2 Agent Strategy and Multimodal Feedback

The source of this significant performance improvement is further revealed through a qualitative analysis of the agent's behavior. The execution traces, exemplified in Appendix B, show a dynamic problem-solving strategy enabled by the framework. The agent does not commit to a single modality but regularly switches between GUI automation and direct code execution within the same task. This behavior is driven by an instant feedback loop; the agent uses the output or error messages from a code execution step to inform its next GUI action, and conversely, uses the visual evidence from a screenshot to formulate its next block of code. This iterative, multimodal reasoning process allows the agent to build a comprehensive understanding of the environment's state and adapt its strategy accordingly, directly addressing SRQ2 by demonstrating how a flexible action space enables more robust and intelligent solution paths.

5.3 Improvements to Developer Experience (Dev-X)

Beyond agent performance, a primary goal of this research was to improve the developer experience (Dev-X) of the benchmarking process itself, addressing SRQ1 and SRQ3. The design of the *CodeAgent* framework was explicitly guided by principles known to enhance Dev-X and, by extension, Developer Productivity (Dev-P), as systematically reviewed by Razzaq et al. [19]. As detailed in the comparative analysis in Appendix C, the proposed framework directly addresses negative Dev-X factors present in the baseline. By replacing a monolithic architecture with a decoupled, containerized system built on open standards (Docker, SSH, OpenTelemetry), the new framework reduces *Code Complexity* and improves *Technology*

Upgradability. This reliance on familiar tools enhances *Developer Fluency* by leveraging the existing skills of researchers. Furthermore, the modular design and enhanced observability features reduce *Work Fragmentation* and improve *Mental Model Support*, allowing developers to iterate and debug more efficiently, thus accelerating the research and development cycle.

5.4 Limitations and Threats to Validity

Several limitations and threats to validity should be considered when interpreting the results of this study. These are categorized into external, construct, and conclusion validity.

- **External Validity (Generalizability):** The findings of this study have two main threats to generalizability. First, the evaluation was scoped to the 44 Jupyter-based tasks from the *Spider2-V* benchmark. While representative, the agent's performance and behavioral patterns may not generalize to the full 494 tasks, especially those heavily reliant on different software tools or pure GUI interaction. Second, the results are intrinsically tied to the specific LLM used in the experiments. Different large language models may exhibit different reasoning capabilities and success rates, potentially altering the outcomes.
- **Construct Validity:** A threat to construct validity lies in the evaluation process itself. The evaluation scripts rely on canonical GUI states and software versions; minor tool-version drift or non-deterministic interface rendering between runs could lead to false negatives. More significantly, the agent's ability to self-evaluate its success is unreliable. The analysis revealed a high number of **False Positives (51 instances)**, where the agent incorrectly reported success. This indicates a flaw in the construct of the agent's termination logic, a challenge that is independent of the framework itself.
- **Conclusion Validity:** Although a deterministic model configuration was used, a degree of stochasticity can emerge from complex agent-environment interactions or subtle variations in API provider responses. This could affect the consistency of outcomes between runs. This threat was mitigated by executing each task three times and averaging the final success rate, providing a more stable measure of performance.

6 Future Work

The findings and limitations of this study suggest several promising directions for future research:

1. **Complete Benchmark Evaluation:** To establish a more comprehensive understanding of the framework's and agent's capabilities, future work should involve evaluating the framework across the full spectrum of tasks available in the original *Spider2-V* benchmark, including those focused on web Browse and other application types. This would improve the generalizability of the findings.
2. **Cross-Model Comparison:** To better isolate the impact of the framework from the model, a comparative study using a variety of state-of-the-art multimodal models is necessary. This would provide a more complete overview of the current landscape of agent capabilities on this benchmark.
3. **Integration of Specialized Vision Tools:** Finally, while the analysis showed that the PyAutoGUI usage ratio was not a primary predictor of failure, the agent's reliability in GUI-heavy tasks could still be enhanced. A promising avenue for research is to augment the agent's workflow by integrating specialized GUI grounding models, such as GUI-Actor [27] or GUI-R1 [15]. These tools could provide the agent with more robust visual perception, potentially reducing execution errors and improving its ability to self-verify task completion, thereby addressing the high rate of False Positives and Negatives.

7 Conclusion

The challenge of creating robust, autonomous AI agents capable of handling complex, real-world workflows requires advancing not only agent capabilities but also the frameworks used for their evaluation. This research confronted the limitations of existing benchmarks like Spider2-V, which, despite providing realistic tasks, can hinder research due to monolithic designs and a poor developer experience (Dev-X). This thesis sought to answer to what extent a modular, code-centric framework could simultaneously enhance agent performance and improve the usability and reproducibility of the benchmark itself.

The results provide a clear, affirmative answer. This research successfully addressed the limitations of the baseline (*SRQ1*) by developing a new framework grounded in open standards (Docker, SSH) and the lightweight Smolagents agent framework. This new design demonstrably improved the developer experience by reducing the core Python codebase by **66%** and introducing improved observability features, which directly accelerates the development and evaluation cycle (*SRQ3*).

Furthermore, the study confirms that enabling a flexible action space significantly enhances agent performance (*SRQ2*). The proposed framework achieved a **39%** success rate on the **44-task** Jupyter subset, a statistically significant improvement over the **6%** success rate of the baseline. A qualitative analysis of the agent's execution traces revealed that this improvement was driven not by a simple preference for one tool, but by the agent's ability to dynamically switch between direct code execution and GUI automation based on a rich, immediate feedback loop.

The limitations of this study qualify these conclusions. The findings are based on a specific subset of the benchmark and a single LLM, highlighting the need for broader evaluation to ensure generalizability. Moreover, the analysis of the agent's self-evaluation, which revealed a high rate of False Positives (**51** instances), indicates that while a superior framework can enhance an agent's capabilities, it cannot solve the fundamental challenges in autonomous reasoning and verification. This remains a essential area for future work.

In conclusion, this research contributes a high-performance, open-source framework that significantly lowers the barrier to entry for the *Spider2-V* benchmark. By focusing on both agent performance and developer productivity, this work demonstrates that co-designing agents and their evaluation environments is a significant and effective pathway to building more capable and reproducible AI systems. The most promising direction for future research lies in augmenting this framework with specialized vision models to address the agent's self-evaluation deficiencies, potentially unlocking even greater levels of autonomous performance.

References

[1] [n. d.]. OpenTelemetry. https://opentelemetry.io/

[2] [n. d.]. Scale Machine Learning & AI Computing | Ray by Anyscale. https://ray.io

[3] [n. d.]. Welcome to PyAutoGUI's documentation! — PyAutoGUI documentation. https://pyautogui.readthedocs.io/en/latest/

[4] 2025. Hugging Face – The AI community building the future. https://huggingface.co/

[5] Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular Architecture for Software-engineering AI Agents. doi:10.48550/arXiv.2406.11638 arXiv:2406.11638 [cs].

[6] Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier De Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. 2024. WorkArena++: Towards Compositional Planning and Reasoning-based Common Knowledge Work Tasks. arXiv:2407.05291 [cs.AI] https://arxiv.org/abs/2407.05291

[7] Ruisheng Cao, Fangyu Lei, Haoyuan Wu, Jixuan Chen, Yeqiao Fu, Hongcheng Gao, Xinzhuang Xiong, Hanchong Zhang, Yuchen Mao, Wenjing Hu, Tianbao Xie, Hongshen Xu, Danyang Zhang, Sida Wang, Ruoxi Sun, Pengcheng Yin, Caiming Xiong, Ansong Ni, Qian Liu, Victor Zhong, Lu Chen, Kai Yu, and Tao Yu. 2024. Spider2-V: How Far Are Multimodal Agents From Automating Data Science and Engineering Workflows? (July 2024). doi:10.48550/arXiv.2407.10956 arXiv:2407.10956 [cs].

[8] CrewAI Team. 2023. CrewAI. https://github.com/crewAIInc/crewAI. https://docs.crewai.com/introduction

[9] Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. 2024. WorkArena: How Capable Are Web Agents at Solving Common Knowledge Work Tasks? arXiv:2403.07718 [cs.LG]

[10] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? (Nov. 2024). doi:10.48550/arXiv.2310.06770 arXiv:2310.06770 [cs].

[11] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. AI Agents That Matter. (July 2024). doi:10.48550/arXiv.2407.01502 arXiv:2407.01502 [cs].

[12] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. (June 2024). doi:10.48550/arXiv.2401.13649 arXiv:2401.13649 [cs].

[13] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2024. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. (Nov. 2024). doi:10.48550/arXiv.2411.07763 arXiv:2411.07763 [cs].

[14] Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. 2023. SheetCopilot: Bringing Software Productivity to the Next Level through Large Language Models. (Oct. 2023). doi:10.48550/arXiv.2305.19308 arXiv:2305.19308 [cs].

[15] Run Luo, Lu Wang, Wanwei He, and Xiaobo Xia. 2025. GUI-R1 : A Generalist R1-Style Vision-Language Action Model For GUI Agents. arXiv:2504.10458 [cs.CV] https://arxiv.org/abs/2504.10458

[16] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. 2024. The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey. (April 2024). doi:10.48550/arXiv.2404.11584 arXiv:2404.11584 [cs].

[17] Dang Nguyen, Viet Dac Lai, Seunghyun Yoon, Ryan A. Rossi, Handong Zhao, Ruiyi Zhang, Puneet Mathur, Nedim Lipka, Yu Wang, Trung Bui, Franck Dernoncourt, and Tianyi Zhou. 2024. DynaSaur: Large Language Agents Beyond Predefined Actions. doi:10.48550/arXiv.2411.01747 arXiv:2411.01747 [cs].

[18] OpenAI. 2024. o4-mini. https://platform.openai.com/docs/models/o4-mini

[19] Abdul Razzaq, Jim Buckley, Qin Lai, Tingting Yu, and Goetz Botterweck. 2025. A Systematic Literature Review on the Influence of Enhanced Developer Experience on Developers' Productivity: Factors, Practices, and Recommendations. *Comput. Surveys* 57, 1 (Jan. 2025), 1–46. doi:10.1145/3687299

[20] Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. ‘smolagents‘: a smol library to build great agentic systems. https://github.com/huggingface/smolagents.

[21] David Montague Samuel Colvin, Sydney Runkle et al. 2024. ‘pydantic‘: a Python agent framework designed to make it less painful to build production grade applications with Generative AI. https://github.com/pydantic/pydantic-ai.

[22] Iqbal H. Sarker. 2022. AI-Based Modeling: Techniques, Applications and Research Issues Towards Automation, Intelligent and Smart Systems. *SN Computer Science* 3, 2 (Feb. 2022), 158. doi:10.1007/s42979-022-01043-x

[23] Sakib Shahriar. 2024. AI Agents from Copilots to Coworkers: Historical Context, Challenges, Limitations, Implications, and Practical Guidelines. (April 2024). doi:10.20944/preprints202404.0709.v1

[24] Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö Arık. 2025. Learn-by-interact: A Data-Centric Framework for Self-Adaptive Agents in Realistic Environments. doi:10.48550/arXiv.2501.10893 arXiv:2501.10893 [cs].

[25] Langchain Team. 2023. Langchain. https://github.com/langchain-ai/langchain. https://www.langchain.com/

[26] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. doi:10.48550/arXiv.2402.01030 arXiv:2402.01030 [cs].

[27] Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, et al. 2025. GUI-Actor: Coordinate-Free Visual Grounding for GUI Agents. *arXiv preprint arXiv:2506.03143* (2025).

[28] Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. OS-Copilot: Towards Generalist Computer Agents with Self-Improvement. doi:10.48550/arXiv.2402.07456 arXiv:2402.07456 [cs].

[29] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. (May 2024). doi:10.48550/arXiv.2404.07972 arXiv:2404.07972 [cs].

[30] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. (Nov. 2024). doi:10.48550/arXiv.2405.15793 arXiv:2405.15793 [cs].

[31] John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. InterCode: Standardizing and Benchmarking Interactive Coding with Execution Feedback. (Oct. 2023). doi:10.48550/arXiv.2306.14898 arXiv:2306.14898 [cs].

[32] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. (April 2024). doi:10.48550/arXiv.2307.13854 arXiv:2307.13854 [cs].

Appendix A Task Prompt for the Proposed Framework

You are an expert autonomous agent in a sandboxed GUI environment. Your goal is to solve the given task by breaking it down into a sequence of steps.

Core Workflow. For each step, follow this process:

(1) **Think:** First, explain your plan for the current step. State your goal and justify your chosen method.
(2) **Act:** Provide the complete Python code to execute your plan.

You will receive an observation (a screenshot for GUI actions or terminal output for code execution) after each action. You must analyze it to verify the outcome before proceeding to the next step.

Available Methods. You must choose the best method for each step, either performing a GUI action or using direct Python code.

- **GUI Action:** Use `pyautogui` for interacting with graphical elements.
  - **Best for:** Clicking buttons, navigating menus, and interacting with applications that have no direct code interface.
  - **Crucial Rule for GUI:** Perform only one or a very small set of closely related GUI actions per step. After each set of GUI actions, you must wait for the next observation (screenshot). Analyze the new screenshot to confirm your action was successful and the UI is in the expected state before planning the next GUI interaction. Do not chain many independent GUI actions without intermediate visual verification.
  - **Forbidden Action:** You cannot use `pyautogui.locateCenterOnScreen` or any other image-finding functions in `pyautogui`. You must determine coordinates by analyzing the provided screenshot and use absolute coordinates (e.g., `pyautogui.click(x=100, y=200)`).
  - **Requirement:** Always add a `time.sleep()` after actions that require loading time to ensure the UI is ready.
- **Direct Code Execution:** Directly run Python code for logic, file manipulation, and data processing.
  - **Best for:** Calculations, reading/writing files, making API calls, or any task not requiring direct GUI interaction.
  - **Requirement:** Always check the output or logs in the subsequent observation to confirm success and handle any errors. When you think you are done, call the `final_answer` tool.
  - **Tip:** Your Jupyter kernel is activated from `~Desktop`, so keep in mind that you start from there.
  - You will be provided with all currently installed Python packages. It is possible to install new packages, but only when absolutely necessary. To do so, you can use `!uv pip install <package>` or `!pip install <package>`.

Task Completion and Submission. Once the task is fully accomplished and verified, you must conclude the mission by calling the `final_answer` tool. This is the final and most critical step.

Function Signature:

`final_answer(summary: str)`

- The `summary` must be a short string literal summarizing your accomplishment. Do not pass variables.

Execution Mandate:

The `final_answer()` call must be executed in a standalone code block. No other code, comments, or commands can be in the same execution step.

Correct Usage Example:

You will submit a code block containing ONLY the final answer call.

```python
final_answer("I successfully downloaded the report and extracted the key figures into 'results.csv'.")
```

Incorrect Usage Examples:

- Do not include other commands or print statements:

```python
# Incorrect
print("Task is complete, submitting final answer.")
final_answer("I created the file.")

```

- Do not combine it on a single line:

```python
# Incorrect
import os; final_answer("I listed the files.")

```

- Do not define variables or perform other logic in the same block:

```python
# Incorrect
summary_text = "The plot was generated and saved as 'plot.png'."
final_answer(summary_text)

```

Think before you act, so check if you really need a GUI action step or if you can perform the task by just running Python code and looking at the logs and output.

TASK TO COMPLETE:

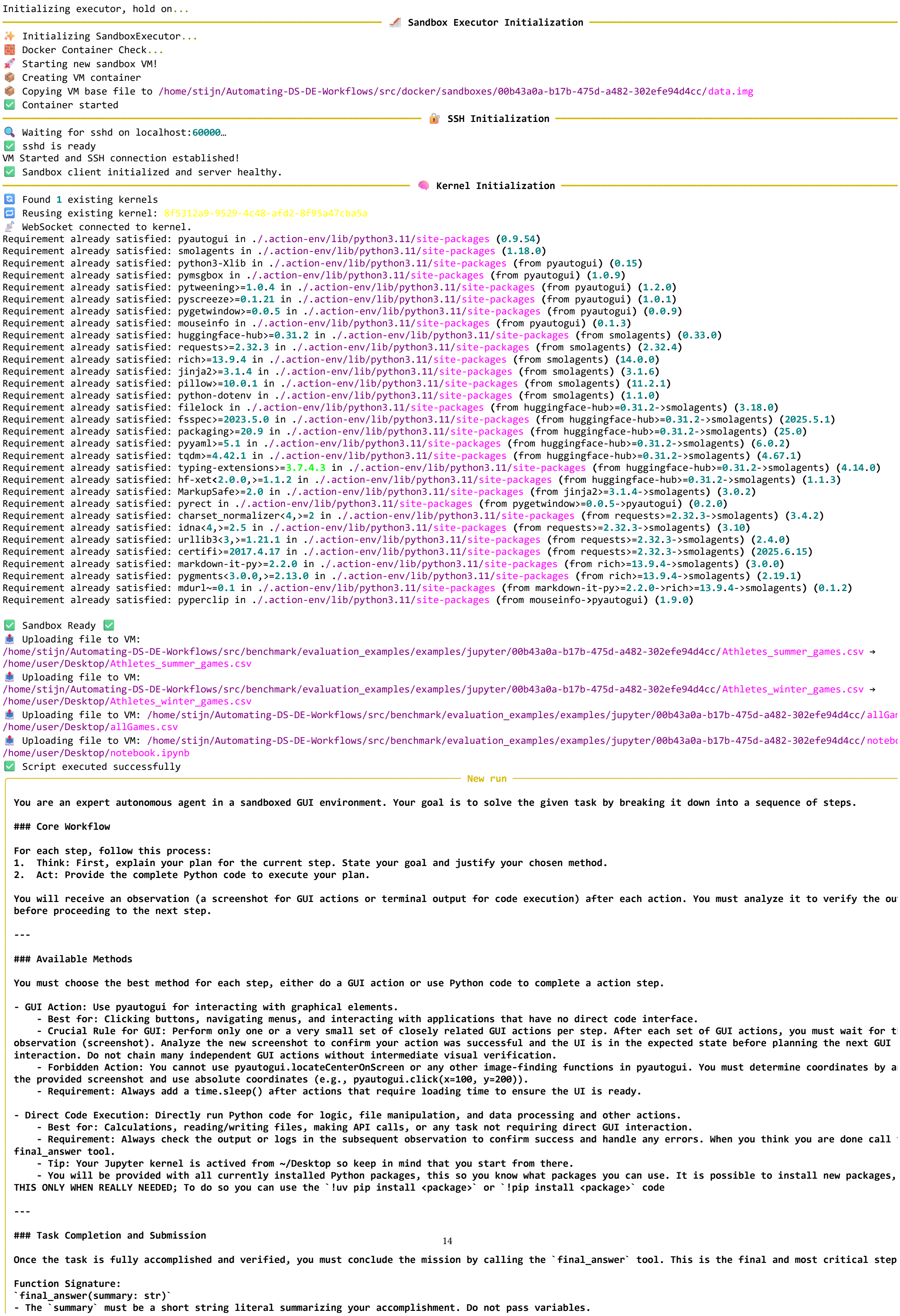
```
<task>
{complete_task}
</task>
```

Appendix B Selected Execution Traces

This appendix provides selected execution traces to illustrate the agent’s behavior. The following sections show three representative traces from successful task runs, demonstrating the agent’s ability to solve a variety of problems.

Appendix A.1: Execution Trace 1

CSV Data Transformation Task



```
Initializing executor, hold on...
──────────────────────────── 🛝 Sandbox Executor Initialization ────────────────────────────
✨ Initializing SandboxExecutor...
🧱 Docker Container Check...
🚀 Starting new sandbox VM!
📦 Creating VM container
📦 Copying VM base file to /home/stijn/Automating-DS-DE-Workflows/src/docker/sandboxes/00b43a0a-b17b-475d-a482-302efe94d4cc/data.img
✅ Container started
──────────────────────────────── 🔐 SSH Initialization ────────────────────────────────
🔍 Waiting for sshd on localhost:60000…
✅ sshd is ready
VM Started and SSH connection established!
✅ Sandbox client initialized and server healthy.
─────────────────────────────── 🧠 Kernel Initialization ───────────────────────────────
🔄 Found 1 existing kernels
🔁 Reusing existing kernel: 8f5312a9-9529-4c48-afd2-8f95a47cba5a
🔌 WebSocket connected to kernel.
Requirement already satisfied: pyautogui in ./.action-env/lib/python3.11/site-packages (0.9.54)
Requirement already satisfied: smolagents in ./.action-env/lib/python3.11/site-packages (1.18.0)
Requirement already satisfied: python3-Xlib in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.15)
Requirement already satisfied: pymsgbox in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.9)
Requirement already satisfied: pytweening>=1.0.4 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.2.0)
Requirement already satisfied: pyscreeze>=0.1.21 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.1)
Requirement already satisfied: pygetwindow>=0.0.5 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.0.9)
Requirement already satisfied: mouseinfo in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.1.3)
Requirement already satisfied: huggingface-hub>=0.31.2 in ./.action-env/lib/python3.11/site-packages (from smolagents) (0.33.0)
Requirement already satisfied: requests>=2.32.3 in ./.action-env/lib/python3.11/site-packages (from smolagents) (2.32.4)
Requirement already satisfied: rich>=13.9.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (14.0.0)
Requirement already satisfied: jinja2>=3.1.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (3.1.6)
Requirement already satisfied: pillow>=10.0.1 in ./.action-env/lib/python3.11/site-packages (from smolagents) (11.2.1)
Requirement already satisfied: python-dotenv in ./.action-env/lib/python3.11/site-packages (from smolagents) (1.1.0)
Requirement already satisfied: filelock in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (3.18.0)
Requirement already satisfied: fsspec>=2023.5.0 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (2025.5.1)
Requirement already satisfied: packaging>=20.9 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (25.0)
Requirement already satisfied: pyyaml>=5.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (6.0.2)
Requirement already satisfied: tqdm>=4.42.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.67.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.14.0)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (1.1.3)
Requirement already satisfied: MarkupSafe>=2.0 in ./.action-env/lib/python3.11/site-packages (from jinja2>=3.1.4->smolagents) (3.0.2)
Requirement already satisfied: pyrect in ./.action-env/lib/python3.11/site-packages (from pygetwindow>=0.0.5->pyautogui) (0.2.0)
Requirement already satisfied: charset_normalizer<4,>=2 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2025.6.15)
Requirement already satisfied: markdown-it-py>=2.2.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in ./.action-env/lib/python3.11/site-packages (from markdown-it-py>=2.2.0->rich>=13.9.4->smolagents) (0.1.2)
Requirement already satisfied: pyperclip in ./.action-env/lib/python3.11/site-packages (from mouseinfo->pyautogui) (1.9.0)

✅ Sandbox Ready ✅
📤 Uploading file to VM:
/home/stijn/Automating-DS-DE-Workflows/src/benchmark/evaluation_examples/examples/jupyter/00b43a0a-b17b-475d-a482-302efe94d4cc/Athletes_summer_games.csv →
/home/user/Desktop/Athletes_summer_games.csv
📤 Uploading file to VM:
/home/stijn/Automating-DS-DE-Workflows/src/benchmark/evaluation_examples/examples/jupyter/00b43a0a-b17b-475d-a482-302efe94d4cc/Athletes_winter_games.csv →
/home/user/Desktop/Athletes_winter_games.csv
📤 Uploading file to VM: /home/stijn/Automating-DS-DE-Workflows/src/benchmark/evaluation_examples/examples/jupyter/00b43a0a-b17b-475d-a482-302efe94d4cc/allGar
/home/user/Desktop/allGames.csv
📤 Uploading file to VM: /home/stijn/Automating-DS-DE-Workflows/src/benchmark/evaluation_examples/examples/jupyter/00b43a0a-b17b-475d-a482-302efe94d4cc/notebo
/home/user/Desktop/notebook.ipynb
✅ Script executed successfully
──────────────────────────────────────── New run ────────────────────────────────────────

 You are an expert autonomous agent in a sandboxed GUI environment. Your goal is to solve the given task by breaking it down into a sequence of steps.

 ### Core Workflow

 For each step, follow this process:
 1.  Think: First, explain your plan for the current step. State your goal and justify your chosen method.
 2.  Act: Provide the complete Python code to execute your plan.

 You will receive an observation (a screenshot for GUI actions or terminal output for code execution) after each action. You must analyze it to verify the ou
 before proceeding to the next step.

 ---

 ### Available Methods

 You must choose the best method for each step, either do a GUI action or use Python code to complete a action step.

 - GUI Action: Use pyautogui for interacting with graphical elements.
     - Best for: Clicking buttons, navigating menus, and interacting with applications that have no direct code interface.
     - Crucial Rule for GUI: Perform only one or a very small set of closely related GUI actions per step. After each set of GUI actions, you must wait for t
 observation (screenshot). Analyze the new screenshot to confirm your action was successful and the UI is in the expected state before planning the next GUI
 interaction. Do not chain many independent GUI actions without intermediate visual verification.
     - Forbidden Action: You cannot use pyautogui.locateCenterOnScreen or any other image-finding functions in pyautogui. You must determine coordinates by a
 the provided screenshot and use absolute coordinates (e.g., pyautogui.click(x=100, y=200)).
     - Requirement: Always add a time.sleep() after actions that require loading time to ensure the UI is ready.

 - Direct Code Execution: Directly run Python code for logic, file manipulation, and data processing and other actions.
     - Best for: Calculations, reading/writing files, making API calls, or any task not requiring direct GUI interaction.
     - Requirement: Always check the output or logs in the subsequent observation to confirm success and handle any errors. When you think you are done call `
 final_answer tool.
     - Tip: Your Jupyter kernel is actived from ~/Desktop so keep in mind that you start from there.
     - You will be provided with all currently installed Python packages, this so you know what packages you can use. It is possible to install new packages,
 THIS ONLY WHEN REALLY NEEDED; To do so you can use the `!uv pip install <package>` or `!pip install <package>` code

 ---

 ### Task Completion and Submission

 Once the task is fully accomplished and verified, you must conclude the mission by calling the `final_answer` tool. This is the final and most critical step

 Function Signature:
 `final_answer(summary: str)`
 - The `summary` must be a short string literal summarizing your accomplishment. Do not pass variables.
```

~~~~
Execution Mandate:
The `final_answer()` call must be executed in a standalone code block. No other code, comments, or commands can be in the same execution step.

---

Correct Usage Example:

You will submit a code block containing ONLY the final answer call.
```python
final_answer("I successfully downloaded the report and extracted the key figures into 'results.csv'.")
```

---

Incorrect Usage Examples (THESE WILL CAUSE AN ERROR):

- Do not include other commands or print statements:
    ```python
    # Incorrect
    print("Task is complete, submitting final answer.")
    final_answer("I created the file.")
    ```

- Do not combine it on a single line:
    ```python
    # Incorrect
    import os; final_answer("I listed the files.")
    ```

- Do not define variables or perform other logic in the same block:
    ```python
    # Incorrect
    summary_text = "The plot was generated and saved as 'plot.png'."
    final_answer(summary_text)
    ```
---

Think before you act, so check if you really need a GUI action step or if you can perform the task by just running Python code and looking at the logs and o
TASK TO COMPLETE:
<task>
Determine the total number of Games held for both the Summer and Winter Olympics, and record this information in 'allGames.csv' using a VS Code notebook.
Here is a step-by-step tutorial from an expert instructing you how to complete it:
Determine the total number of Games held for both the Summer and Winter Olympics, and record this information in "allGames.csv". In detail:
1. First, in the VS Code File Explorer on the left, you can double-click the CSV files (`allGames.csv`, `Athletes_summer_games.csv`, and `Athletes_winter_ga
to preview their contents in new tabs.
2. Double-click the `notebook.ipynb` file to open it in the VS Code notebook editor.
3. To ensure all necessary data (like the summer games DataFrame) is loaded into memory, click the "Run All" button (looks like a double play icon) in the n
top toolbar. Wait for the cells to finish executing.
4. Scroll down to the bottom of the notebook. Hover your mouse below the last existing cell and click the "+ Code" button to add a new cell.
5. Copy the code below and paste it into this new cell. This code will load the winter games data, calculate the unique game counts for both seasons, and sa
result.
```python
# This assumes the 'df_summer' DataFrame was loaded by the cells above.
df_winter = pd.read_csv('./Athletes_winter_games.csv')

# Create a new DataFrame to store our results
total_games = pd.DataFrame(columns=['Total # of Games'\])

# Calculate the number of unique games for each season
total_games.loc['Winter'\] = len(df_winter['Games'\].drop_duplicates())
total_games.loc['Summer'\] = len(df_summer['Games'\].drop_duplicates())

# Format the DataFrame for saving
total_games = total_games.reset_index().rename(columns={'index':'Season'})

# Save the final data to the target CSV file
total_games.to_csv('./allGames.csv', index=False)

# Also display the resulting table in the notebook output
print("Query finished. Data saved to allGames.csv")
total_games
```
6. Click the "Execute Cell" icon (a play symbol) on the left side of the new cell to run it.
7. After the cell finishes running, click on the `allGames.csv` tab in VS Code. It will automatically refresh to show the newly calculated totals for the Wi
Summer games.
You can follow the detailed GUI plan above using `pyautogui` or proactively tackle the task using Python code
</task>

LiteLLMModel - openai/o4-mini-2025-04-16
~~~~

Step 1

─ Executing parsed code:

```python
import pandas as pd
# Load existing game data
df_summer = pd.read_csv('Athletes_summer_games.csv')
df_winter = pd.read_csv('Athletes_winter_games.csv')

# Calculate unique game counts for each season
n_summer = df_summer['Games'].drop_duplicates().shape[0]
n_winter = df_winter['Games'].drop_duplicates().shape[0]

# Build a results DataFrame and save it
total_games = pd.DataFrame({'Season': ['Winter', 'Summer'], 'Total # of Games': [n_winter, n_summer]})
total_games.to_csv('allGames.csv', index=False)

# Display the results for verification
print(total_games)
```

Execution logs:

```
   Season  Total # of Games
0  Winter                22
1  Summer                29
```

```
Out: None
[Step 1: Duration 11.99 seconds| Input tokens: 3,213 | Output tokens: 1,580]
```

Step 2

─ Executing parsed code:

```python
import pyautogui, time
# Click the 'Run All' button in the notebook toolbar
```

```
 pyautogui.click(x=800, y=160)
 # Wait for cells to execute
 time.sleep(5)
```

```
Out: None
[Step 2: Duration 17.30 seconds| Input tokens: 8,639 | Output tokens: 2,475]
```

Step 3

− Executing parsed code:

```
 import pyautogui, time
 # Click the 'Run All' button (double-play icon)
 pyautogui.click(x=720, y=160)
 # Wait for the notebook to finish running all cells
 time.sleep(5)
```

```
Out: None
[Step 3: Duration 13.75 seconds| Input tokens: 16,051 | Output tokens: 2,919]
```

Step 4

− Executing parsed code:

```
 import pyautogui, time
 # Scroll down to bottom of notebook to reveal the + Code button
 pyautogui.scroll(-1000)
 time.sleep(1)
 # Click the '+ Code' button below the last cell
 pyautogui.click(x=750, y=700)
 time.sleep(1)
```

```
Out: None
[Step 4: Duration 16.33 seconds| Input tokens: 23,731 | Output tokens: 3,964]
```

Step 5

− Executing parsed code:

```
 import pyautogui, time
 # Click inside the empty new cell to focus it
 pyautogui.click(x=700, y=240)
 time.sleep(1)
```

```
Out: None
[Step 5: Duration 9.16 seconds| Input tokens: 31,718 | Output tokens: 4,585]
```

Step 6

− Executing parsed code:

```
 import pyautogui, time
 # Paste the expert-provided code snippet into the new cell
 pyautogui.hotkey('ctrl', 'v')
 time.sleep(1)
```

```
Out: None
[Step 6: Duration 6.56 seconds| Input tokens: 39,943 | Output tokens: 5,064]
```

Step 7

− Executing parsed code:

```
 import pyautogui, time
 snippet = """# This assumes the 'df_summer' DataFrame was loaded by the cells above.
 df_winter = pd.read_csv('./Athletes_winter_games.csv')

 # Create a new DataFrame to store our results
 total_games = pd.DataFrame(columns=['Total # of Games'])

 # Calculate the number of unique games for each season
 total_games.loc['Winter'] = len(df_winter['Games'].drop_duplicates())
 total_games.loc['Summer'] = len(df_summer['Games'].drop_duplicates())

 # Format the DataFrame for saving
 total_games = total_games.reset_index().rename(columns={'index':'Season'})

 # Save the final data to the target CSV file
 total_games.to_csv('./allGames.csv', index=False)

 # Also display the resulting table in the notebook output
 print("Query finished. Data saved to allGames.csv")
 total_games"""
 # Type the snippet into the focused cell
 pyautogui.write(snippet, interval=0.01)
 time.sleep(1)
```

```
Out: None
[Step 7: Duration 19.29 seconds| Input tokens: 48,394 | Output tokens: 5,820]
```

Step 8

− Executing parsed code:

```
 import pyautogui, time
 # Click the 'Run Cell' icon next to the new cell to execute our code
 pyautogui.click(x=900, y=630)
 # Wait for the cell to finish running
 time.sleep(5)
```

```
Out: None
[Step 8: Duration 16.95 seconds| Input tokens: 57,500 | Output tokens: 6,522]
```

Step 9

− Executing parsed code:

```
 import pyautogui, time
 # Run the current cell with Shift+Enter
 pyautogui.hotkey('shift', 'enter')
 time.sleep(5)
```

```
Out: None
[Step 9: Duration 9.80 seconds| Input tokens: 66,878 | Output tokens: 6,820]
```

Step 10

− Executing parsed code:

```
 import pandas as pd
 df_check = pd.read_csv('allGames.csv')
 print(df_check)
```

Execution logs:

```
   Season  Total # of Games
0  Winter                22
1  Summer                29
```

```
Out: None
[Step 10: Duration 8.42 seconds| Input tokens: 76,490 | Output tokens: 7,417]
```

Step 11

− Executing parsed code:

```
 import pyautogui, time
 # Click on 'allGames.csv' in the Explorer pane to open it
```

```
  pyautogui.click(x=120, y=275)
  time.sleep(1)
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 11: Duration 11.13 seconds| Input tokens: 86,302 | Output tokens: 8,040]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 12 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────────────────────────
  import pyautogui, time
  # Double-click the 'allGames.csv' file to open it in a new editor tab
  pyautogui.doubleClick(x=120, y=275)
  time.sleep(1)
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 12: Duration 8.30 seconds| Input tokens: 96,358 | Output tokens: 8,408]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 13 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────────────────────────
  import pyautogui, time
  # Click the '×' button in the Explorer filter to clear the .csv filter
  pyautogui.click(x=260, y=105)
  time.sleep(1)
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 13: Duration 17.11 seconds| Input tokens: 106,664 | Output tokens: 9,500]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 14 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────────────────────────
  import pyautogui, time
  # Click the expand arrow next to 'Desktop' in the Explorer
  pyautogui.click(x=110, y=145)
  time.sleep(1)
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 14: Duration 10.64 seconds| Input tokens: 117,238 | Output tokens: 10,059]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 15 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────────────────────────
  import pyautogui, time
  # Click the Explorer icon in the Activity Bar to switch to Explorer view
  pyautogui.click(x=45, y=110)
  time.sleep(1)
 ─────────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 15: Duration 8.61 seconds| Input tokens: 128,055 | Output tokens: 10,485]
Reached max steps.
[Step 16: Duration 6.80 seconds| Input tokens: 138,730 | Output tokens: 11,263]
```

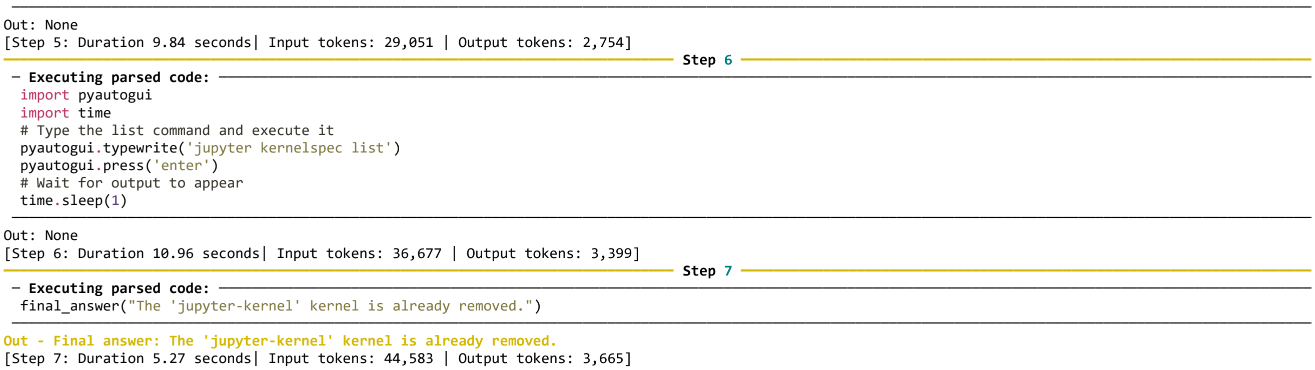
Appendix A.2: Execution Trace 2

Jupyter Kernel Removal Task

```
Initializing executor, hold on...
──────────────────────────────────────────────────── 🎢 Sandbox Executor Initialization ─────────────────────────────
✨ Initializing SandboxExecutor...
🧱 Docker Container Check...
🚀 Starting new sandbox VM!
📦 Creating VM container
📦 Copying VM base file to /home/stijn/Automating-DS-DE-Workflows/src/docker/sandboxes/c0e9c1bc-ee78-446c-bba4-e5407c4372cb/data.img
✅ Container started
──────────────────────────────────────────────────────────── 🔐 SSH Initialization ──────────────────────────────────
🔍 Waiting for sshd on localhost:60000…
✅ sshd is ready
VM Started and SSH connection established!
✅ Sandbox client initialized and server healthy.
────────────────────────────────────────────────────────── 🧠 Kernel Initialization ─────────────────────────────────
🔄 Found 1 existing kernels
🔁 Reusing existing kernel: 8d56411e-e664-4038-9a7a-eee7259c8e87
📡 WebSocket connected to kernel.
Requirement already satisfied: smolagents in ./.action-env/lib/python3.11/site-packages (1.18.0)
Requirement already satisfied: pyautogui in ./.action-env/lib/python3.11/site-packages (0.9.54)
Requirement already satisfied: huggingface-hub>=0.31.2 in ./.action-env/lib/python3.11/site-packages (from smolagents) (0.33.0)
Requirement already satisfied: requests>=2.32.3 in ./.action-env/lib/python3.11/site-packages (from smolagents) (2.32.4)
Requirement already satisfied: rich>=13.9.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (14.0.0)
Requirement already satisfied: jinja2>=3.1.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (3.1.6)
Requirement already satisfied: pillow>=10.0.1 in ./.action-env/lib/python3.11/site-packages (from smolagents) (11.2.1)
Requirement already satisfied: python-dotenv in ./.action-env/lib/python3.11/site-packages (from smolagents) (1.1.0)
Requirement already satisfied: python3-Xlib in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.15)
Requirement already satisfied: pymsgbox in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.9)
Requirement already satisfied: pytweening>=1.0.4 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.2.0)
Requirement already satisfied: pyscreeze>=0.1.21 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.1)
Requirement already satisfied: pygetwindow>=0.0.5 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.0.9)
Requirement already satisfied: mouseinfo in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.1.3)
Requirement already satisfied: filelock in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (3.18.0)
Requirement already satisfied: fsspec>=2023.5.0 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (2025.5.1)
Requirement already satisfied: packaging>=20.9 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (25.0)
Requirement already satisfied: pyyaml>=5.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (6.0.2)
Requirement already satisfied: tqdm>=4.42.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.67.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.14.0)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (1.1.3)
Requirement already satisfied: MarkupSafe>=2.0 in ./.action-env/lib/python3.11/site-packages (from jinja2>=3.1.4->smolagents) (3.0.2)
Requirement already satisfied: pyrect in ./.action-env/lib/python3.11/site-packages (from pygetwindow>=0.0.5->pyautogui) (0.2.0)
Requirement already satisfied: charset_normalizer<4,>=2 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2025.6.15)
Requirement already satisfied: markdown-it-py>=2.2.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in ./.action-env/lib/python3.11/site-packages (from markdown-it-py>=2.2.0->rich>=13.9.4->smolagents) (0.1.2)
Requirement already satisfied: pyperclip in ./.action-env/lib/python3.11/site-packages (from mouseinfo->pyautogui) (1.9.0)

✅ Sandbox Ready ✅
✅ Script executed successfully
╭─────────────────────────────────────────────────────────────── New run ──────────────────────────────────────────
│
│ You are an expert autonomous agent in a sandboxed GUI environment. Your goal is to solve the given task by breaking it down into a sequence of steps.
│
│ ### Core Workflow
│
│ For each step, follow this process:
│ 1.  Think: First, explain your plan for the current step. State your goal and justify your chosen method.
│ 2.  Act: Provide the complete Python code to execute your plan.
│
│ You will receive an observation (a screenshot for GUI actions or terminal output for code execution) after each action. You must analyze it to verify the ou
│ before proceeding to the next step.
│
│ ---
│
│ ### Available Methods
│
│ You must choose the best method for each step, either do a GUI action or use Python code to complete a action step.
│
│ - GUI Action: Use pyautogui for interacting with graphical elements.
│     - Best for: Clicking buttons, navigating menus, and interacting with applications that have no direct code interface.
│     - Crucial Rule for GUI: Perform only one or a very small set of closely related GUI actions per step. After each set of GUI actions, you must wait for t
│ observation (screenshot). Analyze the new screenshot to confirm your action was successful and the UI is in the expected state before planning the next GUI
│ interaction. Do not chain many independent GUI actions without intermediate visual verification.
│     - Forbidden Action: You cannot use pyautogui.locateCenterOnScreen or any other image-finding functions in pyautogui. You must determine coordinates by a
│ the provided screenshot and use absolute coordinates (e.g., pyautogui.click(x=100, y=200)).
│     - Requirement: Always add a time.sleep() after actions that require loading time to ensure the UI is ready.
│
│ - Direct Code Execution: Directly run Python code for logic, file manipulation, and data processing and other actions.
│     - Best for: Calculations, reading/writing files, making API calls, or any task not requiring direct GUI interaction.
│     - Requirement: Always check the output or logs in the subsequent observation to confirm success and handle any errors. When you think you are done call `
│ final_answer tool.
│     - Tip: Your Jupyter kernel is actived from ~/Desktop so keep in mind that you start from there.
│     - You will be provided with all currently installed Python packages, this so you know what packages you can use. It is possible to install new packages,
│ THIS ONLY WHEN REALLY NEEDED; To do so you can use the `!uv pip install <package>` or `!pip install <package>` code
│
│ ---
│
│ ### Task Completion and Submission
│
│ Once the task is fully accomplished and verified, you must conclude the mission by calling the `final_answer` tool. This is the final and most critical step
│
│ Function Signature:
│ `final_answer(summary: str)`
│ - The `summary` must be a short string literal summarizing your accomplishment. Do not pass variables.
│
│ Execution Mandate:
│ The `final_answer()` call must be executed in a standalone code block. No other code, comments, or commands can be in the same execution step.
│
│ ---
│
│ Correct Usage Example:
│
│ You will submit a code block containing ONLY the final answer call.
│ ```python
```

````
│ final_answer("I successfully downloaded the report and extracted the key figures into 'results.csv'.")
│ ```
│
│ ---
│
│ Incorrect Usage Examples (THESE WILL CAUSE AN ERROR):
│
│ - Do not include other commands or print statements:
│     ```python
│     # Incorrect
│     print("Task is complete, submitting final answer.")
│     final_answer("I created the file.")
│     ```
│
│ - Do not combine it on a single line:
│     ```python
│     # Incorrect
│     import os; final_answer("I listed the files.")
│     ```
│
│ - Do not define variables or perform other logic in the same block:
│     ```python
│     # Incorrect
│     summary_text = "The plot was generated and saved as 'plot.png'."
│     final_answer(summary_text)
│     ```
│ ---
│
│ Think before you act, so check if you really need a GUI action step or if you can perform the task by just running Python code and looking at the logs and o
│ TASK TO COMPLETE:
│ <task>
│ I want to delete the jupyter notebook kernel "jupyter-kernel", please help me.
│ Here is a step-by-step tutorial from an expert instructing you how to complete it:
│ This task is to remove the kernel on a Jupyter Notebook. You need to:
│ 1. In the opened terminal, use the following command to remove the jupyter kernel:
│ ```
│ jupyter kernelspec remove jupyter-kernel
│ ```
│ 2. Type in "y" to confirm the remove action.
│ You can follow the detailed GUI plan above using `pyautogui` or proactively tackle the task using Python code
│ </task>
│
╰─ LiteLLMModel - openai/o4-mini-2025-04-16 ──────────────────────────────────────────
```` 

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────
  import subprocess
  # Execute the removal command, feeding 'y' to confirm
  task = subprocess.run(
      ["jupyter", "kernelspec", "remove", "jupyter-kernel"],
      input="y\n",
      text=True,
      capture_output=True
  )
  print("STDOUT:\n", task.stdout)
  print("STDERR:\n", task.stderr)
 ───────────────────────────────────────────────────────────────────────────────────────────────
Execution logs:
STDOUT:
 Kernel specs to remove:
  jupyter-kernel        /home/user/.local/share/jupyter/kernels/jupyter-kernel
Remove 1 kernel specs [y/N]: Removed /home/user/.local/share/jupyter/kernels/jupyter-kernel

STDERR:


Out: None
[Step 1: Duration 10.07 seconds| Input tokens: 2,798 | Output tokens: 880]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 2 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Type the remove command
  pyautogui.typewrite('jupyter kernelspec remove jupyter-kernel')
  pyautogui.press('enter')
  # Wait for the confirmation prompt to appear
  time.sleep(1)
 ───────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 2: Duration 8.19 seconds| Input tokens: 7,672 | Output tokens: 1,408]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 3 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Close the system search overlay
  time.sleep(0.5)
  pyautogui.press('esc')
  time.sleep(0.5)
 ───────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 3: Duration 7.60 seconds| Input tokens: 14,562 | Output tokens: 1,776]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 4 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Click inside the terminal window to focus
  pyautogui.click(x=700, y=450)
  time.sleep(0.5)
 ───────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 4: Duration 6.80 seconds| Input tokens: 21,696 | Output tokens: 2,187]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 5 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Type the removal command and execute it
  pyautogui.typewrite('jupyter kernelspec remove jupyter-kernel')
  pyautogui.press('enter')
  # Allow time for the confirmation prompt
  time.sleep(1)
```



```
Out: None
[Step 5: Duration 9.84 seconds| Input tokens: 29,051 | Output tokens: 2,754]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 6 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Type the list command and execute it
  pyautogui.typewrite('jupyter kernelspec list')
  pyautogui.press('enter')
  # Wait for output to appear
  time.sleep(1)
 ───────────────────────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 6: Duration 10.96 seconds| Input tokens: 36,677 | Output tokens: 3,399]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ Step 7 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ──────────────────────────────────────────────────────────────────────────────────
  final_answer("The 'jupyter-kernel' kernel is already removed.")
 ───────────────────────────────────────────────────────────────────────────────────────────────────────────
Out - Final answer: The 'jupyter-kernel' kernel is already removed.
[Step 7: Duration 5.27 seconds| Input tokens: 44,583 | Output tokens: 3,665]
```

Appendix A.3: Execution Trace 3

AdaBoost Notebook Task

~~~
Initializing executor, hold on...

✨ Initializing SandboxExecutor...
🧱 Docker Container Check...
🚀 Starting new sandbox VM!
📦 Creating VM container
📦 Copying VM base file to /home/stijn/Automating-DS-DE-Workflows/src/docker/sandboxes/1fe7d03d-d0d5-465b-987d-4583af499387/data.img
✅ Container started

🔍 Waiting for sshd on localhost:60000…
✅ sshd is ready
VM Started and SSH connection established!
✅ Sandbox client initialized and server healthy.

🔄 Found 1 existing kernels
🔁 Reusing existing kernel: 2f1f9cb2-a594-43ad-8a13-840d10300030
📡 WebSocket connected to kernel.
Requirement already satisfied: smolagents in ./.action-env/lib/python3.11/site-packages (1.18.0)
Requirement already satisfied: pyautogui in ./.action-env/lib/python3.11/site-packages (0.9.54)
Requirement already satisfied: huggingface-hub>=0.31.2 in ./.action-env/lib/python3.11/site-packages (from smolagents) (0.33.0)
Requirement already satisfied: requests>=2.32.3 in ./.action-env/lib/python3.11/site-packages (from smolagents) (2.32.4)
Requirement already satisfied: rich>=13.9.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (14.0.0)
Requirement already satisfied: jinja2>=3.1.4 in ./.action-env/lib/python3.11/site-packages (from smolagents) (3.1.6)
Requirement already satisfied: pillow>=10.0.1 in ./.action-env/lib/python3.11/site-packages (from smolagents) (11.2.1)
Requirement already satisfied: python-dotenv in ./.action-env/lib/python3.11/site-packages (from smolagents) (1.1.0)
Requirement already satisfied: python3-Xlib in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.15)
Requirement already satisfied: pymsgbox in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.9)
Requirement already satisfied: pytweening>=1.0.4 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.2.0)
Requirement already satisfied: pyscreeze>=0.1.21 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (1.0.1)
Requirement already satisfied: pygetwindow>=0.0.5 in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.0.9)
Requirement already satisfied: mouseinfo in ./.action-env/lib/python3.11/site-packages (from pyautogui) (0.1.3)
Requirement already satisfied: filelock in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (3.18.0)
Requirement already satisfied: fsspec>=2023.5.0 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (2025.5.1)
Requirement already satisfied: packaging>=20.9 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (25.0)
Requirement already satisfied: pyyaml>=5.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (6.0.2)
Requirement already satisfied: tqdm>=4.42.1 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.67.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (4.14.0)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.2 in ./.action-env/lib/python3.11/site-packages (from huggingface-hub>=0.31.2->smolagents) (1.1.3)
Requirement already satisfied: MarkupSafe>=2.0 in ./.action-env/lib/python3.11/site-packages (from jinja2>=3.1.4->smolagents) (3.0.2)
Requirement already satisfied: pyrect in ./.action-env/lib/python3.11/site-packages (from pygetwindow>=0.0.5->pyautogui) (0.2.0)
Requirement already satisfied: charset_normalizer<4,>=2 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.4.2)
Requirement already satisfied: idna<4,>=2.5 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2.4.0)
Requirement already satisfied: certifi>=2017.4.17 in ./.action-env/lib/python3.11/site-packages (from requests>=2.32.3->smolagents) (2025.6.15)
Requirement already satisfied: markdown-it-py>=2.2.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (3.0.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in ./.action-env/lib/python3.11/site-packages (from rich>=13.9.4->smolagents) (2.19.1)
Requirement already satisfied: mdurl~=0.1 in ./.action-env/lib/python3.11/site-packages (from markdown-it-py>=2.2.0->rich>=13.9.4->smolagents) (0.1.2)
Requirement already satisfied: pyperclip in ./.action-env/lib/python3.11/site-packages (from mouseinfo->pyautogui) (1.9.0)

✅ Sandbox Ready ✅
📤 Uploading file to VM: /home/stijn/Automating-DS-DE-Workflows/src/benchmark/evaluation_examples/examples/jupyter/1fe7d03d-d0d5-465b-987d-4583af499387/AdaBo
✅ Script executed successfully

 You are an expert autonomous agent in a sandboxed GUI environment. Your goal is to solve the given task by breaking it down into a sequence of steps.

 ### Core Workflow

 For each step, follow this process:
 1.  Think: First, explain your plan for the current step. State your goal and justify your chosen method.
 2.  Act: Provide the complete Python code to execute your plan.

 You will receive an observation (a screenshot for GUI actions or terminal output for code execution) after each action. You must analyze it to verify the ou

 ---

 ### Available Methods

 You must choose the best method for each step, either do a GUI action or use Python code to complete a action step.

 - GUI Action: Use pyautogui for interacting with graphical elements.
     - Best for: Clicking buttons, navigating menus, and interacting with applications that have no direct code interface.
     - Crucial Rule for GUI: Perform only one or a very small set of closely related GUI actions per step. After each set of GUI actions, you must wait for t
 independent GUI actions without intermediate visual verification.
     - Forbidden Action: You cannot use pyautogui.locateCenterOnScreen or any other image-finding functions in pyautogui. You must determine coordinates by a
     - Requirement: Always add a time.sleep() after actions that require loading time to ensure the UI is ready.

 - Direct Code Execution: Directly run Python code for logic, file manipulation, and data processing and other actions.
     - Best for: Calculations, reading/writing files, making API calls, or any task not requiring direct GUI interaction.
     - Requirement: Always check the output or logs in the subsequent observation to confirm success and handle any errors. When you think you are done call 
     - Tip: Your Jupyter kernel is actived from ~/Desktop so keep in mind that you start from there.
     - You will be provided with all currently installed Python packages, this so you know what packages you can use. It is possible to install new packages,

 ---

 ### Task Completion and Submission

 Once the task is fully accomplished and verified, you must conclude the mission by calling the `final_answer` tool. This is the final and most critical step

 Function Signature:
 `final_answer(summary: str)`
 - The `summary` must be a short string literal summarizing your accomplishment. Do not pass variables.

 Execution Mandate:
 The `final_answer()` call must be executed in a standalone code block. No other code, comments, or commands can be in the same execution step.

 ---

 Correct Usage Example:

 You will submit a code block containing ONLY the final answer call.
 ```python
 final_answer("I successfully downloaded the report and extracted the key figures into 'results.csv'.")
 ```

 ---
~~~

~~~
Incorrect Usage Examples (THESE WILL CAUSE AN ERROR):

- Do not include other commands or print statements:
    ```python
    # Incorrect
    print("Task is complete, submitting final answer.")
    final_answer("I created the file.")
    ~~~

- Do not combine it on a single line:
    ```python
    # Incorrect
    import os; final_answer("I listed the files.")
    ```

- Do not define variables or perform other logic in the same block:
    ```python
    # Incorrect
    summary_text = "The plot was generated and saved as 'plot.png'."
    final_answer(summary_text)
    ```

Think before you act, so check if you really need a GUI action step or if you can perform the task by just running Python code and looking at the logs and o
TASK TO COMPLETE:
<task>
Help me tune the AdaBoost Classifier in a VS Code notebook to achieve a 1.0 accuracy score on the famous Iris Dataset. Add codes only to the existing cells,
Here is a step-by-step tutorial from an expert instructing you how to complete it:
This task requires you to tune the AdaBoost classifier to achieve a 1.0 accuracy. Follow the steps:
1. First, locate the correct code cell to edit. Scroll through the notebook to find the section marked with the header **"Building the AdaBoost Model"**.
2. In the code cell immediately following that header, find the line where the `AdaBoostClassifier` is defined. It will look similar to `abc = AdaBoostClass
3. Modify the hyperparameters in that line. Change the value of `n_estimators` from `1` to `40`.
4. In the same line, change the value of `learning_rate` from `0.1` to `0.8`. The updated line should now be: `abc = AdaBoostClassifier(n_estimators=40, lea
5. After editing the code, execute the entire notebook to apply the changes and see the new accuracy. To do this, click the **"Run All"** icon (looks like a
6. Wait for all the cells to finish running. The output of the final cell will now display an "Accuracy score" of `1.0`.
7. Finally, save the updated notebook by pressing the keyboard shortcut `Ctrl+S`.
You can follow the detailed GUI plan above using `pyautogui` or proactively tackle the task using Python code
</task>

LiteLLMModel - openai/o4-mini-2025-04-16

Executing parsed code:

```python
import glob
notebooks = glob.glob("*.ipynb")
print(notebooks)
```

Execution logs:

```
['AdaBoost.ipynb']

Out: None
[Step 1: Duration 14.38 seconds| Input tokens: 3,053 | Output tokens: 858]
```

Executing parsed code:

```python
import pyautogui
import time
# Scroll down to find the AdaBoost model cell
pyautogui.scroll(-1000)
time.sleep(1)
```

```
Out: None
[Step 2: Duration 6.36 seconds| Input tokens: 8,030 | Output tokens: 1,213]
```

Executing parsed code:

```python
import pyautogui
import time
# Scroll down further to locate the model building section
pyautogui.scroll(-1000)
time.sleep(1)
```

```
Out: None
[Step 3: Duration 5.69 seconds| Input tokens: 14,954 | Output tokens: 1,562]
```

Executing parsed code:

```python
import nbformat

# Load the notebook
nb = nbformat.read('AdaBoost.ipynb', as_version=4)

# Find the section and update hyperparameters
for idx, cell in enumerate(nb.cells):
    if cell.cell_type == 'markdown' and 'Building the AdaBoost Model' in cell.source:
        # The next cell should be the code defining AdaBoostClassifier
        code_cell = nb.cells[idx + 1]
        old = 'AdaBoostClassifier(n_estimators=1, learning_rate=0.1)'
        new = 'AdaBoostClassifier(n_estimators=40, learning_rate=0.8)'
        code_cell.source = code_cell.source.replace(old, new)
        break

# Write changes back to the notebook
nbformat.write(nb, 'AdaBoost.ipynb')
print("Notebook hyperparameters updated.")
```

Execution logs:

```
Notebook hyperparameters updated.

Out: None
[Step 4: Duration 7.20 seconds| Input tokens: 22,098 | Output tokens: 2,210]
```

Executing parsed code:

```python
import pyautogui
import time
# Click the 'Run All' icon (double-play) in the notebook toolbar
pyautogui.click(x=750, y=75)
time.sleep(10)  # Wait for all cells to execute
```

```
Out: None
[Step 5: Duration 16.88 seconds| Input tokens: 29,790 | Output tokens: 2,787]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Click the 'Run All' button in the notebook toolbar
  pyautogui.click(x=910, y=75)
  time.sleep(10)  # Wait for all cells to finish running
──────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 6: Duration 17.43 seconds| Input tokens: 37,754 | Output tokens: 3,352]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Click the 'Run All' button in the notebook toolbar
  pyautogui.click(x=840, y=75)
  time.sleep(15)  # Wait for all cells to finish running
──────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 7: Duration 26.84 seconds| Input tokens: 45,976 | Output tokens: 4,378]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Click the 'Run All' button (play icon) in the notebook toolbar
  pyautogui.click(x=730, y=75)
  time.sleep(15)  # Allow time for the notebook to finish running
──────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 8: Duration 21.41 seconds| Input tokens: 54,469 | Output tokens: 4,891]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Save the notebook to apply external edits
  pyautogui.hotkey('ctrl', 's')
  time.sleep(1)  # Wait for save to complete
──────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 9: Duration 11.56 seconds| Input tokens: 63,237 | Output tokens: 6,027]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Click the 'Run All' button in the notebook toolbar
  pyautogui.click(x=800, y=80)
  time.sleep(15)  # Wait for all cells to finish running
──────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 10: Duration 19.36 seconds| Input tokens: 72,250 | Output tokens: 6,402]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import nbformat
  from nbconvert.preprocessors import ExecutePreprocessor

  # Load the notebook
  nb = nbformat.read('AdaBoost.ipynb', as_version=4)
  # Execute all cells
  ep = ExecutePreprocessor(timeout=600, kernel_name='python3')
  _ep, _ = ep.preprocess(nb, {'metadata': {'path': '.'}})
  # Save the executed notebook
  nbformat.write(nb, 'AdaBoost.ipynb')
  print("Notebook executed and saved.")
──────────────────────────────────────────────────────────────────────────────────────
Execution logs:
Notebook executed and saved.

Out: None
[Step 11: Duration 13.15 seconds| Input tokens: 81,523 | Output tokens: 6,967]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Save the updated notebook in VS Code
  tyautogui.hotkey('ctrl', 's')
  time.sleep(1)  # Wait for save to complete
──────────────────────────────────────────────────────────────────────────────────────
[31m---------------------------------------------------------------------------[39m
[31mNameError[39m                                 Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[26\][39m[32m, line 4[39m
[32m      2[39m [38;5;28;01mimport[39;00m[38;5;250m [39m[34;01mtime[39;00m
[32m      3[39m [38;5;66;03m# Save the updated notebook in VS Code[39;00m
[32m----> [39m[32m4[39m [43mtyautogui[49m.hotkey([33m'[39m[33mctrl[39m[33m'[39m, [33m'[39m[33ms[39m[33m'[39m)
[32m      5[39m time.sleep([32m1[39m)  [38;5;66;03m# Wait for save to complete[39;00m

[31mNameError[39m: name 'tyautogui' is not defined
Code execution failed: [31m---------------------------------------------------------------------------[39m
[31mNameError[39m                                 Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[26\][39m[32m, line 4[39m
[32m      2[39m [38;5;28;01mimport[39;00m[38;5;250m [39m[34;01mtime[39;00m
[32m      3[39m [38;5;66;03m# Save the updated notebook in VS Code[39;00m
[32m----> [39m[32m4[39m [43mtyautogui[49m.hotkey([33m'[39m[33mctrl[39m[33m'[39m, [33m'[39m[33ms[39m[33m'[39m)
[32m      5[39m time.sleep([32m1[39m)  [38;5;66;03m# Wait for save to complete[39;00m

[31mNameError[39m: name 'tyautogui' is not defined
[31m---------------------------------------------------------------------------[39m
[31mNameError[39m                                 Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[26\][39m[32m, line 4[39m
[32m      2[39m [38;5;28;01mimport[39;00m[38;5;250m [39m[34;01mtime[39;00m
[32m      3[39m [38;5;66;03m# Save the updated notebook in VS Code[39;00m
[32m----> [39m[32m4[39m [43mtyautogui[49m.hotkey([33m'[39m[33mctrl[39m[33m'[39m, [33m'[39m[33ms[39m[33m'[39m)
[32m      5[39m time.sleep([32m1[39m)  [38;5;66;03m# Wait for save to complete[39;00m

[31mNameError[39m: name 'tyautogui' is not defined
[Step 12: Duration 7.66 seconds| Input tokens: 91,185 | Output tokens: 7,415]
```

```
─ Executing parsed code: ─────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Save the notebook in VS Code
  pyautogui.hotkey('ctrl', 's')
  time.sleep(1)  # Wait for save to complete
 ─────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 13: Duration 6.75 seconds| Input tokens: 101,471 | Output tokens: 7,847]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Save the notebook in VS Code
  pyautogui.hotkey('ctrl', 's')
  time.sleep(1)  # Ensure the save completes
 ─────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 14: Duration 8.91 seconds| Input tokens: 112,003 | Output tokens: 8,534]
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
 ─ Executing parsed code: ────────────────────────────────────────────────────────────────
  import pyautogui
  import time
  # Save the notebook to persist changes
  pyautogui.hotkey('ctrl', 's')
  time.sleep(1)  # Wait to ensure save completes
 ─────────────────────────────────────────────────────────────────────────────────────────
Out: None
[Step 15: Duration 8.43 seconds| Input tokens: 122,779 | Output tokens: 9,221]
Reached max steps.
[Step 16: Duration 4.92 seconds| Input tokens: 133,259 | Output tokens: 9,659]
```

Appendix C Comparative Analysis of Frameworks

Table 5: Comparative Analysis of Frameworks based on Developer Experience (Dev-X) and Productivity (Dev-P) Themes, adapted from Razzaq et al.

Dev-XP Theme	Dev-X Factor / Practice	Spider2-V (Baseline)	CodeAgent (Proposed Framework)
1. Code Quality	Code Complexity	The monolithic architecture with custom inference code and intertwined logic creates a high cognitive load and is difficult to maintain.	A modular design with minimal abstractions and a significantly smaller Python codebase (≈4.5 kLoC) simplifies understanding and modification.
	Architectural Modelling	Follows a monolithic pattern, which can lead to high coupling, technical debt, and challenges in component evolution.	Employs a containerized, decoupled architecture (*Docker*, *Qemu*, *SSH*) that isolates components and improves maintainability and reusability.
2. Usability & Tool Support	Standardization & Consistency	Ad-hoc scripts and varied interfaces across tasks hinder reproducibility and create a steep learning curve for new developers.	Leverages a stack of well-documented, open-source tools & frameworks (*Smolagents*, *Docker*, *Paramiko*, *Jupyter Kernel Gateway*) that benefit from broad community consensus. This reliance on open standards creates a predictable and consistent developer experience.
	Technology Upgradability	Hardwired interfaces and custom abstractions make integration of new tools or models a significant engineering effort.	A plugin-based architecture allows new tools, models, and services to be integrated as self-contained modules with minimal friction.
3. Developer Flow	Work Fragmentation	Interdependencies between the agent, evaluation logic, and environment require frequent manual resets and context switching.	Decoupled components (*Agent*, *Tools*, *Sandbox*) are hot-swappable, allowing developers to iterate on a single component without disrupting others.
	Developer Concentration	Debugging requires manually inspecting VM snapshots, a process that is slow, non-repeatable, and breaks concentration.	Rich logging and telemetry enable targeted debugging. Furthermore, the sandbox GUI is directly accessible via any web browser (using *noVNC*), allowing for real-time observation without breaking developer flow.
	Environment Accessibility	Requires manual installation and configuration of specific virtualization software (e.g., VMWare), adding setup overhead.	The containerized sandbox is self-contained and universally accessible through any standard web browser, eliminating external software dependencies.
4. Developer Information Needs	Situational Awareness (Metrics)	Performance metrics are absent during runs. Results are only exposed post-hoc via static reports, offering limited insight.	Real-time metrics are exported via *OpenTelemetry*, providing step-level observability into agent behavior and resource usage.
	Mental Model Support	The ‘black box’ nature of the agent’s reasoning process makes it difficult to understand its decisions and failure modes.	Modular callbacks, rich logging, and traceable workflows make agent behavior transparent, inspectable, and easier to comprehend.

Appendix D Use of Generative AI in Thesis Preparation

In accordance with academic guidelines for transparency, this section details how a large language model-based Generative AI (GenAI) was utilized as an assistive tool during the research and writing process for this thesis. The AI's role was strictly that of a productivity and refinement tool, with all core ideas, analysis, and final conclusions originating from the author.

The usage can be categorized into three main areas:

1. Writing and Style Refinement. The GenAI was frequently used as an advanced grammar and style checker. Drafted paragraphs or sentences were submitted to the model with prompts such as, 'Improve the flow and formality of this text' or 'Rephrase this more concisely.' The goal of this process was to enhance readability and ensure the language met a professional, academic standard. All suggestions provided by the AI were critically reviewed, edited, and adapted by the author to ensure they accurately reflected the intended meaning.

2. Conceptual Brainstorming and Structuring. The model served as a conceptual 'sparring partner' for organizing the structure of the thesis. It was prompted to brainstorm alternative ways to present data and structure complex sections like the 'Results' and 'Discussion'. For example, the AI provided suggestions on how to organize the comparative analysis tables and offered different visualization strategies for the quantitative data. This process helped in clarifying ideas and selecting the most effective presentation format for the findings.

3. Technical Formatting and Code Assistance. A significant use of the GenAI was for technical formatting and coding support. This included:

- **LaTeX Formatting:** Generating complex LaTeX code for tables, particularly those requiring packages like 'booktabs', 'multirow', and 'tabularx' to ensure professional, readable, and correctly aligned layouts.
- **Python Scripting:** Assisting in the development and refinement of Python scripts using 'pandas', 'matplotlib', and 'seaborn'. This involved debugging code, suggesting more efficient 'pandas' operations, and providing code to improve the aesthetic quality and clarity of the plots presented in this thesis.

Ultimately, the author retains full responsibility for the content, analysis, and conclusions presented in this work. The GenAI was a tool used to augment the author's workflow, not to generate original academic insights.