DSAgentBench: Can Agents Automate End-to-End Data-Science Workflows in Real Computer Environments?

Anonymous ACL submission

Abstract

Real-world data science involves long-horizon workflows that span data wrangling, exploration, modeling, visualization, and validation, and require coordinated use of tools such as notebooks, IDEs, terminals, browsers, and databases within real operating environments. Yet existing benchmarks lack real-computer interaction and do not evaluate whether agents can execute complete end-to-end data-science workflows in realistic computing environments, failing to capture the multi-stage, multi-tool nature of data-science practice. We introduce DSAgentBench, the first benchmark to evaluate whether agents can automate full data-science workflows inside real computer environments. DSAgentBench contains 275 diverse tasks covering the entire data-science life-cycle, reflecting the complexity and tool coordination required in practice. Each task requires grounding decisions in intermediate outputs and coordinated tool use, and includes a deterministic evaluator that verifies analytical correctness, visual outputs, and model performance rather than code-only execution. Our extensive experiments with 15 closed- and open-source models show that even the strongest agent, Claude-4.6-Sonnet, achieves only 56.70% task success, while all open-source agents remain below 1%, frequently failing at tool orchestration, OS grounding, and multi-step reasoning. These results reveal a substantial capability gap between current agentic systems and real data-science workflows, positioning DSAgentBench as a foundation for developing grounded, verifiable, autonomous data-science agents. We release DSAgentBench at <redacted>.

1 Introduction

Data science drives critical decisions across industries by transforming raw data into actionable insights, from fraud detection to drug discovery and climate modeling (Sarker, 2021; Adeniran et al., 2024; Donoho, 2017). Achieving such outcomes requires a broad skill set that includes programming, statistical reasoning, visualization literacy, domain knowledge, machine learning, and effective communication (Cao, 2017). Crucially, real-world data science practice extends far beyond generating correct code: analysts must inspect heterogeneous datasets, scrape information from web sources, manage dependencies, execute and debug scripts, generate visualizations, and iteratively refine predictive models (Dale, 2022; Wickham et al., 2023). These workflows span multiple tools (Zhang et al., 2020; Hong et al., 2025), including database s, notebooks, IDEs, terminals, and web interfaces, and demand continuous analytical reasoning within computing environments (Figure 1).

Figure 1: Example agent workflow task in DSAgentBench, showing selected stages from a multi-step workflow: the agent retrieves data, executes code, and produces the final outputs. The illustrated trajectory is generated by GPT-4o.

Recent advances in large language models have shown promise in automating portions of the data science workflow (Rahman et al., 2025a; Jiang

et al., 2024). Models can convert natural language instructions into executable Python code, infer dataset schemas, generate statistical summaries, and propose analysis strategies (Liu et al., 2023; Lei et al., 2024; Chen et al., 2024). However, these abilities address only a limited subset of real-world data science practice. Producing syntactically and functionally correct code is fundamentally different from operating autonomously within real computing environments, which requires navigating file systems, coordinating tools, interpreting errors, and refining analyses based on intermediate outputs (Huang et al., 2024). Industry reports reinforce this gap, noting that current AI systems struggle to reliably execute multi-step, cross-tool analytical workflows despite strong performance on isolated tasks (OpenAI, 2024). This gap raises a central question: *Can AI agents perform long-horizon reasoning to autonomously execute end-to-end data-science workflows in real computer environments?*

Vision–language–model agents offer a promising path toward such autonomy by enabling interaction with computer environments through visual observations and GUI-based actions (Wang et al., 2024a; Tang et al., 2025b). Benchmarks such as OSWorld (Xie et al., 2024), WebArena (Zhou et al., 2023), VisualWebArena (Koh et al., 2024), ScreenSpot-Pro (Li et al., 2025), and DashboardQA (Kartha et al., 2025) show that agents can control applications and execute multi-step tasks across web and mobile environments. However, these benchmarks primarily evaluate general computer interaction, not whether agents can reason through and execute end-to-end data-science workflows.

Conversely, existing data-science benchmarks evaluate analytical capabilities without real system interaction. Most benchmarks, including DS-1000 (Lai et al., 2023), DABStep (Egg et al., 2025), MLAgentBench (Huang et al., 2023), DSBench (Jing et al., 2024), and DSEval (Zhang et al., 2024) evaluate code correctness through static execution, without requiring agents to launch applications, navigate file systems, manage dependencies, or interact with terminals and IDEs. DA-CODE (Huang et al., 2024) moves closer to realistic workflows by evaluating task planning and multi-file analysis, but remains confined to a sandboxed notebook environment without operating-system access or cross-tool coordination. Consequently, existing benchmarks test whether code runs in isolation, not whether agents can autonomously operate systems and refine analyses through real tool interaction. Moreover, existing benchmarks evaluate isolated skills, not end-to-end data-science workflows from data acquisition to validation (Table 1).

To address this gap, we propose DSAgentBench, a data-science environment that enables agents to perform real analytical workflows inside a functioning operating system. We leverage the OSWorld framework (Xie et al., 2024) and augment it with tools essential for data science, including Jupyter Notebook, automated access to external data sources (Kaggle API, OpenML), and SQLite databases. The environment supports task-specific configuration that automatically prepares datasets and enables agents to carry out end-to-end data-science workflows across multiple tools, from data acquisition to analysis and visualization. Using this environment, we construct 275 long-horizon, human-authored tasks comparable in scale to existing OSWorld-level benchmarks (Xie et al., 2024), each paired with a deterministic evaluator that verifies analytical correctness rather than surface-level code execution. Our evaluation shows that even the strongest agent, Claude-4.6-Sonnet, achieves only 56.70% task success, while all open-source agents remain below 1%, exposing major limitations in grounding, tool orchestration, and long-horizon reasoning.

In summary, our contributions are: (*i*) **DSAgentBench**, the first benchmark for evaluating autonomous data-science workflows inside real operating systems, covering the full data-science lifecycle. (*ii*) **An extension of OSWorld** that enables interaction with core data-science tools and external data sources (e.g., Kaggle, OpenML, SQLite). (*iii*) **Deterministic, execution-based evaluation** that assesses analytical correctness, visualization outputs, and model performance beyond code-only execution, along with an extensive evaluation of 15 open- and closed-source agents. (*iv*) **In-depth analysis and ablations** that identify key limitations in grounding, planning, and analytical reasoning and outline directions for future agent development.

2 Related Work

Data Science Benchmarks and Agents. Early program synthesis benchmarks such as HumanEval (Chen, 2021) evaluate functional code correctness using unit tests and contain no data analysis tasks. More recent data-science benchmarks capture richer analytical capabilities but remain limited to isolated stages of the workflow. For

Benchmark	Full OS Interaction	Web Tools & Browsers	Terminal + GUI Control	Cross-App Usage	Intermediate State	Controlled Exec Env	Multimodal Support	Data-Science Tasks	Visualization Evaluation	Environment Scalability
HumanEval (Chen, 2021)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
KRAMABench (Lai et al., 2025)	✗	✗	✗	✗	✓	code	✗	▲	✗	✗
DS-1000 (Lai et al., 2023)	✗	✗	✗	✗	✗	✗	✗	▲	✗	✗
DABStep (Egg et al., 2025)	✗	✗	✗	✗	✓	code	✗	▲	✗	✗
MLAgentBench (Huang et al., 2023)	✗	✗	✗	✗	✗	✗	✗	▲	✗	✗
DSBench (Jing et al., 2024)	✗	✗	✗	✗	✗	✗	✓	▲	✗	✗
DSEval (Zhang et al., 2024)	✗	✗	✗	✗	✗	✓	✗	▲	✗	✗
ARCADE (Yin et al., 2023)	✗	✗	✗	✗	✗	✗	✗	▲	✗	✗
DA-CODE (Huang et al., 2024)	✗	✗	✗	✗	✗	code	✗	▲	▲	▲
OSWORLD (Xie et al., 2024)	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓
DSAgentBench (Ours)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: **Comparison of DSAgentBench with existing benchmarks across unified agent and data-science capabilities.** ✓ = Yes, ▲ = Partial, ✗ = No.

example, MLAgentBench (Huang et al., 2023), DABStep (Egg et al., 2025), DSBench (Jing et al., 2024), DS-1000 (Lai et al., 2023), and DSEval (Zhang et al., 2024) focus on subsets of data manipulation, modeling, or reasoning, while visualization benchmarks such as ChartQA (Masry et al., 2022), Text2Vis (Rahman et al., 2025b), and VisEval (Chen et al., 2024) evaluate chart generation and visual reasoning without involving data wrangling or iterative analysis. Benchmarks targeting more complex pipelines, including DA-CODE (Huang et al., 2024), KRAMABench (Lai et al., 2025), and ARCADE (Yin et al., 2023), still operate in static, sandboxed environments without operating-system interaction. Consequently, existing benchmarks test code generation or isolated reasoning rather than autonomous execution of end-to-end data-science workflows. DSAgentBench fills this gap by enabling agents to execute complete data-science workflows inside a functioning operating system, requiring coordinated reasoning, tool orchestration, and environment interaction (Tab. 1).

Recent work has also explored large language models as agents for data science by generating code (Qiao et al., 2023; Guo et al., 2024; Li et al., 2024) or managing relevant context across tasks (Hong et al., 2025). However, these approaches are typically evaluated through case studies or controlled settings, making it difficult to assess robustness, long-horizon reasoning, and tool coordination in realistic computer environments.

GUI and Computer-Control Agents Vision-language agents have enabled autonomous computer control through multimodal observation and GUI interaction. Early work focused on web environments: WebShop (Yao et al., 2022) and Mind2Web (Deng et al., 2023) evaluated navigation and instruction following on simulated and real websites, while WebArena (Zhou et al., 2023) and VisualWebArena (Koh et al., 2024) introduced realistic web applications requiring visual grounding and multi-step planning. OSWorld (Xie et al., 2024) extended this line of work to full desktop control across Ubuntu, Windows, and macOS, where agents operate applications using keyboard and mouse actions guided by screenshots and accessibility trees. Additional benchmarks target mobile interfaces (AndroidWorld (Rawles et al., 2024)), office productivity (OfficeBench (Wang et al., 2024b)), and UI grounding (ScreenSpot-Pro (Li et al., 2025)). Recent work further improves desktop grounding through instruction-tuned models (CogAgent (Hong et al., 2024), ShowUI (Lin et al., 2025), Ferret-UI (You et al., 2024)), grounding benchmarks and agents (OS-ATLAS (Wu et al., 2024), GroundCUA (Feizi et al., 2025), JEDI (Xie et al., 2025b)), and reinforcement-learning-based agents (GUI-R1 (Luo et al., 2025), GUI-G2 (Tang et al., 2025a), InfiGUI-G1 (Liu et al., 2025)).

These benchmarks establish core infrastructure for OS-level autonomy but primarily evaluate general computing operations, such as application use and interface navigation, rather than analytical reasoning. They assess whether agents can operate computers rather than function as data scientists who load data, analyze results, train models, visualize findings, and debug workflows.

3 DSAgentBench Benchmark

This section introduces DSAgentBench, a diverse set of problems, and describes its data collection and annotation pipeline.

3.1 Problem Formulation

DSAgentBench frames data science tasks as an autonomous, long-horizon data-driven decision-making problem in real computing environments. Formally, the benchmark consists of a set of tasks $\mathcal{T} = \{(\mathcal{C}_i, \mathcal{I}_i, \mathcal{V}_i)\}_{i=1}^{N}$, where $\mathcal{C}_i$ is a task configuration defining the initial system setup, $\mathcal{I}_i$ is a natural language instruction describing the analytical objective, and $\mathcal{V}_i$ is a deterministic Python evaluator.

The task configuration $\mathcal{C}_i$ defines the initialized operating-system state, including available datasets, file system structure, installed libraries, and appli-

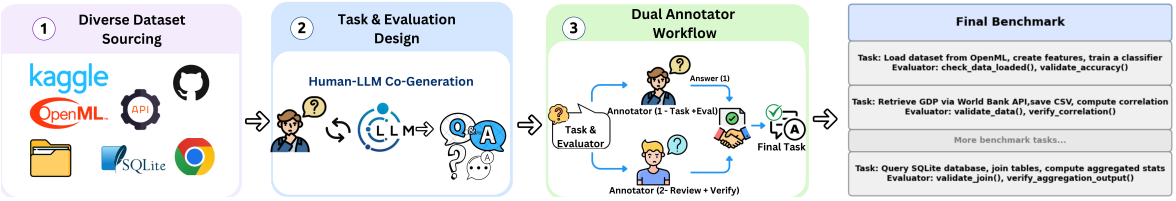

Figure 2: Benchmark Construction Pipeline: We source heterogeneous real-world datasets (1); design tasks and evaluators through human–LLM collaboration with pre- and post-execution configurations (2); and apply dual-annotator review and verification (3). The result is a curated benchmark of reproducible tasks with complete configurations and deterministic evaluation scripts.

cations (e.g., IDEs, terminals, browsers), as well as optional initialization or cleanup procedures. An agent interacts with the environment through multimodal observations and actions to execute an end-to-end data-science workflow. DSAgentBench evaluates final outcomes rather than prescribing a fixed workflow path: agents may use any available applications as long as outputs satisfy the deterministic evaluator. A task is successful if and only if the agent's outputs satisfy $\mathcal{V}_i$.

3.2 Dataset Construction

As shown in Figure 2, DSAgentBench was built through a rigorously controlled, three-stage process: (*i*) sourcing datasets from diverse real-world platforms, (*ii*) collaborative task and evaluator design, and (*iii*) dual-annotator verification, where each task was independently executed and validated for accuracy and reproducibility.

3.2.1 Dataset Sourcing

We collected data from diverse real-world sources to ensure agents encounter the variety of formats typical in practical data science. We primarily collected structured tabular data, supplemented by image and text data, and exhibit varying structural complexity, ranging from flat single-table and multi-table CSVs to normalized relational schemas requiring multi-table joins. These datasets are drawn from Kaggle competition files (prioritizing top-downloaded and highly rated examples), popular OpenML datasets used in academic evaluation, SQLite databases reflecting production relational storage, GitHub repositories such as Plotly datasets, and web-accessible data retrieved through API.

3.2.2 Task and Evaluator Design

To ensure that DSAgentBench reflects real-world data science practice, we grounded its design in both prior literature and empirical analysis. Prior studies show that Python-based environments such as Jupyter notebooks, VS Code, spreadsheets, GitHub, and terminals form the core of everyday data science workflows (Zhang et al., 2020). Building on this foundation, we manually analyzed 100 high-ranking Kaggle notebooks from popular and historically influential competitions to identify recurring analytical workflows and question types. Two annotators with over five years of data-science experience independently derived an initial task taxonomy, which we then expanded using LLMs to surface underrepresented patterns and ensure comprehensive coverage of the data-science lifecycle.

Based on this taxonomy, DSAgentBench organizes tasks into six capability categories aligned with key stages of the data-science workflow (Table 5). **Data acquisition** tasks assess an agent's ability to locate, retrieve, and load data from heterogeneous sources and formats. **Exploratory data analysis** tasks evaluate initial sense-making through statistical summaries, filtering, and pattern discovery (e.g., outlier detection and correlation analysis). **Feature engineering** tasks test data transformation and representation, including feature construction, encoding, scaling, and dimensionality reduction. **Modeling** tasks focus on training and comparing predictive models under realistic constraints such as data imbalance and metric selection. **Evaluation and deployment** tasks measure statistical rigor via validation strategies, hyperparameter tuning, and significance testing. Finally, **visualization and reporting** tasks assess an agent's ability to communicate insights through correctly grounded visualizations, interactive plot inspection, structured analytical reports, and PowerPoint reporting artifacts (see App. A.1 for details).

Guided by this taxonomy, four expert annotators collaboratively developed 275 tasks over a three-month period (approximately 400 working hours). Each task includes: (i) a natural language instruction, (ii) an executable environment configuration, and (iii) a deterministic Python evaluation function for automatic verification. Tasks were authored by experts using a curated taxonomy; Vision-language models (GPT-5, Claude 4.5 Sonnet, and Gemini 3) were used only to refine wording and identify edge cases, while all task logic, expected outputs,

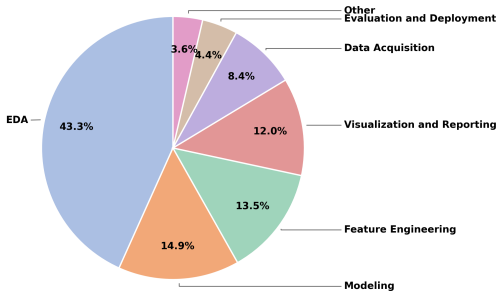

Figure 3: Task category distribution in our benchmark, reflecting real-world data science practice.

and evaluators were defined and validated by humans. This human-in-the-loop process ensures that tasks reflect realistic analytical reasoning rather than automatically generated QA pairs. Evaluators score semantic correctness of the final outputs (not just successful code execution), enabling rigorous, execution-based benchmarking.

3.2.3 Dual Annotator Verification

Each task underwent structured dual review for quality assurance. One human annotator (the "creator") developed the task specification and evaluation logic, while a second annotator (the "verifier") independently assessed instruction clarity, verified executability by running the task with a baseline agent, and validated evaluator correctness. Initial agreement on task quality defined as independent approval by both annotators that the task and evaluator were correct and required no revision was 86%, with the remaining tasks revised through iterative discussion to refine instructions, fix evaluation logic, or adjust environment configurations. All 275 tasks ultimately achieved mutual approval, ensuring every benchmark entry is reproducible, technically correct, and fully executable.

3.3 Dataset Diversity & Statistics

DSAgentBench is designed to stress-test agent robustness under realistic data-science conditions by varying task structure, difficulty, data modality, source, and tool requirements (Table 2). Task categories are well distributed across the data-science lifecycle (Figure 3), with exploratory data analysis (EDA) comprising the largest share (43.3%). This distribution aligns with industry observations that data scientists spend 40-60% of their time on data preparation and exploration (Anaconda, 2025).

The benchmark emphasizes challenging, long-horizon reasoning. Nearly half of the tasks are hard (47.6%), with 46.9% medium and only 5.5% easy. Difficulty levels are assigned based on multiple factors, including (i) the number of required steps (easy: 1–2; medium: 3–4; hard: 5+), (ii) analytical sophistication (e.g., simple filtering versus model building and comparison), and (iii) tool coordination requirements (single versus multiple tools). Two expert annotators independently labeled task difficulty, resolving disagreements through discussion. Moreover, 56.7% of tasks involve multi-stage workflows—such as load–transform–visualize or model–evaluate–refine—requiring agents to iteratively execute code, inspect intermediate outputs, and adapt later actions, mirroring real-world data science, with tasks averaging 4–5 analytical steps.

Dimension	Distribution Breakdown
Complexity	Hard: 47.6% | Medium: 46.9% | Easy: 5.5%
Stage Type	Multi-Stage: 56.7% | Single-Stage: 43.3%
Modality	Tabular: 95.3% | Image: 3.6% | Text: 1.1%
Source	GitHub: 37.1% | Kaggle: 29.8% | OpenML: 18.9% | SQLite: 7.6% | Web: 6.5%
Tools Used	Python: 100.0% | VS Code: 81.1% | Jupyter Notebook: 18.9% | Chrome: 10.2%

Table 2: Key characteristics of DSAgentBench.

Data and source diversity further increase task complexity. While tabular data dominates (95.3%), reflecting real-world structured workflows, the benchmark also includes image (3.6%) and text (1.1%) components, introducing multimodal reasoning challenges; task design remains modality-agnostic. Datasets are drawn from a wide range of real-world platforms, including GitHub (37.1%), Kaggle (29.8%), OpenML (18.9%), and SQLite databases (7.6%). Finally, tasks run in real operating systems and require coordinated use of multiple tools—typically Python alongside VS Code, Jupyter Notebook, and Chrome—often involving three or more tools per task. This moves evaluation beyond static code generation to realistic workflow execution and cross-application reasoning.

4 Methodology

4.1 Environment Architecture

DSAgentBench extends OSWorld (Xie et al., 2024) to provide a realistic execution environment for end-to-end analytical workflows (Figure 4). The environment runs Ubuntu OS with Python and common data-science libraries pre-installed, and captures screenshots at a resolution of 1920×1080 to provide clear visual context. Visual Studio Code and Jupyter Notebook are available for script development and interactive analysis, and Chrome is included for accessing documentation or external data when needed. The environment supports automated data retrieval through the Kaggle API, OpenML, direct URLs, and SQLite databases.

Formally, the environment is represented as a tuple $(\mathcal{O}, \mathcal{A}, T)$, where $\mathcal{O}$ is the observation space, $\mathcal{A}$ is the action space, and T is the transition func-

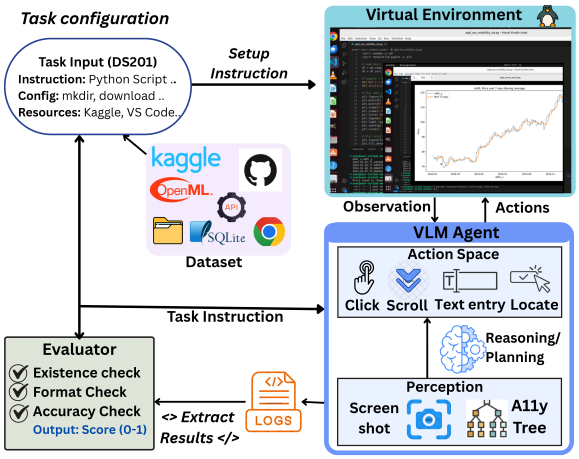

Figure 4: DSAGENTBENCH execution workflow: the agent perceives screenshots and the A11y tree, executes GUI-based actions, and is evaluated by a custom script.

tion. At each timestep t, the agent receives an observation $o_t \in \mathcal{O}$, selects an action $a_t \in \mathcal{A}$, and obtains the next observation o_{t+1} after the system transitions to state $s_{t+1} = T(s_t, a_t)$.

Observation Space: At each timestep o_t, the agent receives one of two modalities: *(i)* a 1920×1080 **Screenshot** of the desktop interface, providing pixel-level visual context; or *(ii)* an integrated **Screenshot + Accessibility Tree (A11y)** modality that augments the screenshot with structured UI metadata, including element roles, accessible names, bounding boxes, and interaction states, extracted via AT-SPI[1]. These two settings allow us to study how structured UI metadata complements visual observations for grounding in data science workflows. Observations are captured after each action with sufficient delay to ensure that windows and outputs have fully stabilized.

Action Space: Agents interact with the environment through a structured action space $\mathcal{A}$ supporting the core operations required for data-science workflows. Actions include *(i)* mouse-based GUI interactions (e.g., clicking, dragging, scrolling), *(ii)* keyboard input for text entry and shortcut-based control (e.g., navigating IDE interfaces, code execution and file operations), and *(iii)* meta actions including `WAIT` to pause execution and allow rendering, `DONE` to signal task completion, and `FAIL` to terminate execution with an error. Each action executes atomically, after which the environment updates the desktop state and returns the next observation o_{t+1}, enabling realistic human–computer interaction across diverse analytical workflows.

All agents operate through a standard GUI action space to evaluate end-to-end desktop competence without relying on privileged task-specific APIs. This ensures fair comparison while still supporting terminal, CLI, and script-based workflows via typical interactions, reflecting how data science is performed in real computing environments.

[1] https://docs.gtk.org/atspi2/

4.2 Task Execution Workflow

Each task begins with environment initialization, loading a clean virtual-machine snapshot and applying task-specific setup steps (e.g., directory creation, dataset retrieval, application launch) to produce the initial observation. The agent then follows an iterative perception–action loop: at each timestep, it receives the current observation, selects an action based on the task instruction and interaction history, executes the action in the environment, and observes the updated desktop state. Execution ends when the agent signals completion via `DONE`, explicitly aborts with `FAIL`, or reaches the maximum step budget of 15 actions (see example Figure 5). All observations, actions, and timestamps are logged throughout execution to enable reproducible evaluation, trajectory analysis, and failure diagnosis. Further details on implementation and evaluation prompts are provided in Appendix Sections B and C, respectively.

5 Evaluation

5.1 Models

We evaluate a diverse set of vision-language agents spanning closed-source, hybrid, and open-source architectures. Closed-source agents include GPT-4o (OpenAI et al., 2024), GPT-5-mini, GPT-5 (OpenAI, 2025), O4-mini (OpenAI et al., 2024), Claude Sonnet 4, 4.5 and 4.6, Gemini 2.5 Pro (Comanici et al., 2025), and OpenAI's Computer Agent (ope), all of which adopt unified architectures where a single vision-language model handles both UI grounding and action planning. Hybrid models combine open-source grounding models with proprietary planning models, including Jedi-3B and Jedi-7B paired with GPT-4o (Xie et al., 2025a). Open-source models include UI-TARS 2B and UI-TARS 7B (Qin et al., 2025), GUI-OWL-7B (Ye et al., 2025), and OpenCUA-72B (Wang et al., 2025) representing fully open architectures trained specifically for GUI interaction.

5.2 Execution-Based Evaluation and Metrics

Each task is paired with a deterministic Python evaluator executed after the agent completes its interaction sequence. After task termination, the evaluator collects relevant artifacts from the virtual machine, including scripts, generated files, visual-

Model	Screenshot							Screenshot + A11y Tree						
	DA	EDA	FE	Model	Vis	Eval	Overall	DA	EDA	FE	Model	Vis	Eval	Overall
GPT-4o	0.00	27.06	24.24	16.67	7.14	33.33	19.34	13.04	29.66	35.14	21.95	15.62	8.33	24.54
O4-mini	4.35	1.68	2.70	0.00	3.03	0.00	1.82	0.00	4.20	5.41	0.00	0.00	0.00	2.55
GPT-5-mini	8.70	17.28	25.00	3.57	21.43	0.00	15.20	4.55	24.14	27.03	7.50	21.88	0.00	19.03
GPT-5	17.39	27.41	27.03	26.83	9.09	16.67	23.63	26.08	33.44	29.72	32.22	15.15	33.33	29.81
Gemini-2.5-Pro	4.55	17.86	16.13	17.24	10.71	16.67	14.49	4.35	22.78	33.33	14.81	30.77	0.00	20.81
OpenAI CUA	0.00	10.59	12.50	0.00	14.29	0.00	8.13	4.35	9.52	6.06	2.86	6.67	0.00	6.61
Claude-4-Sonnet	0.00	7.46	4.17	0.00	3.85	0.00	4.55	0.00	4.86	0.00	1.00	11.11	0.00	4.64
Claude-4.5-Sonnet	4.76	4.17	0.00	3.45	3.23	0.00	3.17	8.33	12.31	12.50	4.55	4.17	0.00	9.21
Claude-4.6-Sonnet	**43.48**	**57.98**	**51.35**	**41.46**	**39.39**	**50.00**	**50.55**	**47.82**	**64.88**	**56.75**	**46.34**	**42.42**	**66.67**	**56.70**
Human Performance	73.91	90.76	91.89	73.17	78.79	83.33	85.09	73.91	90.76	91.89	73.17	78.79	83.33	85.09

Table 3: DSAgentBench accuracy (%) under two observation settings (Screenshot and Screenshot + A11y Tree) across data-science lifecycle tasks: Data Acquisition (DA), Exploratory Data Analysis (EDA), Feature Engineering (FE), Modeling, Visualization (Vis), and Evaluation (Eval). Results for open-source models are reported in Table 15.

Model	Screenshot							Screenshot + A11y Tree						
	Single	Multi	Easy	Medium	Hard	VS Code	Jupyter	Single	Multi	Easy	Medium	Hard	VS Code	Jupyter
GPT-4o	26.97	13.52	38.46	24.74	11.79	14.69	**39.22**	35.59	16.13	46.67	34.38	12.31	23.87	27.45
O4-mini	4.20	0.00	20.00	1.55	0.00	1.79	1.92	5.04	0.64	13.33	3.88	0.00	2.69	1.92
GPT-5-mini	24.71	8.00	38.46	24.47	9.41	15.23	11.00	31.03	9.87	66.67	27.20	5.47	14.68	38.00
GPT-5	27.73	20.51	53.33	27.13	16.79	25.10	17.30	39.49	22.43	66.67	32.55	22.89	27.35	40.38
Gemini-2.5-Pro	19.12	10.92	46.15	15.62	9.18	17.31	2.40	32.10	12.93	53.85	28.74	9.28	17.45	31.25
OpenAI CUA	13.64	4.00	23.08	13.40	1.20	9.49	2.30	10.68	3.60	15.38	10.43	1.84	5.21	12.60
Claude-4-Sonnet	6.67	2.93	8.33	4.65	4.00	5.60	0.00	8.06	2.10	25.00	6.33	0.64	4.86	3.70
Claude-4.5-Sonnet	3.16	3.17	16.67	3.51	1.05	4.14	0.00	14.49	4.82	36.36	10.00	3.28	8.74	10.20
Claude-4.6-Sonnet	**59.66**	**43.60**	**73.33**	**55.04**	**43.52**	**54.01**	35.71	**63.86**	**51.24**	**86.68**	**60.62**	**49.41**	**56.92**	**55.77**

Table 4: DSAgentBench **accuracy** (%) across stage type, task complexity, and tool usage under different settings.

izations, or trained models, and scores them against task-specific evaluation criteria defined for that task. For example, in a *correlation analysis task*, the evaluator verifies that the required output file exists, loads the computed Pearson correlation coefficient, and checks whether it matches the expected value within a tolerance of $\epsilon = 0.01$. For *visualization tasks*, the evaluator checks that a visualization is generated and verifies required metadata such as labeled axes, titles, legends, and correct data mappings. In addition, GPT-4o is used as a visual judge to assess visualization quality and semantic alignment with the task requirements. To avoid evaluation circularity, we use Gemini-2.5-Pro as the visual judge for all GPT-4o generated outputs, while GPT-4o serves as the judge for outputs from other models. Only $\sim$10% of tasks invoke an LLM-based judge, and only after deterministic validation gates. For *classification tasks*, evaluators verify that trained models meet minimum performance thresholds (e.g., accuracy or F1 ≥ 0.7). Each evaluator produces a continuous score in $[0, 1]$.

We use *task success rate* as the primary evaluation metric, defined as the percentage of tasks achieving an overall score greater than or equal to 0.95. This threshold allows minor tolerance for numerical precision or formatting variation while ensuring semantic correctness of the analytical outcome. We also report the *average evaluation score* across all tasks to provide a detailed assessment of agent performance. Representative evaluator designs are provided in §C (Tables 12 and 13).

5.3 Main Results

Table 3 presents the performance of agents across both observation settings: *Screenshot-only* and the hybrid *Screenshot + A11y Tree* configuration. As human reference points, three participants, two applied scientists and one master's graduate, achieved an overall success rate of 85.09% under the same task environment and deterministic evaluation protocol. All results use the fixed evaluation setup described in App. B.5. The results reveal a clear performance gap between current agents and the demands of real-world end-to-end data science workflows. Claude-4.6-Sonnet achieves the strongest agent performance, reaching 56.70% overall accuracy under the Screenshot + A11y Tree setting, followed by GPT-5 at 29.81%. Other closed-source models, including GPT-4o, Gemini-2.5-Pro, and GPT-5-mini, achieve lower overall accuracy of around 20%, highlighting persistent long-horizon execution challenges. In contrast, open-source agents achieve at most 1% accuracy under the Screenshot-only setting, and none support A11y Tree input (Table 15). Despite generating code and issue interface actions, these models consistently fail to ground instructions in the UI state and complete end-to-end workflows.

Performance varies substantially across task types (Table 3). Data acquisition, model validation, and evaluation tasks remain most challenging, reflecting the difficulty of long-horizon reasoning and tool coordination. Adding accessibility infor-

mation to screenshots generally improves performance, indicating that structured UI metadata aids grounding and interaction (Tables 3 and 4). However, gains vary widely across models, and some struggle to effectively exploit A11y signals. Even under the hybrid setting, overall success rates remain low, underscoring that current agents are still far from autonomous data science workflows.

5.4 Ablation Studies

To diagnose performance trends, we analyze variations across task complexities, workflow structure, and tool usage (Table 4). Tasks executed in Jupyter consistently outperform those using VS Code, largely due to fewer terminal and environment-related failures. Single-stage tasks achieve substantially higher success rates than multi-stage workflows, highlighting the difficulty of maintaining state, recovering from errors, and coordinating tools across longer execution chains. Performance also degrades monotonically with task difficulty, with hard tasks remaining the most challenging due to long-horizon reasoning and iterative refinement.

We additionally ablate the maximum interaction budget by evaluating step limits of 15, 30, and 50 (Table 11). Increasing the budget from 15 to 50 steps yields only marginal gains in task success rate (24.54% $\rightarrow$ 25.81%) and average score (0.55 $\rightarrow$ 0.57). The small gain suggests that performance is not primarily limited by the action budget; instead, failures reflect a combination of grounding, planning, reasoning, tool orchestration, and analytical execution errors. We also run a terminal-first ablation with GPT-4o in the screenshot-only setting, prompting the agent to prefer terminal or command-line workflows whenever they could solve the current subtask more directly and reliably. Relative to the standard screenshot-only setting, performance changes only marginally, from 19.34% to 20.73%, suggesting that terminal-first execution alone does not substantially reduce the benchmark difficulty.

6 Error Analysis

To better understand error patterns, we conduct a targeted analysis of 604 manually inspected runs from closed-source models using Screenshot + A11y Tree observations, along with 150 sampled runs from open-source models (Screenshot-only).

Root Causes of Failure. (Table 10) shows that DSAgentBench evaluates both environment grounding and data-science execution. Open-source agents fail almost entirely due to grounding errors (97–98%), revealing poor alignment with desktop state. Stronger closed-source agents show mixed failures: GPT-4o still fails mainly from grounding, while better-grounded models (Gemini-2.5-Pro, GPT-5, Claude-4.6-Sonnet) exhibit terminal, code, and reasoning failures. This indicates success requires both robust UI control and sound data-science reasoning in real environments.

Temporal Failure Structure and Recovery. Table 14 analyzes when failures occur using the first failure step (FF), defined as the earliest step where an error is detected. CUA and GUI-OWL-7B exhibit predominantly late-stage failures, with over 93% occurring after prolonged interaction, reflecting extended but ineffective exploration due to poor grounding. In contrast, GPT-4o and Gemini-2.5-Pro show a higher proportion of early and mid-trajectory failures, indicating weaker robustness in initial grounding and planning. Open-source models fail especially early, often unable to correctly open or control the terminal. Budget exhaustion further differentiates behavior: CUA and open-source agents almost always exhaust their step budgets, while Gemini 2.5 Pro and GPT-5-Mini terminate earlier on a substantial fraction of tasks.

Efficiency of Successful Runs. As shown in Table 9, Gemini-2.5-Pro completes tasks most efficiently (6.76 steps on average), followed by GPT-5-Mini (7.33), GPT-4o (10.03), and CUA (15.00). This reveals a clear trade-off between exploration depth and execution efficiency: models that act conservatively and terminate earlier achieve faster successes but also incur higher early-failure rates.

7 Conclusion

We present DSAgentBench, the first benchmark for assessing whether agents can automate end-to-end data-science workflows inside real operating systems. Unlike prior benchmarks that focus on isolated code generation or generic GUI interaction, our benchmark requires agents to plan and execute long-horizon, multi-tool workflows spanning data acquisition, analysis, and modeling. Our experiments reveal substantial limitations in current agents, with even the strongest systems achieving low success rates on these tasks. Through systematic evaluation and analysis, we identify key challenges in grounding, tool orchestration, and iterative analytical reasoning. Failure patterns show that DSAgentBench jointly evaluates desktop grounding and data-science reasoning via real tool use, state management, and correct artifact generation. We hope DSAgentBench accelerates progress toward more robust, and autonomous agents capable of performing real-world data science.

Limitations

First, the open-source agents in our evaluation stack do not currently support A11y Tree observations, so we evaluate them under the screenshot-only setting. However, for closed-source agents, we report both screenshot-only and screenshot+A11y Tree results, where A11y provides only modest overall gains.

Second, our detailed error analysis is based on 604 manually inspected trajectories from closed-source models and 150 from open-source models. While this constitutes a substantial qualitative evaluation, it still represents a subset of the full benchmark, and rare failure patterns may not be fully captured.

Finally, our evaluation of visualization-heavy tasks focuses on final artifact quality. The evaluators verify required outputs, data mappings, labels, and task-specific correctness, overall visual clarity and semantic alignment.

Ethical Considerations

For the DSAGENTBENCH benchmark, we applied strict data governance and licensing constraints during dataset construction. All tabular and structured datasets were collected exclusively from openly licensed platforms, specifically GitHub, Kaggle, and OpenML. We included only datasets that are either MIT-licensed or released under clearly permissive open-source or public-use licenses and explicitly suitable for research, redistribution, and modification as specified by their providers. Datasets with ambiguous, restrictive, or missing license information were systematically excluded. This filtering ensures that all components of DSAGENTBENCH comply with licensing requirements and can be safely used, shared, and extended by the research community. Finally, we used AI-based writing assistants only to improve the presentation of the paper.

References

Computer-Using Agent — openai.com. https://openai.com/index/computer-using-agent/. [Accessed 14-07-2025].

Ibrahim Adedeji Adeniran, Christianah Pelumi Efunniyi, Olajide Soji Osundare, Angela Omozele Abhulimen, and UK OneAdvanced. 2024. The role of data science in transforming business operations: Case studies from enterprises. *Computer Science & IT Research Journal*, 5(8):2026–2039.

Anaconda. 2025. 2025 state of data science: Moving from hype toward maturity. Technical report, Anaconda, Inc. Accessed: 2025-12-16.

Longbing Cao. 2017. Data science: a comprehensive overview. *ACM Computing Surveys (CSUR)*, 50(3):1–42.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Nan Chen, Yuge Zhang, Jiahang Xu, Kan Ren, and Yuqing Yang. 2024. Viseval: A benchmark for data visualization in the era of large language models. *IEEE Transactions on Visualization and Computer Graphics*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, and et al. Inderjit Dhillon. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

Kyran Dale. 2022. *Data Visualization with Python and JavaScript: Scrape, Clean, Explore, and Transform Your Data*. " O'Reilly Media, Inc.".

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

David Donoho. 2017. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26(4):745–766.

Alex Egg, Martin Iglesias Goyanes, Friso Kingma, Andreu Mora, Leandro von Werra, and Thomas Wolf. 2025. Dabstep: Data agent benchmark for multi-step reasoning. *arXiv preprint arXiv:2506.23719*.

Aarash Feizi, Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Kaixin Li, Rabiul Awal, Xing Han Lù, Johan Obando-Ceron, Juan A Rodriguez, Nicolas Chapados, and 1 others. 2025. Grounding computer use agents on human demonstrations. *arXiv preprint arXiv:2511.07332*.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024. Ds-agent: automated data science by empowering large language models with case-based reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, and 1 others. 2025. Data interpreter: An LLM agent for data science. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19796–19821.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and 1 others. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14281–14290.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and 1 others. 2024. Da-code: Agent data science code generation benchmark for large language models. *arXiv preprint arXiv:2410.07331*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. 2024. Dsbench: How far are data science agents from becoming data science experts? *arXiv preprint arXiv:2409.07703*.

Aaryaman Kartha, Ahmed Masry, Mohammed Saidul Islam, Thinh Lang, Shadikur Rahman, Ridwan Mahbub, Mizanur Rahman, Mahir Ahmed, Md Rizwan Parvez, Enamul Hoque, and 1 others. 2025. Dashboardqa: Benchmarking multimodal agents for question answering on interactive dashboards. *arXiv preprint arXiv:2508.17398*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Eugenie Lai, Gerardo Vitagliano, Ziyu Zhang, Om Chabra, Sivaprasad Sudhir, Anna Zeng, Anton A Zabreyko, Chenning Li, Ferdi Kossmann, Jialin Ding, and 1 others. 2025. Kramabench: A benchmark for ai systems on data-to-insight pipelines over data lakes. *arXiv preprint arXiv:2506.06541*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. 2025. Screenspot-pro: Gui grounding for professional high-resolution computer use. In *Proceedings of the 33rd ACM International Conference on Multimedia*, pages 8778–8786.

Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, and 1 others. 2024. Autokaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. 2025. Showui: One vision-language-action model for gui visual agent. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19498–19508.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Yuhang Liu, Zeyu Liu, Shuanghe Zhu, Pengxiang Li, Congkai Xie, Jiasheng Wang, Xueyu Hu, Xiaotian Han, Jianbo Yuan, Xinyao Wang, and 1 others. 2025. Infigui-g1: Advancing gui grounding with adaptive exploration policy optimization. *arXiv preprint arXiv:2508.05731*.

Run Luo, Lu Wang, Wanwei He, Longze Chen, Jiaming Li, and Xiaobo Xia. 2025. Gui-r1: A generalist r1-style vision-language action model for gui agents. *arXiv preprint arXiv:2504.10458*.

Ahmed Masry, Do Xuan Long, Jia Qing Tan, Shafiq Joty, and Enamul Hoque. 2022. Chartqa: A benchmark for question answering about charts with visual and logical reasoning. *arXiv preprint arXiv:2203.10244*.

OpenAI. 2024. Computer-using agent. Accessed: 2025-12-14.

OpenAI. 2025. Introducing gpt-5. Accessed: 2025-10-01.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, and 1 others. 2023. Taskweaver: A code-first agent framework. *arXiv preprint arXiv:2311.17541*.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, and 16 others. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *Preprint*, arXiv:2501.12326.

Mizanur Rahman, Amran Bhuiyan, Mohammed Saidul Islam, Md Tahmid Rahman Laskar, Ridwan Mahbub, Ahmed Masry, Shafiq Joty, and Enamul Hoque. 2025a. Llm-based data science agents: A survey of capabilities, challenges, and future directions. *arXiv preprint arXiv:2510.04023*.

Mizanur Rahman, Md Tahmid Rahman Laskar, Shafiq Joty, and Enamul Hoque. 2025b. Text2vis: A challenging and diverse benchmark for generating multimodal visualizations from text. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 31837–31862.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, and 1 others. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*.

Iqbal H Sarker. 2021. Data science and analytics: an overview from data-driven smart computing, decision-making and applications perspective. *SN Computer Science*, 2(5):377.

Fei Tang, Zhangxuan Gu, Zhengxi Lu, Xuyang Liu, Shuheng Shen, Changhua Meng, Wen Wang, Wenqi Zhang, Yongliang Shen, Weiming Lu, and 1 others. 2025a. Gui-g2: Gaussian reward modeling for gui grounding. *arXiv preprint arXiv:2507.15846*.

Fei Tang, Haolei Xu, Hang Zhang, Siqi Chen, Xingyu Wu, Yongliang Shen, Wenqi Zhang, Guiyang Hou, Zeqi Tan, Yuchen Yan, and 1 others. 2025b. A survey on (m) llm-based gui agents. *arXiv preprint arXiv:2504.13865*.

Shuai Wang, Weiwen Liu, Jingxuan Chen, Yuqi Zhou, Weinan Gan, Xingshan Zeng, Yuhan Che, Shuai Yu, Xinlong Hao, Kun Shao, and 1 others. 2024a. Gui agents with foundation models: A comprehensive survey. *arXiv preprint arXiv:2411.04890*.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, and 1 others. 2025. Opencua: Open foundations for computer-use agents. *arXiv preprint arXiv:2508.09123*.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. 2024b. Officebench: Benchmarking language agents across multiple applications for office automation. *arXiv preprint arXiv:2407.19056*.

Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for data science: import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc.".

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and 1 others. 2024. Os-atlas: A foundation action model for generalist gui agents. *arXiv preprint arXiv:2410.23218*.

Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, Yiheng Xu, Junli Wang, Doyen Sahoo, Tao Yu, and Caiming Xiong. 2025a. Scaling computer-use grounding via user interface decomposition and synthesis. *Preprint*, arXiv:2505.13227.

Tianbao Xie, Jiaqi Deng, Xiaochuan Li, Junlin Yang, Haoyuan Wu, Jixuan Chen, Wenjing Hu, Xinyuan Wang, Yuhui Xu, Zekun Wang, and 1 others. 2025b. Scaling computer-use grounding via user interface decomposition and synthesis. *arXiv preprint arXiv:2505.13227*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Jiabo Ye, Xi Zhang, Haiyang Xu, Haowei Liu, Junyang Wang, Zhaoqing Zhu, Ziwei Zheng, Feiyu Gao, Junjie Cao, Zhengxi Lu, Jitong Liao, Qi Zheng, Fei Huang, Jingren Zhou, and Ming Yan. 2025. Mobile-agent-v3: Fundamental agents for gui automation. *Preprint*, arXiv:2508.15144.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, and 1 others. 2023. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 126–173.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *European Conference on Computer Vision*, pages 240–255. Springer.

Amy X Zhang, Michael Muller, and Dakuo Wang. 2020. How do data science workers collaborate? roles, workflows, and tools. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW1):1–23.

Yuge Zhang, Qiyang Jiang, XingyuHan XingyuHan, Nan Chen, Yuqing Yang, and Kan Ren. 2024. Benchmarking data science agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5677–5700.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.

A Appendices

A.1 Task Categories

Category	Lifecycle / Macro Group
Data Loading and Multi-Table Joining	Data Acquisition
Web Data Scraping	Data Acquisition
Summary Statistics	EDA
Data Cleaning and Missing-Value Handling	EDA
Data Filtering and Conditional Queries	EDA
Correlation and Relationship Analysis	EDA
Grouping and Aggregation	EDA
Outlier Detection	EDA
Feature Engineering and Transformation	Feature Engineering
Dimensionality Reduction	Feature Engineering
Standardization and Normalization	Feature Engineering
Feature Importance and Interpretability	Feature Engineering
Classification	Modeling
Regression	Modeling
Clustering	Modeling
Ensemble Methods	Modeling
Imbalanced Data Handling	Modeling
Model Validation and Evaluation	Evaluation and Deployment
Hyperparameter Tuning	Evaluation and Deployment
Statistical Testing	Evaluation and Deployment
Visualization and Chart Generation	Visualization and Reporting
Data Export and Reporting	Visualization and Reporting

Table 5: Lifecycle grouping based on DS-World task category hierarchy.

Data Acquisition These tasks measure an agent's ability to locate, retrieve, and load data from diverse sources, including CSV files, multi-table joins, URL-based datasets, SQLite databases, and web-scraped content. Success depends on correctly identifying file paths and handling retrieval failures, reflecting real-world challenges in the earliest stage of analysis.

Exploratory Data Analysis (EDA) EDA tasks evaluate initial sense-making and inspection of raw data through summary statistics, filtering, outlier detection, correlation analysis, and group-wise trend identification. Agents must form meaningful interpretations before modeling, mirroring how human analysts spend a substantial portion of their workflow time.

Feature Engineering These tasks assess transformation and representation learning, requiring agents to create new features, scale numeric variables, encode categoricals, reduce dimensionality, and analyze feature importance. Effective feature engineering is essential for converting raw data into predictive structure rather than modeling unprocessed inputs.

Modeling Modeling tasks focus on building predictive systems through classification, regression, clustering, ensemble construction, and handling

Task Category	# Tasks	Example Task
Data Acquisition	23	Open the database, inspect the available tables and relevant columns, identify the customer ID and transaction fields, merge the required records, compute total and average transaction amounts per customer, calculate a risk score, and save the aggregated results to an output file.
Exploratory Data Analysis	119	Inspect a SQLite database schema, identify meal- and workout-related fields, join the relevant tables, compute an efficiency index, rank workout categories, and save the top five results to a CSV file.
Feature Engineering	37	Load a diabetes dataset, create a new insulin resistance feature, handle missing or zero values using median imputation, compute correlation with insulin levels, group results by outcome, and save summary statistics to output files.
Modeling	41	Load and merge multiple retail datasets by store and date, train multiple regression models to predict weekly sales, stack them using a meta-learner, perform cross-validation, and save the average error metric to a summary file.
Evaluation and Deployment	12	Perform multi-stage hyperparameter tuning for a gradient boosting model using cross-validation, select the best configuration based on error metrics, and save the final model parameters and performance results.
Visualization and Reporting	33	Analyze the relationship between body mass index, glucose level, and age with diabetes outcome, generate and inspect interactive visualizations, and create a PowerPoint report summarizing the key findings with appropriate charts, labels, and legends..

Table 6: Representative simplified Example Tasks in DSAGENTBENCH by Category

data imbalance. Many tasks require training multiple models and comparing validation metrics, reflecting realistic scenarios where model performance rather than code syntax determines success.

Evaluation and Deployment These tasks measure statistical rigor, model reliability, and interpretability. Examples include hyperparameter tuning, K-fold validation, metric comparison, hypothesis testing, and statistical significance evaluation. Agents must justify output quality using measurable evidence rather than relying solely on execution.

Evaluation and Deployment Visualization tasks test an agent's ability to communicate insights through bar charts, scatter plots, heatmaps, pairplots, and model comparison graphs, as well as structured text or file-based reports. Evaluators verify semantic correctness through labeled axes, titles, legends, and data mappings, ensuring meaningful analysis rather than surface-level image generation. We provide detailed breakdown of lifecycle grouping of task category and task examples in Table 5 and Table 6, respectively.

B Methodology

B.1 Environment and Inference Setup

We use different computational setups for closed-source and open-source models. Closed-source models are accessed exclusively through provider APIs and therefore do not require local model hosting. For these models, we run our experiments on an Ubuntu virtual machine created using VMware, where we deploy the our framework to manage API-based inference and evaluation.

For open-source models, we adopt a self-hosted setup to improve cost efficiency and enable concurrent inference across multiple models. Specifically, we run Docker-based virtual machines on Google Cloud Platform (GCP) instances with GPU support. Each model is served using the vLLM inference engine, and inference is performed by querying the locally hosted vLLM endpoints. Each GCP instance runs `Ubuntu 22.04 LTS` on an `n1-standard-4` machine with a 200 GB persistent standard disk.

B.2 Initial State Setup

As our experiments are conducted within the OSWorld environment, we retain its default system configuration, which includes pre-installed applications such as **VS Code**, **Google Chrome**, and other standard utilities. To support interaction for the UI-TARS model family, we additionally install the `pyperclip` library, enabling programmatic copy–paste operations during code execution.

We then prepare a lightweight execution environment for reproducible analysis. Specifically, we

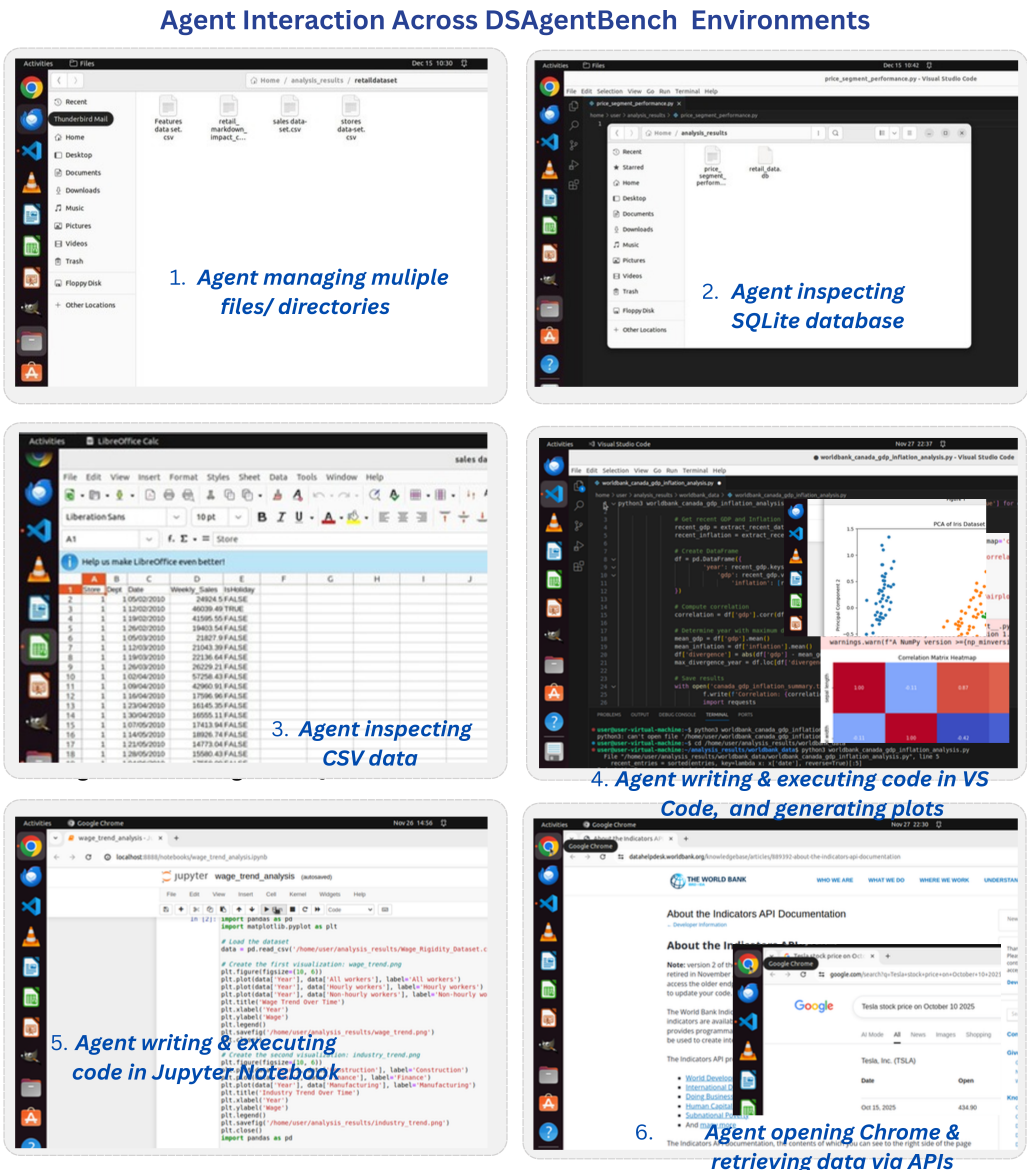

Figure 5: Example environments in DSAgentBench: agent interaction across data-science environments, demonstrating autonomous coordination of file systems, databases, notebooks, IDEs, visualization, execution, debugging, and web-based data retrieval.

install the **Jupyter Notebook** package to enable notebook-based workflows and create a dedicated directory for storing intermediate and final analysis results. Required input data are downloaded to predefined filesystem locations, and a task-specific Python script is initialized as the main execution entry point. Finally, the analysis environment is launched either through Jupyter Notebook or a code editor, enabling interactive development and execution.

B.3 Agent Prompt Templates

This appendix summarizes the canonical system prompts used to control agent behavior under different observation modalities. Rather than listing full implementation-level prompts inline, we abstract each prompt by its observation inputs, output constraints, and grounding assumptions.

For full reproducibility, we will release the complete system prompts, evaluation functions, and model-specific inference scripts for all supported agents as part of the project codebase.

B.4 Unified Action Space

All agents interact with the desktop environment through a unified action space to ensure consistent execution and fair comparison across models and observation modalities (see Tab. 8).

All screen coordinates are defined relative to a fixed resolution of 1920×1080. By constraining agents to this shared action space, performance differences reflect reasoning, grounding, and planning

Observation Modality	Output Format	Prompt Description
Screenshot-only	Code / Action	The agent receives a full-resolution screenshot of the desktop at each step and predicts grounded mouse and keyboard actions based solely on visual information. Interaction is coordinate-based, without access to image matching or additional screenshots. At each step, the agent must return either executable actions or a control token (`WAIT`, `FAIL`, `DONE`).
Screenshot + A11y Tree	Code / Action	The agent jointly observes both the screenshot and the accessibility tree extracted via AT-SPI, enabling alignment between visual context and structured UI metadata. This hybrid prompt improves grounding precision and interaction robustness and is used for all main experiments unless stated otherwise.

Table 7: Canonical agent prompt templates used under different observation modalities. One representative prompt is shown per setting; other variants differ only in minor wording or formatting.

Action Category	Description
Mouse Actions	Handles spatial interactions including absolute cursor movement (`MOVE_TO`, `DRAG_TO`), scrolling (`SCROLL`), and button operations (`CLICK`, `RIGHT_CLICK`, `DOUBLE_CLICK`, `MOUSE_DOWN`, `MOUSE_UP`).
Keyboard Actions	Manages alphanumeric input via `TYPING`, discrete key presses (`PRESS`), simultaneous key combinations (`HOTKEY`), and raw key state manipulation (`KEY_DOWN`, `KEY_UP`).
Control Actions	Special meta-actions to pause execution (`WAIT`) or terminate the episode by signaling failure (`FAIL`) or success (`DONE`).

Table 8: Unified action space listing specific commands and their functions.

capability rather than interface-specific advantages.

B.5 Implementation Details

All agents are evaluated under the unified environment and observation settings described in Section 4.1 to ensure fair and consistent comparison across models. We use default API configurations for closed-source models, with the temperature fixed at 0.1 to improve determinism while retaining limited response diversity. Closed-source models are evaluated via provider APIs without local hosting, using an OSWorld-based Ubuntu virtual machine running the DSworld framework.

Open-source models are evaluated in a self-hosted setting on Google Cloud Platform (GCP) compute instances using Docker-based virtual machines. Models are served using the vLLM inference engine (Kwon et al., 2023), enabling efficient and concurrent inference across multiple model variants. We will release the complete experimental repository, including prompts, environment configurations, evaluation code, and inference scripts, to facilitate reproducibility.

Key evaluation parameters.

- **Observation:** screenshot, screenshot + accessibility tree

Model	Mean Steps	Median Steps	Std
CUA	15.00	15	0.00
GPT-4.0	10.03	15	7.63
Gemini-2.5-Pro	**6.76**	**6**	2.93
GPT-5-Mini	7.33	5	3.82
Claude-Sonnet-4.6	10.93	12	3.04

Table 9: Efficiency of successful runs measured by interaction steps.

Model	Grounding	Terminal	Code	Logic
CUA	**94.44**	2.78	0.46	2.31
GPT-4.0	80.00	5.64	13.33	1.03
Gemini-2.5-Pro	43.08	**36.15**	13.85	6.92
GPT-5-Mini	56.92	7.69	30.77	4.62
GPT-5	41.67	8.33	38.33	11.67
Claude-Sonnet-4.5	39.32	23.07	27.60	10.00
Claude-Sonnet-4.6	32.77	9.24	**43.70**	**14.29**
GUI-OWL-7B	97.09	1.09	1.45	0.36
UI-Tars-1.5-7B	**98.18**	1.82	0.00	0.00
Jedi-7B	81.89	6.69	7.09	4.33

Table 10: Root-cause failure analysis across models (%).

Metric	15 Steps	30 Steps	50 Steps
Task Success Rate (%)	24.54	25.45	25.81
Average Score	0.55	0.56	0.57

Table 11: Ablation results for GPT-4o on DSAgentBench under different interaction step budgets.

- **Action space:** `pyautogui`
- **Resolution:** 1920×1080
- **Decoding:** temperature $= 0.1$, top-$p = 0.9$
- **Max output tokens per call:** 2000
- **Task timeout:** 1800 seconds

B.6 Reproducibility and Environment Portability

DSAgentBench is designed to support reproducible execution while remaining extensible to other desktop environments. Although all experiments in this paper are reported on an Ubuntu-based environment, DSAgentBench is built by extending OSWorld, which supports major desktop operating systems, including Ubuntu/Linux, Windows, and

macOS. Each benchmark task specifies the required datasets, file-system state, applications, libraries, setup procedures, and deterministic evaluation metrics. To avoid dependence on changing external data availability, all datasets collected from public sources such as Kaggle, UCI/OpenML, GitHub, and SQLite repositories are pre-downloaded and will be released together with the benchmark. This task-level configuration makes the benchmark reproducible and portable: the same tasks can be instantiated on other OSWorld-supported operating systems by preparing the corresponding software stack, while reusing the same deterministic evaluators to verify final outputs. In addition, the environment design is extensible to other data-science tools, such as R/RStudio, PyCharm, and cloud-based environments, through additional environment configurations and task-specific setup scripts.

For the small subset of visualization/reporting tasks that require qualitative assessment, we use an LLM-based visual judge only after deterministic validation gates have passed. Rule-based evaluators first verify execution correctness, required output files, data mappings, labels, legends, and task-specific constraints. The LLM judge is then restricted to assessing visual quality and semantic alignment with the task instruction. To reduce judge bias, no model evaluates its own outputs: Gemini-2.5-Pro judges GPT-4o outputs, while GPT-4o judges outputs from other models. We include the judge rubric in Tab. 13.

We further ensure reproducibility by fixing and logging the core execution parameters for every run, including the action space, observation type, screen resolution, execution delay, task timeout, maximum step budget, maximum trajectory length, and model decoding settings. In our setup, agents use a PyAutoGUI action space, a 2.0-second post-action delay, a 1800-second task timeout, temperature of 0.1, and top-p of 0.9. Each task is defined by a complete JSON configuration specifying the instruction, required environment setup, datasets, tools, libraries, and a custom deterministic evaluation function, making each task independently executable and verifiable. To assess run-to-run stability, we repeated the Screenshot+A11y Tree evaluation for GPT-4o and Claude-4.5-Sonnet; the overall accuracies remained within ±1%, with GPT-4o achieving 24.54% and 23.63%, and Claude-4.5-Sonnet achieving 9.21% and 9.81%. This indicates that the benchmark pipeline produces stable and reproducible measurements.

C Evaluation Functions

C.1 Numerical Result Verification

This evaluator targets tasks with a known ground-truth numerical answer (e.g., correlation, volatility, or aggregate counts). It verifies correct script execution and numerical accuracy within a predefined tolerance (see Tab. 12).

This evaluator is fully deterministic and does not rely on language models, ensuring stable and repeatable scores across runs.

C.2 Visualization Quality Evaluation

For visualization tasks, correctness cannot be assessed purely numerically. We therefore combine deterministic checks with grounded semantic validation to assess chart clarity and representational fidelity (see Tab. 13).

This evaluator captures semantic correctness while minimizing subjectivity through constrained prompts and binary judgments.

D Additional Results

We provide more result analysis in this section (see Table 14, Table 15, and Table 16).

D.1 Error Analysis Examples

Additional qualitative error cases produced by different models are illustrated in figure 10, figure 11, figure 12, and figure 13. Across these examples, we observe several recurring failure patterns that are consistent across model families.

A prominent source of errors arises from the models' difficulty in handling environmental notifications and system-level UI elements. In particular, the models frequently fail to correctly interpret or dismiss generic pop-up notifications originating from the code editor environment, which subsequently disrupts task progression and leads to incorrect or incomplete actions.

In addition, the models exhibit systematic weaknesses in maintaining proper code formatting and indentation. We observe multiple instances where a model intends to insert a line break but instead explicitly generates the string token \n within the code. Since this token is not interpreted as an actual newline by the execution environment, it results in syntactically invalid or semantically incorrect code. This mismatch between the model's internal representation of formatting and the execution semantics ultimately propagates downstream errors and task failure.

Evaluation Step	Description
Script Validation	Confirms that the required Python script exists and executes successfully.
Output Verification	Verifies that the expected output file (e.g., `.txt` or `.csv`) is generated.
Value Extraction	Extracts numeric values using robust parsing to avoid formatting artifacts.
Numerical Matching	Compares extracted values against ground-truth references using a small tolerance to account for floating-point variation.
Scoring	Assigns partial credit for intermediate correctness and caps the final score at 1.0.

Table 12: Structure of a deterministic numerical evaluation function.

Evaluation Step	Description
Script Validation	Ensures that the visualization script exists and executes successfully.
Artifact Verification	Confirms that required visualization files are generated and non-empty.
Semantic Validation	Uses a fixed evaluation prompt with GPT-4o as a judge to assess whether each chart correctly represents the intended variables, axes, and trends.
Design Criteria	Checks readability, labeling, scale consistency, and layout clarity.
Scoring	Combines deterministic and semantic signals into a bounded score in the range [0, 1].

Table 13: Structure of a visualization quality evaluation function.

Overall, these error patterns highlight limitations in current models' robustness to interactive development environments and their ability to reliably translate formatting intentions into executable code.

Model	#Tasks	Mean FF Step	Median FF	Early (%)	Mid (%)	Late (%)	Budget Exhaustion (%)	Recovery (%)
CUA	65	13.62	15	3.41	3.41	**93.17**	**100.0**	0.0
GPT-4.0	100	8.77	15	41.54	3.59	54.87	93.85	1.54
Gemini-2.5-Pro	100	5.80	6	42.02	**45.38**	12.61	62.31	**19.23**
GPT-5-Mini	100	5.92	4	**57.38**	19.67	22.95	63.08	3.08
GUI-OWL-7B	50	14.16	15	2.91	4.00	93.09	100.0	0.0
UI-Tars-1.5-7B	50	14.40	15	0.00	0.00	**100.00**	100.0	0.73
Jedi-7B	50	12.10	15	11.02	18.11	70.87	98.43	1.18

Table 14: Model-wise performance and trajectory behavior (normalized over 100 tasks per model for comparison). FF denotes the *first failure step*, i.e., the earliest step in the trajectory where an error is detected.

Model	Screenshot							Screenshot + A11y Tree						
	DA	EDA	FE	Model	Vis	Eval	Overall	DA	EDA	FE	Model	Vis	Eval	Overall
Jedi-3B w/GPT4o	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Jedi-7B w/GPT4o	0.84	0.00	0.00	0.00	1.00	0.00	0.73	N/A	N/A	N/A	N/A	N/A	N/A	N/A
UI-Tars-2B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A	N/A	N/A	N/A	N/A	N/A	N/A
UI-Tars-1.5-7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A	N/A	N/A	N/A	N/A	N/A	N/A
GUI-OWL-7B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCUA-72B	4.35	0.91	0.00	0.00	0.00	0.00	0.73	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 15: DSAGENTBENCH **accuracy** (%) across data-science lifecycle task types and overall performance under Screenshot settings (**Open-Source Models**).

Model	Screenshot							Screenshot + A11y Tree						
	DA	EDA	FE	Model	Vis	Eval	Overall	DA	EDA	FE	Model	Vis	Eval	Overall
Closed-Source Models														
GPT4-o	0.26	**0.59**	**0.63**	**0.50**	0.48	**0.55**	**0.53**	**0.37**	**0.58**	**0.61**	**0.52**	0.60	**0.39**	**0.55**
O4-mini	0.28	0.27	0.35	0.31	0.37	0.16	0.30	0.26	0.37	0.42	0.37	0.42	0.31	0.37
GPT5-mini	**0.31**	0.49	0.55	0.45	**0.55**	0.32	0.47	0.27	0.49	0.57	0.44	0.58	0.31	0.48
GPT5	0.55	0.60	0.62	0.61	0.58	0.47	0.59	0.65	0.67	0.63	0.66	0.67	0.40	0.65
Gemini-Pro-2.5	0.27	0.48	.47	0.48	0.46	0.48	0.45	0.25	0.51	0.61	0.45	**0.68**	0.38	0.50
OpenAI CUA	0.22	0.43	0.46	0.41	0.51	0.32	0.42	0.27	0.40	0.37	0.40	0.44	0.34	0.39
Claude-4-Sonnet	0.22	0.37	0.37	0.39	0.37	0.27	0.35	0.24	0.37	0.35	0.39	0.45	0.17	0.36
Claude-4.5-Sonnet	0.26	0.36	0.36	0.39	0.47	0.36	0.37	0.28	0.41	0.46	0.37	0.46	0.50	0.41
Claude-4.6-Sonnet	0.68	0.72	0.70	0.74	0.75	0.71	0.72	0.64	0.80	0.77	0.76	0.75	0.72	0.76
Open-Source Models														
Jedi-3B w/GPT4o	0.22	0.31	0.30	0.34	0.35	0.31	0.31	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Jedi-7B w/GPT4o	0.20	0.30	0.31	0.34	0.37	0.31	0.31	N/A	N/A	N/A	N/A	N/A	N/A	N/A
UI-Tars-2B	0.08	0.06	0.03	0.08	0.12	0.10	0.07	N/A	N/A	N/A	N/A	N/A	N/A	N/A
UI-Tars-1.5-7B	0.00	0.02	0.00	0.03	0.03	0.02	0.02	N/A	N/A	N/A	N/A	N/A	N/A	N/A
GUI-OWL-7B	0.03	0.05	0.08	0.04	0.01	0.08	0.05	N/A	N/A	N/A	N/A	N/A	N/A	N/A
OpenCUA 72b	0.10	0.09	0.09	0.09	0.18	0.06	0.10	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 16: DSAGENTBENCH **average score (%)** across ds lifecycle task types and overall performance under Screenshot and Screenshot + Accessibility Tree observation settings.

Standard System Prompt (Screenshot Input → Code Output)

```
You are an agent which follow my instruction and perform desktop computer tasks as instructed.
You have good knowledge of computer and good internet connection and assume your code will run
on a computer for controlling the mouse and keyboard.
For each step, you will get an observation of an image, which is the screenshot of the
computer screen and you will predict the action of the computer based on the image.

You are required to use ‘pyautogui‘ to perform the action grounded to the observation, but
DONOT use the ‘pyautogui.locateCenterOnScreen‘ function to locate the element you want to
operate with since we have no image of the element you want to operate with.  DONOT USE
‘pyautogui.screenshot()‘ to make screenshot.
Return one line or multiple lines of python code to perform the action each time, be
time efficient.  When predicting multiple lines of code, make some small sleep like
‘time.sleep(0.5);‘ interval so that the machine could take; Each time you need to predict
a complete code, no variables or function can be shared from history
You need to to specify the coordinates of by yourself based on your observation of current
observation, but you should be careful to ensure that the coordinates are correct.
You ONLY need to return the code inside a code block, like this:
‘‘‘python
# your code here
‘‘‘
Specially, it is also allowed to return the following special code:
When you think you have to wait for some time, return ‘‘‘WAIT‘‘‘;
When you think the task can not be done, return ‘‘‘FAIL‘‘‘, don’t easily say ‘‘‘FAIL‘‘‘, try your
best to do the task;
When you think the task is done, return ‘‘‘DONE‘‘‘.

My computer’s password is ’{CLIENT_PASSWORD}’, feel free to use it when you need sudo rights.
First give the current screenshot and previous things we did a short reflection, then RETURN
ME THE CODE OR SPECIAL CODE I ASKED FOR. NEVER EVER RETURN ME ANYTHING ELSE.
```

Few-Shot System Prompt (Screenshot Input → Code Output)

```
You are an agent which follow my instruction and perform desktop computer tasks as instructed.
You have good knowledge of computer and good internet connection and assume your code will run
on a computer for controlling the mouse and keyboard.
For each step, you will get an observation of an image, which is the screenshot of the
computer screen and the instruction and you will predict the next action to operate on the
computer based on the image.

You are required to use ‘pyautogui‘ to perform the action grounded to the observation, but
DONOT use the ‘pyautogui.locateCenterOnScreen‘ function to locate the element you want to
operate with since we have no image of the element you want to operate with.  DONOT USE
‘pyautogui.screenshot()‘ to make screenshot.
Return one line or multiple lines of python code to perform the action each time, be
time efficient.  When predicting multiple lines of code, make some small sleep like
‘time.sleep(0.5);‘ interval so that the machine could take; Each time you need to predict
a complete code, no variables or function can be shared from history
You need to to specify the coordinates of by yourself based on your observation of current
observation, but you should be careful to ensure that the coordinates are correct.
You ONLY need to return the code inside a code block, like this:
‘‘‘python
# your code here
‘‘‘
Specially, it is also allowed to return the following special code:
When you think you have to wait for some time, return ‘‘‘WAIT‘‘‘;
When you think the task can not be done, return ‘‘‘FAIL‘‘‘, don’t easily say ‘‘‘FAIL‘‘‘, try your
best to do the task;
When you think the task is done, return ‘‘‘DONE‘‘‘.

My computer’s password is ’{CLIENT_PASSWORD}’, feel free to use it when you need sudo rights.
Our past communication is great, and what you have done is very helpful.  I will now give you
another task to complete.
First take a deep breath, think step by step, give the current screenshot a thinking, then
RETURN ME THE CODE OR SPECIAL CODE I ASKED FOR. NEVER EVER RETURN ME ANYTHING ELSE.
```

Figure 6: The system prompts used for the code-generation agent. The **Standard Prompt** (top) focuses on direct execution, while the **Few-Shot Prompt** (bottom) incorporates "deep breath" CoT triggering.

UITARS User Prompt (No Thought)

```
You are a GUI agent.  You are given a task and your action history, with screenshots.  You
need to perform the next action to complete the task.
## Output Format
‘‘‘
Action:  ...
‘‘‘
## Action Space
click(start_box=’<|box_start|>(x1,y1)<|box_end|>’)
left_double(start_box=’<|box_start|>(x1,y1)<|box_end|>’)
right_single(start_box=’<|box_start|>(x1,y1)<|box_end|>’)
drag(start_box=’<|box_start|>(x1,y1)<|box_end|>’, end_box=’<|box_start|>(x3,y3)<|box_end|>’)
hotkey(key=’’)
type(content=’’) #If you want to submit your input, use "\n" at the end of ‘content‘.
scroll(start_box=’<|box_start|>(x1,y1)<|box_end|>’, direction=’down or up or right or
left’)
wait() #Sleep for 5s and take a screenshot to check for any changes.
finished()
call_user() # Submit the task and call the user when the task is unsolvable, or when you
need the user’s help.

## User Instruction
{instruction}
```

UITARS User Prompt (Thought)

```
You are a GUI agent.  You are given a task and your action history, with screenshots.  You
need to perform the next action to complete the task.

## Output Format
‘‘‘
Thought:  ...
Action:  ...
‘‘‘

## Action Space
{action_space}

## Note
- Use {language} in ‘Thought‘ part.
- Write a small plan and finally summarize your next action (with its target element) in
one sentence in ‘Thought‘ part.

## User Instruction
{instruction}
```

Figure 7: The system prompts used for the UITARS agent. The top box shows the standard prompting strategy, while the bottom box includes ‘Thought’ reasoning requirements.

JEDI Grounder System Prompt

```
You are a helpful assistant.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools> XML tags:
<tools>
{{"type": "function", "function": {{"name": "computer_use", "description": "Use a mouse
and keyboard to interact with a computer, and take screenshots.\n* This is an interface to
a desktop GUI. You do not have access to a terminal or applications menu. You must click on
desktop icons to start applications.\n* Some applications may take time to start or process
actions, so you may need to wait and take successive screenshots to see the results of your
actions. E.g. if you click on Firefox and a window doesn't open, try wait and taking another
screenshot.\n* The screen's resolution is {width}x{height}.\n* Whenever you intend to move
the cursor to click on an element like an icon, you should consult a screenshot to determine
the coordinates of the element before moving the cursor.\n* If you tried clicking on a
program or link but it failed to load, even after waiting, try adjusting your cursor position
so that the tip of the cursor visually falls on the element that you want to click.\n* Make
sure to click any buttons, links, icons, etc with the cursor tip in the center of the element.
Don't click boxes on their edges unless asked.", "parameters": {{"properties": {{"action":
{{"description": "The action to perform. The available actions are:\n* `key`: Performs
key down presses on the arguments passed in order, then performs key releases in reverse
order.\n* `type`: Type a string of text on the keyboard.\n* `mouse_move`: Move the cursor
to a specified (x, y) pixel coordinate on the screen.\n* `left_click`: Click the left
mouse button.\n* `left_click_drag`: Click and drag the cursor to a specified (x, y) pixel
coordinate on the screen.\n* `right_click`: Click the right mouse button.\n* `middle_click`:
Click the middle mouse button.\n* `double_click`: Double-click the left mouse button.\n*
`scroll`: Performs a scroll of the mouse scroll wheel.\n* `wait`: Wait specified seconds
for the change to happen.\n* `terminate`: Terminate the current task and report its
completion status.", "enum": ["key", "type", "mouse_move", "left_click", "left_click_drag",
"right_click", "middle_click", "double_click", "scroll", "wait", "terminate"], "type":
"string"}}, "keys": {{"description": "Required only by `action=key`.", "type": "array"}},
"text": {{"description": "Required only by `action=type`.", "type": "string"}},
"coordinate": {{"description": "(x, y): The x (pixels from the left edge) and y (pixels
from the top edge) coordinates to move the mouse to. Required only by `action=mouse_move`,
`action=left_click_drag`, `action=left_click`, `action=right_click`, `action=double_click`.",
"type": "array"}}, "pixels": {{"description": "The amount of scrolling to perform.
Positive values scroll up, negative values scroll down. Required only by `action=scroll`.",
"type": "number"}}, "time": {{"description": "The seconds to wait. Required only by
`action=wait`.", "type": "number"}}, "status": {{"description": "The status of the task.
Required only by `action=terminate`.", "type": "string", "enum": ["success", "failure"]
}}}}, "required": ["action"], "type": "object"}}}}}}
</tools>

For each function call, return a json object with function name and arguments within
<tool_call> XML tags:
<tool_call>
{{"name": <function-name>, "arguments": <args-json-object>}}
</tool_call>
```

Figure 8: Example system prompt for the JEDI architecture. The Grounder prompt provides the precise tool definitions and XML schema for tool calling.

JEDI Planner System Prompt

```
You are an agent which follow my instruction and perform desktop computer tasks as instructed.
You have good knowledge of computer and good internet connection and assume your code will run on a computer for
controlling the mouse and keyboard.

You are on Ubuntu operating system and the resolution of the screen is 1920x1080.
For each step, you will get an observation of an image, which is the screenshot of the computer screen and you will
predict the action of the computer based on the image.

The following rules are IMPORTANT:
- If previous actions didn't achieve the expected result, do not repeat them, especially the last one.  Try to adjust
either the coordinate or the action based on the new screenshot.
- Do not predict multiple clicks at once.  Base each action on the current screenshot; do not predict actions for
elements or events not yet visible in the screenshot.
- You cannot complete the task by outputting text content in your response.  You must use mouse and keyboard to
interact with the computer.  Return ```Fail``` when you think the task can not be done.
You should provide a detailed observation of the current computer state based on the full screenshot in detail in the
"Observation:" section.
Provide any information that is possibly relevant to achieving the task goal and any elements that may affect the task
execution, such as pop-ups, notifications, error messages, loading states, etc..
You MUST return the observation before the thought.
You should think step by step and provide a detailed thought process before generating the next action:
Thought:
- Step by Step Progress Assessment:
     - Analyze completed task parts and their contribution to the overall goal
     - Reflect on potential errors, unexpected results, or obstacles
     - If previous action was incorrect, predict a logical recovery step
- Next Action Analysis:
     - List possible next actions based on current state
     - Evaluate options considering current state and previous actions
     - Propose most logical next action
     - Anticipate consequences of the proposed action

Your thought should be returned in "Thought:" section.  You MUST return the thought before the code.

You are required to use `pyautogui` to perform the action grounded to the observation, but DONOT use the
`pyautogui.locateCenterOnScreen` function to locate the element you want to operate with since we have no image of
the element you want to operate with.  DONOT USE `pyautogui.screenshot()` to make screenshot.
Return exactly ONE line of python code to perform the action each time.  At each step, you MUST generate
the corresponding instruction to the code before a # in a comment (example:  # Click Ÿes, I trust the
authorsbutton\npyautogui.click(x=0, y=0, duration=1)\n)

For the instruction you can decribe the element you want to interact with in detail including the visual description
and function description.  And make it clear and concise.
For example you can describe what the element looks like, and what will be the expected result when you interact with
it.
You need to to specify the coordinates of by yourself based on your observation of current observation, but you should
be careful to ensure that the coordinates are correct.
Remember you should only return ONE line of code, DO NOT RETURN more.  You should return the code inside a code block,
like this:
```python
# your code here
```
Specially, it is also allowed to return the following special code:
When you think you have to wait for some time, return ```WAIT```;
When you think the task can not be done, return ```FAIL```, don't easily say ```FAIL```, try your best to do the task;
When you think the task is done, return ```DONE```.

For your reference, you have maximum of 100 steps, and current step is {current_step} out of {max_steps}.
If you are in the last step, you should return ```DONE``` or ```FAIL``` according to the result.

Here are some guidelines for you:
1. Remember to generate the corresponding instruction to the code before a # in a comment and only return ONE line of
code.
2. If a click action is needed, use only the following functions: pyautogui.click, pyautogui.rightClick or
pyautogui.doubleClick.
3. Return ```Done``` when you think the task is done. Return ```Fail``` when you think the task can not be done.

My computer's password is '{CLIENT_PASSWORD}', feel free to use it when you need sudo rights.
First give the current screenshot and previous things we did a short reflection, then RETURN ME THE CODE OR SPECIAL
CODE I ASKED FOR NEVER EVER RETURN ME ANYTHING ELSE.
```

Figure 9: Example system prompt for the JEDI architecture. The Planner prompt enforces a strict structure for Observation, `Thought` reasoning, and Python code generation.

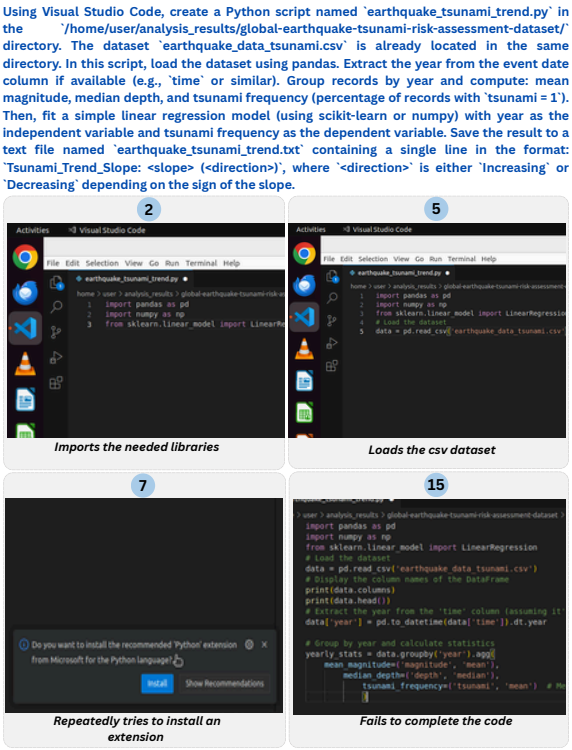

Figure 10: Example of error that occured with the model OpenAI CUA.

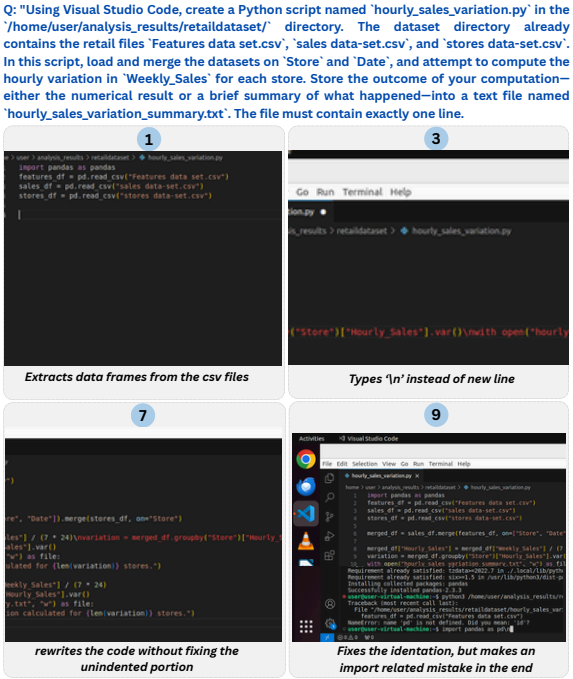

Figure 11: Example of error that occured with the model Jedi-3B.

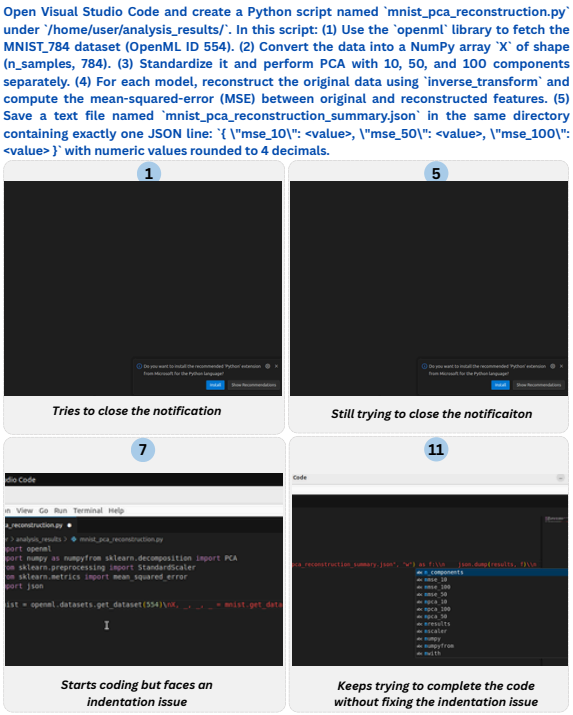

Figure 12: Example of error that occured with the model Jedi-8B.

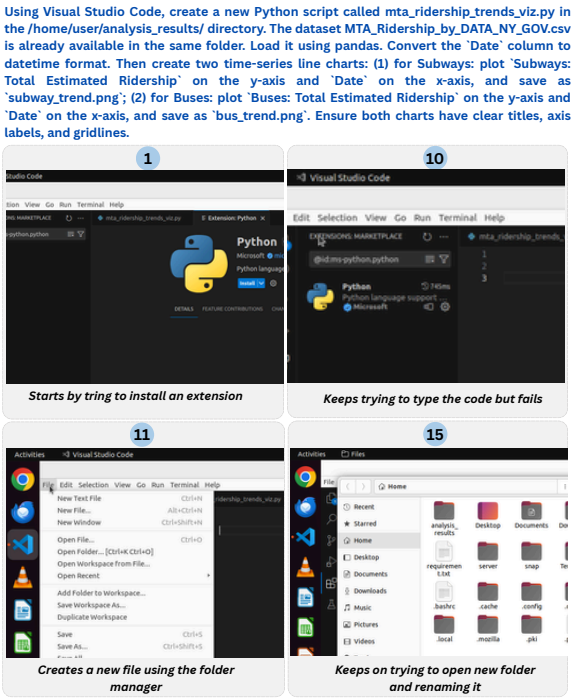

Figure 13: Example of error that occured with the model UI-Tars-1.5-7B.