SDRBench: A Simulated, Resettable RL Gym for Realistic Sales Development Representative Workflows

Manil Lakabi Pranav Raja Yasir Nadeem Mustafa Awwal
Muhammad Inamullah Tushar Chopra Rahul Chocha

Foundry (San Francisco, CA)
manil@foundryrl.com

September 22, 2025

Abstract

We introduce **SDRBench**, a benchmark and reinforcement learning (RL) gym for browser agents that targets the core challenges of long-horizon control, entity persistence, and schema alignment in enterprise SaaS workflows. SDRBench defines 50 deterministic tasks: 45 small–to–medium–horizon tasks (5–20 actions) and 5 long-horizon tasks (20+ actions) spanning Gmail, Google Sheets, LinkedIn-like prospecting, Apollo, and Salesforce. These tasks reflect realistic multi-application knowledge-work pipelines rather than isolated single-site interactions.

To enable reproducible, high-fidelity evaluation, SDRBench is implemented on Foundry's Agent Web Engine (AWE). AWE supplies stable, deterministic, resettable, and highly scalable browser-based environments by combining archived high-fidelity mocks with mock-enrichment, programmatic state models, and state-change logging. This architecture (i) removes common sources of evaluation drift (e.g., CAPTCHAs and live-site variability), (ii) supports snapshot/rewind and deterministic resets for controlled experiments, and (iii) exposes programmatic state and timelines that permit precise, state-delta evaluation and dense reward shaping for RL training.

A key property of SDRBench is a shared datastore that enforces cross-application state consistency. Entities persist globally across environments: a prospect sourced in the LinkedIn-like application must be normalized and logged in Salesforce, referenced in Gmail, and aligned with records in Sheets. This design stresses relational reasoning, memory across heterogeneous schemas, and robustness to cascading errors, dimensions that are central to the deployment of reliable browser agents but are underexplored in existing evaluation settings.

To balance realism with experimental control, SDRBench combines high-fidelity mocked clones with staging sandboxes (for example, Salesforce developer orgs). The environment supports deterministic resets, snapshot/rewind, and complete state introspection, enabling reproducible evaluation and RL training. Tasks are scored by state-delta predicates (for example, correct record creation and linkage) rather than brittle URL- or text-based heuristics, providing unambiguous success criteria while exposing dense shaping signals for learning.

We release SDRBench as a leaderboard-ready suite and include baseline results for several frontier agents. SDRBench establishes a controlled evaluation platform for end-to-end enterprise workflows, enabling research on generalization, credit assignment, and error recovery in browser-native environments.

1 Introduction

Autonomous browser and computer-use agents hold promise for automating a wide range of knowledge-work tasks. Recent systems have shown that large language models (LLMs) can drive agents that navigate websites, fill forms, and retrieve information; yet robust performance on long-horizon workflows remains elusive. In realistic enterprise settings, workflows require propagating entities across multiple applications, aligning heterogeneous schemas, and recovering gracefully from intermediate errors. These requirements reveal weaknesses in current agents: brittle memory, shallow state tracking, and difficulty with credit assignment when actions have delayed or cross-application consequences.

To study these challenges systematically, we introduce **SDRBench**, a benchmark and RL gym based on the day-to-day tasks of Sales Development Representatives (SDRs). SDR workflows are an ideal testbed: they are repeatable, structured, and inherently multi-application, spanning Gmail, Google Sheets, LinkedIn-like prospecting, Apollo, and Salesforce. In the current release, SDRBench defines 50 deterministic tasks: 45 small-to-medium horizon tasks (5–20 actions) capturing routine SDR activities such as lead sourcing and CRM entry, and 5 long-horizon tasks (20+ actions) such as Salesforce configuration and multi-step follow-up creation. The shorter tasks reflect the high-frequency operations that dominate SDR workflows, while the long-horizon tasks stress multi-application memory, schema alignment under constraints, and recovery from cascading errors—capabilities under-tested in prior benchmarks.

A central design choice in SDRBench is a unified datastore that links all environments. This ensures that entities persist globally: a prospect identified in the LinkedIn-like environment is the same record that must later be normalized into Salesforce, referenced in Gmail, and matched in Sheets. This unified backend transforms evaluation from checking isolated page completions into verifying cross-application consistency and relational correctness. It also introduces realistic failure modes, where small early mistakes (for example, a mistyped field) can propagate and invalidate downstream steps.

To balance realism with experimental control, SDRBench combines high-fidelity mocked clones with staging sandboxes (for example, Salesforce developer orgs). The environment supports deterministic resets, snapshot/rewind, and full state introspection, enabling reproducible experiments, controlled ablations, and RL training. Tasks are evaluated through state-delta predicates over environment backends (for example, verifying that a Salesforce contact exists with the correct fields and associations). This yields deterministic, unambiguous success signals while still allowing dense shaping rewards to be exposed for training.

Contributions

Our primary contributions are:

1. **A benchmark for controlled enterprise workflows.** SDRBench couples enterprise SaaS realism (Gmail, Sheets, LinkedIn-like prospecting, Apollo, Salesforce) with deterministic control (resets, snapshots, state introspection), enabling reproducible evaluation of long-horizon, cross-application workflows.

2. **State-based evaluation for reproducibility.** We define evaluation via environment deltas (record existence, field correctness, cross-application linkage), eliminating reliance on human or LLM judges and providing deterministic success criteria for scientific comparison and ablation studies.

3. **Dense reward schemas for RL research.** We design reward functions derived from state changes, exposing dense shaping signals while preserving binary end-task correctness, making

SDRBench suitable both for leaderboard evaluation and RL training on sparse, delayed-reward tasks.

4. **Baseline evaluations of frontier agents.** We provide initial results for several state-of-the-art agents, establishing baselines and highlighting common failure modes relevant to credit assignment, schema alignment, and error recovery.

2 Related Work

Web interaction benchmarks. WebArena offers self-hosted simulated sites and programmatic evaluation; WebVoyager and Mind2Web leverage live websites (with LLM/human evaluation); WebBench scales coverage to hundreds of top websites, stressing *write* tasks but inherits live-web fragility (anti-bot, evolving DOMs). Classic suites (MiniWoB, WebShop) target synthetic or single-site tasks.

Agents for computer use. OpenAI CUA, Anthropic CUA, Browser Use, Skyvern, and Nova Act (Amazon) are representative modern systems. SDRBench complements these by providing a *trainable* and *controllable* benchmark for realistic enterprise workflows.

3 Comparison to Existing Browser-Agent Benchmarks

We contrast SDRBench with widely used benchmarks. Green check (✓) indicates presence/strength; red cross (✗) indicates absence/weakness for that attribute.

Table 1: **Benchmark comparison (placeholder).** SDRBench emphasizes multi-app enterprise workflows, resettable simulation, and programmatic state-based evaluation suitable for RL.

Benchmark	Env Type	Enterprise	Multi-App	Resets	State Eval	Dense Rewards	RL-friendly
SDRBench (ours)	Mixed (Mock + Staging)	✓	✓	✓	✓	✓	✓
WebArena	Simulated	✗	✓	✓	✓	✗	✓
WebVoyager	Live Web	✗	✗	✗	✗	✗	✗
Mind2Web	Live / Snapshots	✗	✗	✗	✗	✗	✗
WebBench (Halluminate)	Mostly Live	✗	✗	✗	✗	✗	✗
WebShop	Simulated (Single-site)	✗	✗	✓	✓	✓	✓
MiniWoB++	Simulated (Toy)	✗	✗	✓	✓	✓	✓

4 Data Setup

4.1 Methodology and Task Development

4.1.1 Observational Research Approach

The development of our SDR benchmark dataset was grounded in extensive field research and direct observation. Our team conducted multiple shadowing sessions with practicing Sales Development Representatives across various industries and company sizes. During these sessions, we documented actual workflows, tool usage patterns, and the decision-making processes that SDRs employ in their daily operations. This ethnographic approach ensured that our benchmark tasks reflect genuine workplace scenarios rather than theoretical constructs.

Through these observations, we identified recurring patterns in SDR workflows, common pain points in tool navigation, and the critical junctures where automation could provide the most value. Each task in our benchmark represents a distilled version of real activities we witnessed, stripped of company-specific details but retaining the essential complexity and multi-step nature of actual SDR work.

4.1.2 Task Selection Criteria

While modern SDR roles heavily emphasize cold calling and voice-based outreach—often comprising 40–60% of an SDR's daily activities—we made a deliberate decision to focus our benchmark on application-specific operational tasks. This choice was driven by several factors:

1. **Measurability**: Application-based tasks provide clear, objective success criteria through UI state changes, database updates, and system-generated confirmations. These discrete outcomes enable consistent evaluation across different AI agents.

2. **Reproducibility**: Digital tasks within CRM systems, sales engagement platforms, and communication tools can be precisely replicated across multiple test runs, ensuring benchmark reliability and enabling meaningful performance comparisons.

3. **Automation Readiness**: While voice interactions remain challenging for current AI systems, operational tasks represent the immediate frontier for SDR automation. These tasks consume significant time yet follow relatively structured patterns, making them ideal candidates for AI assistance.

4. **Cross-platform Complexity**: SDRs typically navigate between 5–7 different applications daily. Our tasks deliberately span multiple platforms (Salesforce, Apollo.io, Gmail, LinkedIn, Google Calendar) to test an agent's ability to maintain context and execute workflows across system boundaries.

4.2 Task Distribution and Categorization

Our benchmark tasks are carefully balanced across multiple dimensions to ensure comprehensive coverage of SDR responsibilities. Each task is labeled across three primary dimensions:

4.3 Data Quality and Validation

Each task underwent rigorous validation to ensure:

Table 2: Task Distribution by Workflow Stage

Workflow Stage	**Number of Tasks**	**Description**
Prospecting	14	Finding and researching new potential customers
Initial Outreach	7	First contact attempts with prospects
Qualification	9	Assessing lead fit and readiness
Nurturing	11	Ongoing engagement and relationship building
Handoff	4	Transitioning qualified leads to Account Executives
Re-engagement	5	Reviving cold or lost opportunities

Table 3: Task Distribution by Activity Type

Activity Type	**Number of Tasks**	**Core Skills Tested**
Data Management	25	CRM hygiene, record updates, deduplication
Email	16	Composition, personalization, scheduling
Research	12	Company/contact investigation, competitive intel
Campaign Management	8	Sequence creation, A/B testing, enrollment
Social Selling	7	LinkedIn engagement, connection requests
Meeting Coordination	4	Calendar management, scheduling
Cross-functional Collaboration	7	Internal handoffs, team coordination

- **Completeness**: All necessary context and data are provided for task completion
- **Clarity**: Instructions are unambiguous and follow consistent formatting
- **Realism**: Tasks reflect actual SDR workflows as observed in field research
- **Measurability**: Success criteria are objective and automatically verifiable
- **Platform Authenticity**: Tasks use actual platform interfaces and workflows

The resulting dataset represents approximately 200 hours of SDR activity, condensed into reproducible benchmark scenarios that test the full spectrum of operational skills required in modern sales development roles.

5 The Human Advantage: Automation as Liberation

Our field research revealed that SDRs spend approximately 65% of their time on administrative tasks such as data entry, CRM updates, sequence management and leaving only 35% for actual prospect conversations. This benchmark deliberately targets these operational tasks because they represent the mechanical work that consumes SDR productivity without leveraging their uniquely human capabilities.

The true value of SDRs lies not in their ability to update Salesforce records or enroll contacts in sequences, but in their capacity for emotional intelligence, creative problem discovery, and authentic

Table 4: Task Distribution by Platform Focus

Platform Category	Number of Tasks	Primary Tools
CRM Operations	32	Salesforce
Sales Engagement	8	Apollo.io
Communication	13	Gmail, Google Calendar
Social Platform	8	LinkedIn
Multi-platform	5	Tasks requiring 3+ platforms

Table 5: Task Horizon Distribution

Horizon Type	Description	Number of Tasks	Percentage
Short Horizon	Single-session tasks that can be completed in one continuous workflow	45	90%
Long Horizon	Multi-session tasks requiring callbacks, follow-ups, or time-delayed actions	5	10%
Total		**50**	**100%**

relationship building. These are the skills that convert prospects into opportunities: reading voice tone, navigating organizational politics, uncovering unspoken pain points, and building genuine trust through conversation.

By automating the operational burden documented in this benchmark, organizations can potentially double an SDR's daily conversation time from 2–3 hours to 5–6 hours. This isn't about replacing SDRs—it's about freeing them to do what they do best: connect with people, understand complex business problems, and build the relationships that drive revenue. The future of sales development is human creativity amplified by machine efficiency, with each operating where they deliver maximum value.

6 Agent Setup

With the benchmark tasks and data infrastructure in place, the next step was to evaluate how state-of-the-art autonomous browser agents perform on SDRBench. To ensure meaningful comparison, we designed a standardized execution harness that provisions fresh browser sessions, routes traffic through controlled environments, and exposes a consistent interface for navigation, input, and state inspection. This allowed us to focus on agent reasoning and execution quality rather than environment variability.

The agents used in the final benchmarking were:

- OpenAI CUA
- Anthropic CUA
- Browser Use (Claude Sonnet 4)
- Browser Use (GPT-4o Mini)

Two additional candidates were initially tested but subsequently dropped:

- Amazon Nova Act
- Skyvern

6.1 Setup and Prompting Approach

All agents were evaluated under a one-shot paradigm with no manual intervention at any point during task execution. Each agent received identical prompting structure to ensure fair comparison:

- **System Prompt:** A comprehensive prompt defining the agent's role as an autonomous browser agent with explicit instructions for one-shot task completion. The prompt included:
  - Universal principles emphasizing autonomous execution without user interaction
  - Site-specific navigation constraints limiting agents to 8 predefined URLs
  - Environment-specific interaction patterns for each platform
  - Failure handling protocols requiring clear status reporting with reasoning
- **Navigation Constraints:** Agents were strictly limited to predefined sandboxed environments with explicit URL whitelisting. For example:
  - Apollo: `https://app.apollo.io` (CRM and sequence management)
  - Salesforce: `https://test.lightning.force.com` (lead and opportunity tracking)
  - LinkedIn: `https://linkedin.com` (social selling activities)
  - Gmail: `https://mail.google.com` (email communication)

  Any attempt to navigate outside these URLs resulted would result in task failure.
- **Environment-Specific Instructions:** Detailed navigation patterns for each platform were provided. For instance:
  - Apollo: "Hover left edge → sidebar expands → click target page"
  - LinkedIn: "Search → People filter → view profiles → Connect button"
  - Salesforce: "Menu button (9 dots) → Leads/Contacts/Opportunities"
- **Task Context and Credentials:** Standardized test credentials were provided for all environments:
  - Default login: `test@example.com` / `password123`
  - Test payment card: `4111111111111111`, exp: 12/25, CVV: 123
  - Test identity: John Doe, ZIP: 12345

This standardized prompting approach ensured all agents operated under identical constraints, preventing any single model from gaining advantage through prompt engineering or additional clarifications. The one-shot requirement particularly tested each agent's ability to interpret ambiguous instructions and recover from errors autonomously, reflecting real-world automation scenarios where human intervention is not feasible.

6.2 Browser Session Setup

Each run used a custom browser session provisioned via Playwright. We connected to this session using a **CDP (Chrome DevTools Protocol) URL**, and routed all network requests through a custom proxy to our mocked environments (Apollo, LinkedIn, Salesforce, etc.).

To support all the browser agents, we implemented an evaluation harness layer on top of them to ensure consistency in runs, logging, monitoring and output. This layer exposed browser actions (navigation, click, type, read, etc.) in a consistent format, allowing agents to interact with the browser programmatically. This abstraction was used by two agents e.g. OpenAI CUA and Anthropic CUA, both of which rely on us providing the agent loop and calling functions on the browser.

6.3 OpenAI CUA and Anthropic CUA

Both OpenAI CUA and Anthropic CUA were integrated via similar execution pipelines. Both agents were given relevant LLM Models, Anthropic was configured with (Claude Sonnet 4), and OpenAI CUA with `computer-use-preview`.

6.3.1 Message Construction

Unlike simple string-based prompts, these agents were provided a structured message list, each with its message content and a role:

- **System/Assistant Role:** Contained the complete system prompt, including navigation constraints, environment descriptions, and execution principles.
- **User Role:** Contained the task description and any supporting context (credentials, payment details, initial state).
- **Optional Extra Messages:** In some cases, credentials or payment information were included as separate user messages for clarity.

This multi-message format mirrored the agents' native API expectations (Anthropic expects assistant/system messages, OpenAI expects system and user messages), while preserving a consistent evaluation setup.

6.3.2 Agent Loop & Computer Abstraction

Once messages were prepared, each agent was run inside a step-based loop:

1. **Send Messages:** Submit the full message list to the model.
2. **Receive Tool Calls:** Capture the model's proposed action (convert structured JSON from CUA, native tool-calls format from Operator).
3. **Execute Actions:** Map the tool calls to our computer abstraction layer, which wraps Playwright and exposes a normalized API (scroll, click, type, read_text, etc.).
4. **Return Observations:** Feed execution results back into the conversation state as a new message alongside a screenshot of the current state.
5. **Iterate:** Repeat until the model signals completion or a hard stop is reached (an error or step budget reached).

This design kept execution deterministic and agent-agnostic: both models interacted with the browser under identical conditions.

6.3.3 Custom Tools & Reliability Enhancements

We augmented the default toolset with custom utilities:

- **Browser Navigation:** `goto()`, `back()`, and `forward()` functions were exposed as tools, enabling deterministic navigation.
- **Robust Element Handling:** `wait_for_selector()` and retry loops stabilized interactions with dynamic UIs (e.g., Salesforce's slow page loads).

6.4 Browser Use

The Browser Use family of agents created their own Playwright-backed browser instance, and connected to our main Playwright session via a CDP URL. This gave the agent direct control over the browser while still routing traffic through our mock environments. No computer abstraction layer was required.

6.4.1 Prompting

Browser Use agents were given a single combined prompt string, which can be regarded as a string obtained by joining the message list.

6.4.2 Execution Approach

Browser Use executes tasks in a single, uninterrupted pass, through a single function call to its SDK.

- **DOM-Based Interaction:** Combines *vision* capabilities with its own lightweight DOM tree, allowing direct element interaction (visual bounding boxes around elements can be seen as well).
- **Single-Step Execution:** No iterative tool call, the execution continues until completion or timeout.
- **Lower Hallucination Rate:** The DOM approach reduces common vision-only errors such as clicking invisible buttons or ignoring disabled states.

Browser Use was benchmarked using two different LLM Models, (Claude Sonnet 4) and (GPT-4o Mini). Claude demonstrated more reliable planning and higher success rates on multi-step workflows, while GPT provided significantly faster responses.

6.5 Skyvern (Dropped)

Skyvern was evaluated with both (Claude Sonnet 4) and (GPT-4o Mini) as LLM Models. Its setup mirrored Browser Use, employing a concatenated prompt string and connecting to our browser session via a CDP URL.

However, Skyvern was excluded from final benchmarking due to:

- Excessive latency from scraping the DOM after every reasoning step, even during non-action phases.
- Redundant actions and inefficient navigation, leading to extremely long episode runtimes.
- Frequent misuse of in-page search boxes for navigation, failing to leverage its own Playwright abstraction.

6.6 Amazon Nova Act (Dropped)

Amazon Nova Act follows an act-based paradigm, where each request must be expressed as a structured sequence of acts. To integrate it into the benchmarking architecture, we built an intermediary LLM stage that translated task descriptions into a chain of actions (up to five sub-actions each within the same environment). These were converted into Nova Act calls with appended system prompts, effectively turning the prompt into a step-by-step plan.

Although this approach worked for isolated tasks, it introduced:

- **Context Fragmentation:** Each act was stateless, requiring manual stitching of context between steps, often failing on multi-site workflows.
- **Pipeline Complexity:** The extra translation layer increased latency and debugging overhead, complicating maintenance.

While Nova Act technically supported Playwright with CDP, its browser implementation was fragile—relying on its own orchestration and life cycle logic, and failing to gracefully recover from navigation errors or DOM lookup failures. This resulted in frequent run terminations. These limitations, along with poor browser context retention, led to Nova Act being dropped from final benchmarking.

7 System and Environment Setup

SDRBench is implemented on **Foundry**'s simulation platform, which enables benchmarks to run *consistently*, *deterministically*, and *repeatably*. This section describes the architecture and subsystems that make benchmarking and labeling feasible for complex, real-world web applications.

7.1 Components

The system comprises two primary components:

- **Agent Web Engine (AWE)** — Provides stable, resettable, and scalable browser-based environments for agents.
- **Data Warehouse (Shared Data Layer)** — A centralized datastore that connects simulated applications to a common cross-application entity model.

7.2 Agent Web Engine (AWE)

AWE 1 is the foundation of Foundry's platform. It supplies *stable*, *deterministic*, *resettable*, and *highly scalable* environments for browser agents, mitigating common issues such as CAPTCHAs, rate limits, and evaluation drift caused by live website changes. AWE implements these guarantees through the subsystems described below.

1. **App Mocking Subsystem.** Foundry generates high-fidelity mocks directly from live sites by archiving pages and preserving styling with pixel-level accuracy. These mocks are visual replicas (1:1) of the originals; interactivity and statefulness are added subsequently.
2. **Feature Definition.** Application features are enumerated and scoped across the entire application according to benchmarking requirements. Human reviewers verify fidelity against specification. As part of this step, a *programmatic state representation* of the application is established to ensure correctness and repeatability.

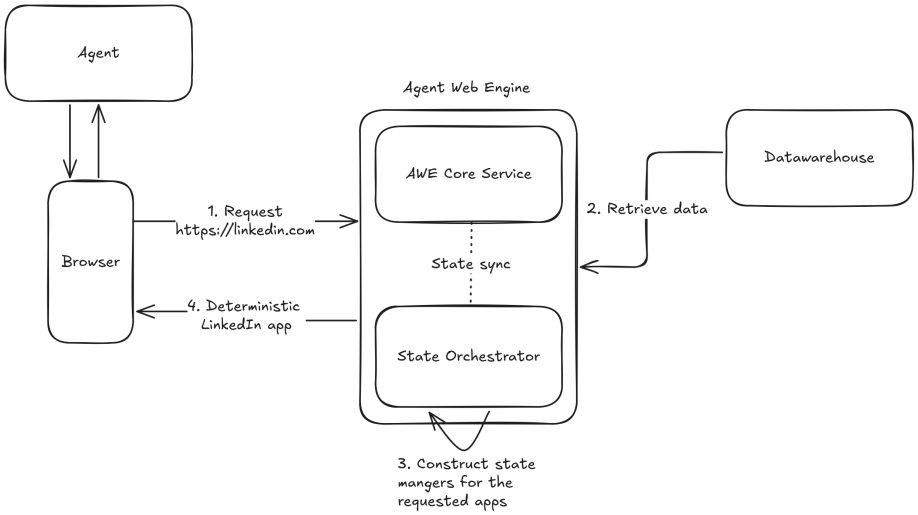


Figure 1: **AWE System Overview**

3. **Mock Enrichment.** Defined features are implemented against the application state model and injected into the archived page mocks, adding the behavior required for agent interactions and evaluation.

4. **State Management.** A per-application *state manager* orchestrates all operations. Each interaction that changes application state is modeled as an *action*; state updates are performed via a pure function:

$$\mathrm{F}(\mathrm{S}_{\mathrm{n}-1},\ \mathrm{A}) \rightarrow \mathrm{S}_{\mathrm{n}}$$

5. **State Change Logging and Browser Session Capture.** Middleware attached to state-change triggers logs each event with timestamps and metadata (e.g., feature used, prior values, updated values). For example:

```
# Old state
state = Amazon({
    "cart": {"items": [], "otherProperties": "..."},
    "otherFeatures": "..."
})

# Log old state
Log(state)

event = {"action": "add_item", "payload": [CartItem, "..."]}

# Log change
Log(event)

# Update state
state = Update(old_state, event)
```

```
# Log new state
Log(state)
```

For task-execution capture, we record mouse movements, clicks, and other browser events (using RRWeb) and correlate them with AWE's state-change timeline [11].

6. **Interoperability.** Because application states are programmatic, they can interact between applications. For example, a sequencing tool can trigger an email via Gmail in the same task context:

```
# inside a sequencing workflow
def handle_sequence(...):
    # ... sequence logic
    gmail = Context.get_state(Gmail, ...)  # includes required metadata
    gmail.send_email({
        "to": ["..."],
        "cc": ["..."],
        "bcc": ["..."],
        "subject": "subject",
        "body": "body",
        "other_properties": "..."
    })
```

Similar cross-application flows include placing an order on Amazon and subsequently receiving an order-confirmation email in the mail application for downstream checks (e.g., tracking).

7. **Resetting environments:** The applications states are tracked as a group of environments under one session for any given task. The programmatic interface for each environment in AWE exposes a 'reset' function that allows you to clear state changes and rewind the episode back to initial state.

8. **Tree exploration:** We cannot emphasis enough that states are just programmatic representation of an application Foundry supports. This means our possibilities here are theoretically endless in terms of operations we can perform on the application state. One of which is the support to rewind and explore state trajectory from a particular point in time. This combined with rule based heuristics to evaluate tasks unlocks the ability to do gradient based policy optimizations in RL (more on that later).

7.3 Data Warehouse (Shared Data Layer)

AWE integrates with a central datastore that is shared across all simulated applications. This provides a coherent, cross-application entity view (e.g., people, companies, accounts), enabling workflows that span multiple systems.

- **Example.** An agent finds a "VP of Data and Analytics" on LinkedIn and enriches the corresponding `Contact/Account` in Salesforce. Both LinkedIn and Salesforce views refer to the same person record in the shared datastore.

- **Adapter Abstraction.** Each application integrates via a lightweight adapter that defines:
  1. the transformation from the shared schema to app-specific structures
  2. the feature surfaces inside the application where the data should be available.

  This "Lego-style" plug-in model keeps integrations uniform and maintainable.

The net effect is that we avoid building full-fledged, bespoke replicas of every site. Instead, applications are *programmatic representations* backed by compact state models that are easy to mutate, reset, and scale. AWE orchestrates actions and state transitions while the Data Warehouse supplies cross-application context for end-to-end tasks.

7.4 Staging on Live Systems

For certain complex applications — or where safe, stateless sandboxes already exist — we layer AWE's capabilities on top of staging environments. For example, with Salesforce we use *Scratch Orgs* and add a thin shim for state tracking, resets, and determinism [12].

7.5 Open-Source Clones

To accelerate support for some applications, we leverage MIT-licensed open-source clones where appropriate. For example, a Google Calendar clone can be integrated via a thin adapter that translates UI actions into AWE state updates, preserving deterministic orchestration and logging semantics.

7.6 Observation and action space

Due to the breadth of the events and state information we capture during task run, we're able to provide the following as required:
Observations. (i) *Textual*: structured, filtered DOM text with interactable elements; (ii) *Visual* (optional): screenshot for multimodal agents.
Actions. Click(element), Type(text, field), Press(key), Navigate(url), Scroll(direction), Copy/Paste (optional).

8 Evaluation Methodology: Deterministic State-Based Verification

8.1 Ground Truth Establishment

Our evaluation methodology employs a deterministic, state-based approach that eliminates subjective interpretation from performance assessment. The process begins with human demonstration: for each benchmark task, an expert SDR performs the complete workflow in our simulated environment while we capture the full state trajectory. The final state achieved by the human operator becomes our ground truth, which is the definitive correct outcome for a task. The ground truth is then used to evaluate or create rule-based heuristics to evaluate the agents on a task.

This human-in-the-loop approach ensures our success criteria reflect actual SDR workflows rather than theoretical ideals. The ground truth captures not just the end goal, but the specific state changes required in each application: which fields must be populated, what records need creation or modification, and how different systems should be synchronized.

8.2 Heuristic Function Development

From each ground truth state, we derive programmatic heuristics as Python functions that encode the essential success criteria for task completion. These functions perform deterministic checks on the final state achieved by an AI agent, comparing it against the ground truth to verify task success. Importantly, these heuristic functions are designed to always pass when run against the ground truth itself, providing a validation mechanism for our evaluation logic.

For example, in task GTS-18 "Enroll a List of Contacts into an Apollo Sequence," the heuristic function verifies:

- A sequence named "Retail IT - Initial Outreach" exists
- Exactly 2 contacts have been added to the sequence
- All enrolled contacts have their status set to "active"

Tasks often include multiple heuristics that verify different aspects of completion, allowing us to measure partial success and identify which specific subtasks an agent accomplished even when full task completion wasn't achieved.

8.3 Evaluation on Tracked States

The core innovation of our evaluation system lies in comprehensive state tracking across simulated environments. Our Agent Web Engine (AWE) server maintains complete state representations throughout task execution. Each user action whether a form submission, navigation, or data modification triggers a request, signaling AWE to mutate and persist the state accordingly. This architecture ensures that simulated sites remain visually functional for AI agents while maintaining deterministic state tracking through programmatic representation of the apps without requiring traditional backend, database or related infrastructure per app.

This approach provides several critical advantages:

- **Deterministic Verification**: Success criteria are binary and unambiguous—either the required state changes occurred or they didn't
- **Partial Progress Assessment**: Multiple heuristics per task reveal how many subtasks were completed successfully.
- **Reproducibility**: The same initial state always produces consistent evaluation results across all evaluated agents.
- **Efficiency**: State-based evaluation is computationally lightweight compared to visual or semantic analysis.

8.4 Evaluation Parameters

The benchmark evaluation employed standardized parameters to ensure consistent and fair comparison across all agents. For execution control, we implemented differentiated step budgets based on task complexity: short horizon tasks were allocated 100 steps while long horizon tasks received 200 steps, reflecting their increased operational requirements. A universal timeout of 30 minutes was enforced for all task executions, after which the system would halt and record the current progress state, allowing partial credit for incomplete tasks. To accommodate platform-specific latencies, particularly for resource-intensive applications like Salesforce, we configured navigation timeouts of 5 minutes

to prevent premature failure due to slow page loads. The evaluation framework incorporated two distinct retry mechanisms to ensure robustness: first, complete task retries with exponential backoff were triggered upon task execution failures, allowing agents to recover from transient errors; second, API-level retries were implemented with a minimum of 3 attempts for handling HTTP 500 errors and rate limiting responses from the agent providers (OpenAI, Anthropic, and Browser Use), ensuring that temporary service disruptions did not unfairly penalize agent performance. These parameters were uniformly applied across all executions to maintain experimental validity and enable meaningful cross-agent comparisons.

9 Results

Figure 1: Success Rates by Workflow Stage

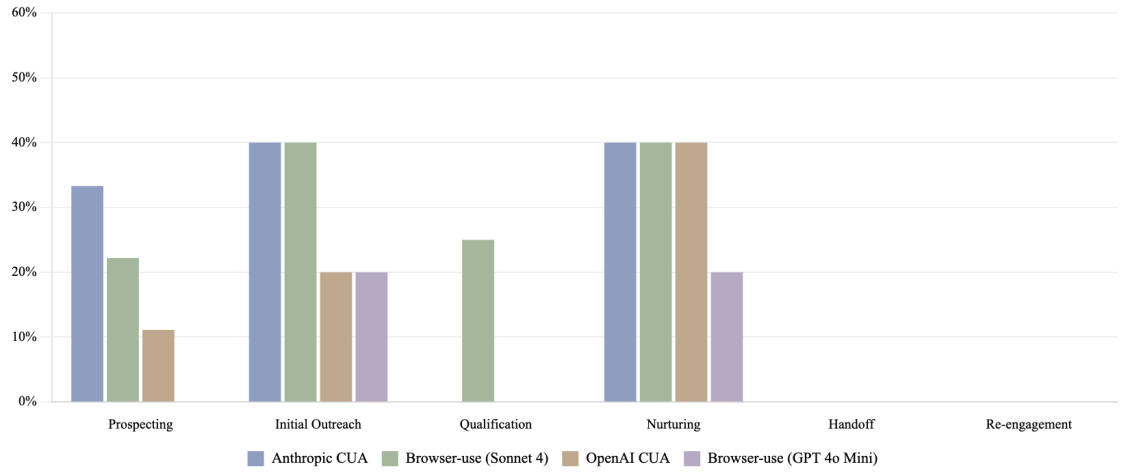


Figure 2: Success Rates by Activity Type

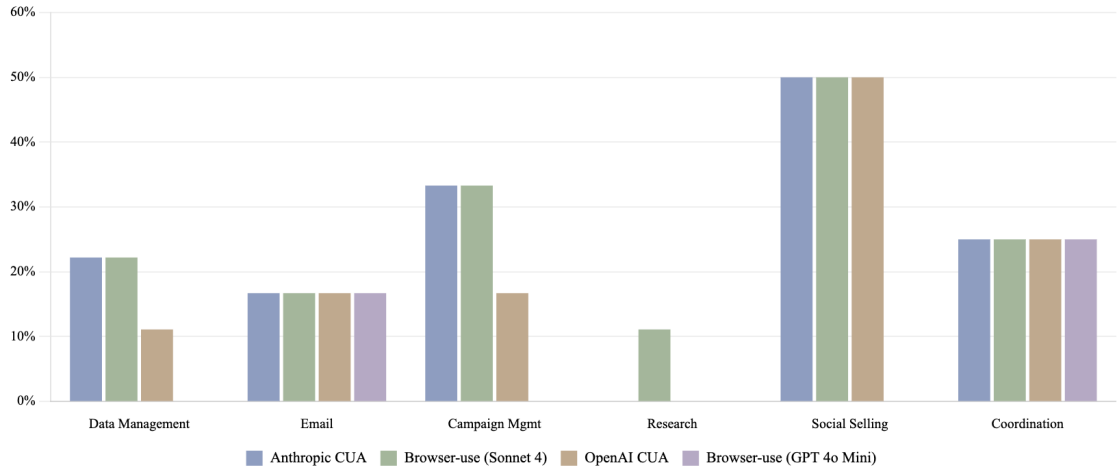


Figure 3: Success Rates by Platform

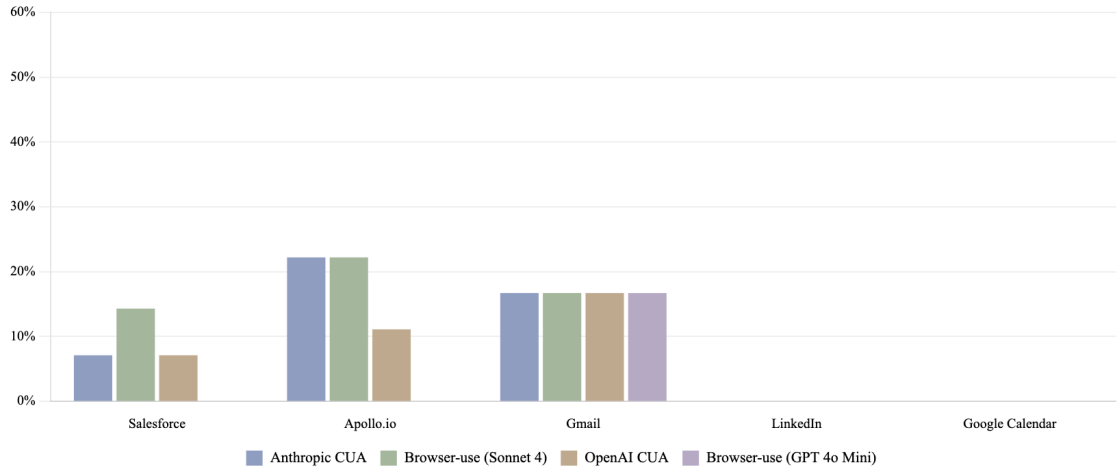
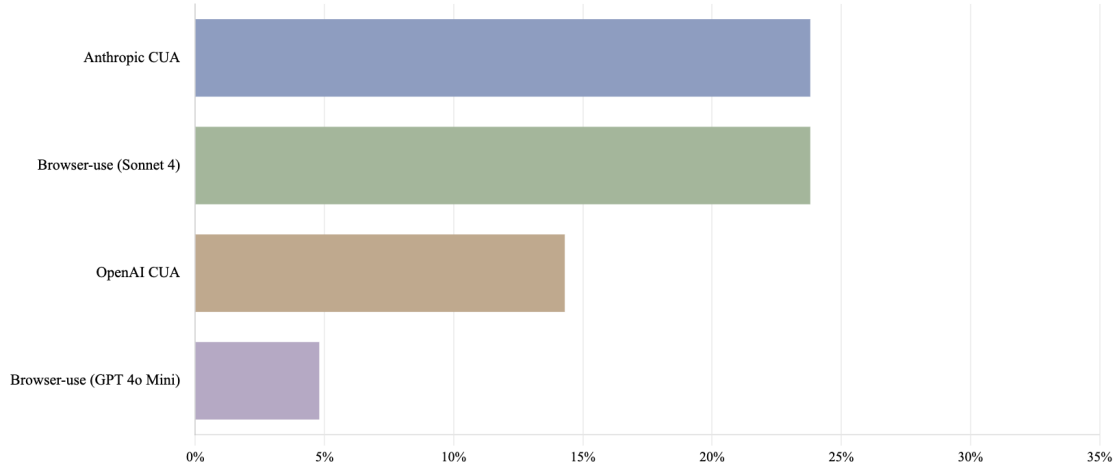


Figure 4: Overall Agent Comparison



Our evaluation reveals several important insights about current agent capabilities in SDR task automation:

- Anthropic CUA and Browser-use (Sonnet 4) achieved the highest success rate at 23.8%, demonstrating superior performance in SDR task automation.
- Anthropic CUA demonstrated the most efficient runtime at 554 seconds average, combining high success rate with speed.
- Browser-use (GPT 4o Mini) showed limited effectiveness with only 4.8% success rate and the longest runtime (922s).
- The success rates indicate that agent architectures are becoming more adept at handling SDR operational tasks.
- Approximately 67% of tasks remain unsolved, indicating substantial room for improvement in agent capabilities.

10 Behavior analysis

We present a comprehensive behavioral analysis of four browser automation agents evaluated across a random sample of 19 execution instances each, totaling 76 agent runs. The analysis is based on action distribution data, execution metrics, and observed behavioral patterns during automated web interaction tasks.

The analysis is conducted using quantitative data collected from agent execution logs, including action counts, timing metrics, and behavioral observations. Each agent was evaluated across 19 distinct execution scenarios, with detailed tracking of eight primary action types: click, mouse_move, scroll, type, mouse_up, mouse_down, double_click, and right_click. Agent behaviors were analyzed through statistical distributions, execution efficiency metrics, and qualitative observations from replay analysis.

10.1 Agent-Specific Behavioral Analysis

10.1.1 Anthropic CUA (Claude Sonnet 4)

Execution Profile: 19 executions, 5,666 total actions, 298.2 average actions per execution

Action Distribution

- Type: 52.2% (2,956 actions) – Highest typing proportion among CUA agents
- Click: 21.6% (1,222 actions) – Balanced clicking behavior
- Mouse Move: 9.3% (525 actions) – Moderate navigation
- Scroll: 2.5% (140 actions) – Limited but targeted scrolling
- Mouse Up/Down: 7.1% each (404 actions) – Consistent interaction patterns

Behavioral Characteristics Anthropic CUA demonstrates a text-heavy interaction paradigm, dedicating over half of its actions to typing activities. Replay analysis indicates frequent micro-scrolling that can produce long scrolling sequences leading to timeouts or step-budget exhaustion, suggesting suboptimal viewport management despite fast completion times.

Notable Execution Patterns Analysis of individual executions reveals significant variance in action counts (range: 98–799 actions), indicating adaptive behavior based on task complexity. The agent shows consistent preference for keyboard-based interactions over mouse-heavy approaches.

10.1.2 OpenAI CUA

Execution Profile 19 executions, 5,737 total actions, 301.9 average actions per execution

Action Distribution

- Type: 37.9% (2,173 actions) – Moderate typing focus
- Click: 20.6% (1,183 actions) – Balanced clicking approach
- Mouse Move: 14.7% (846 actions) – Highest mouse movement ratio among all agents
- Scroll: 0.4% (23 actions) – Minimal scrolling usage
- Mouse Up/Down: 13.2% each (756 actions) – Elevated interaction frequency

Behavioral Characteristics OpenAI CUA exhibits the highest mouse movement activity among all evaluated agents, with 14.7% of actions dedicated to cursor positioning. Replay analysis reveals coordinate-persistence: repeated clicking at imprecise positions without exploring alternatives, often beginning around step 20 and continuing until timeout or step-budget exhaustion.

Notable Execution Patterns The agent shows remarkable consistency in execution length but displays problematic coordination accuracy that can significantly impact task completion rates when precision clicking is required.

10.1.3 Browser Use (GPT-4o Mini)

Execution Profile 19 executions, 13,057 total actions, 687.2 average actions per execution

Action Distribution

- Type: 47.3% (6,174 actions) – High text input emphasis
- Click: 21.7% (2,839 actions) – Highest absolute click count
- Mouse Up/Down: 10.7% each (1,400 actions) – Substantial interaction volume
- Mouse Move: 7.8% (1,019 actions) – Moderate cursor management
- Scroll: 1.7% (222 actions) – Limited scrolling usage

Behavioral Characteristics Browser Use (GPT-4o Mini) demonstrates the most action-intensive approach among all agents, executing 687.2 actions per execution on average—more than double that of other agents. This extensive action profile suggests either highly thorough task exploration or potential inefficiencies in execution strategy. The agent maintains a typing-focused approach while demonstrating the highest absolute click frequency, indicating comprehensive interaction with web elements.

Notable Execution Patterns Individual execution analysis reveals extreme variance (range: 109–2,188 actions), with some executions requiring over 20 times more actions than others, suggesting significant task complexity sensitivity or potential execution inefficiencies.

10.1.4 Browser Use (Claude Sonnet 4)

Execution Profile 19 executions, 5,922 total actions, 311.7 average actions per execution

Action Distribution

- Type: 63.8% (3,776 actions) – Highest typing proportion across all agents
- Click: 18.9% (1,121 actions) – Lowest clicking percentage
- Mouse Up/Down: 5.0% each (296 actions) – Minimal interaction overhead
- Mouse Move: 4.9% (290 actions) – Lowest mouse movement among all agents
- Scroll: 2.3% (135 actions) – Moderate scrolling usage

Behavioral Characteristics Browser Use (Claude Sonnet 4) exhibits the most keyboard-centric approach among all evaluated agents, dedicating nearly two-thirds of actions to typing activities. The agent demonstrates exceptional efficiency in mouse usage, with the lowest mouse movement percentage (4.9%) and minimal clicking relative to other agents. This behavior profile suggests a highly deliberate, text-focused interaction strategy that minimizes unnecessary cursor movements.

Notable Execution Patterns The agent shows good execution consistency with moderate variance (range: 71–834 actions) and appears to optimize for direct interaction paths rather than exploratory behaviors.

10.2 Comparative Analysis

10.2.1 Behavioral Clustering

- **Text-Dominant Agents**: Browser Use (Claude Sonnet 4) and Anthropic CUA prioritize keyboard interactions, with typing comprising 63.8% and 52.2% of actions respectively.
- **Mouse-Active Agents**: OpenAI CUA demonstrates the highest mouse movement activity (14.7%), while Browser Use (GPT-4o Mini) shows the highest absolute action volume.
- **Efficiency Spectrum**: Action efficiency ranges from highly efficient (Anthropic CUA: 298.2 actions/execution) to resource-intensive (Browser Use GPT-4o Mini: 687.2 actions/execution).

10.2.2 Action Efficiency Comparison

Agent	Avg Actions/Execution	Typing %	Mouse Movement %	Clicking %
Anthropic CUA	298.2	52.2%	9.3%	21.6%
OpenAI CUA	301.9	37.9%	14.7%	20.6%
Browser Use (GPT-4o Mini)	687.2	47.3%	7.8%	21.7%
Browser Use (Claude Sonnet 4)	311.7	63.8%	4.9%	18.9%

Table 6: Action efficiency comparison across agents.

Table data corresponds to values shown in figures 2–5.

10.3 Visual Analyses

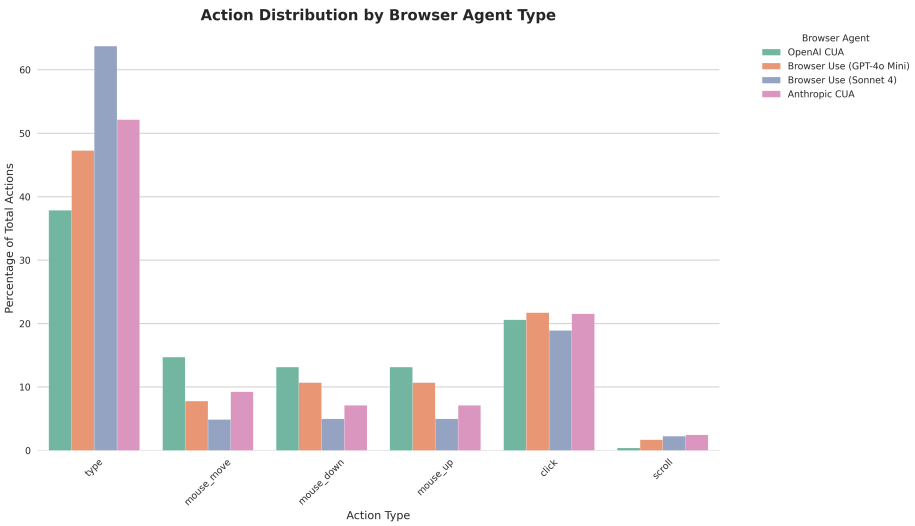


Figure 2: Distribution of action types per agent, contrasting text-centric and mouse-active strategies.

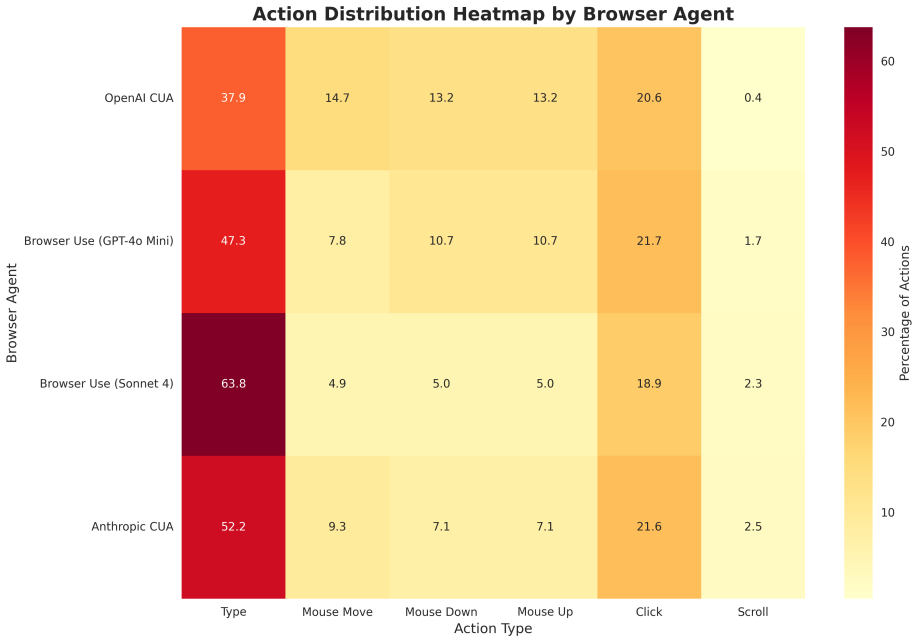


Figure 3: Heatmap of action-type densities across agents, highlighting behavioral clusters and outliers.

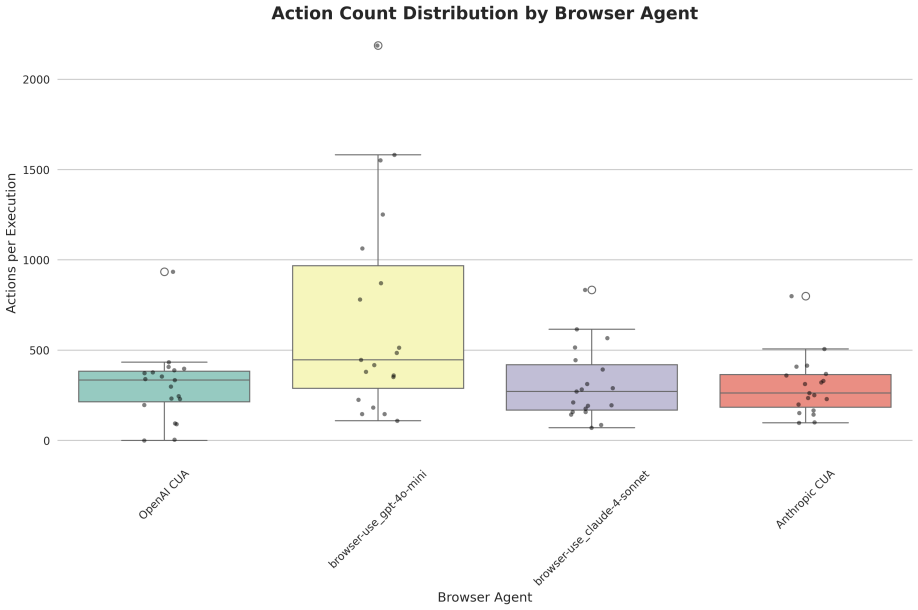


Figure 4: Distribution of total actions per execution; GPT-4o Mini exhibits substantially higher action counts.

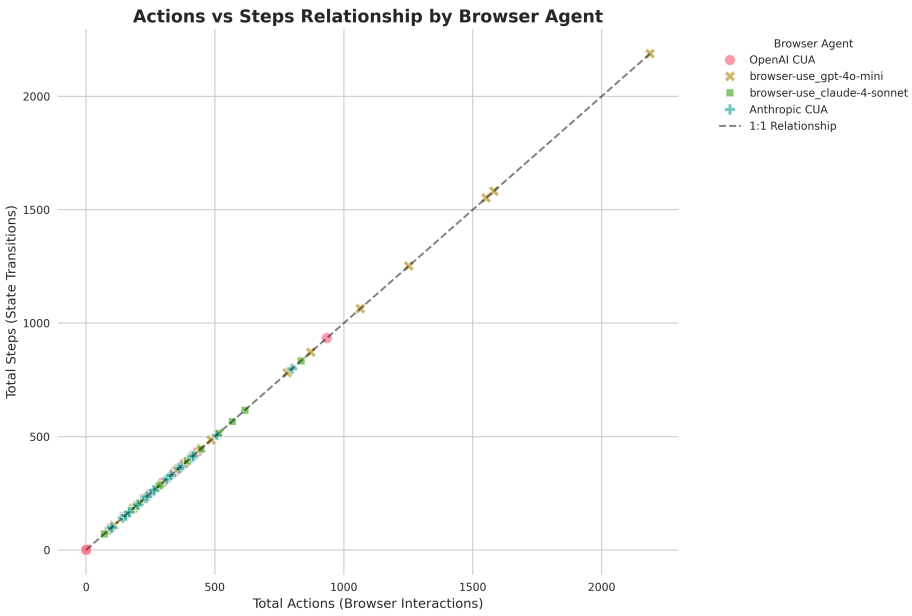


Figure 5: Relationship between total actions and execution steps, revealing efficiency patterns and outliers.

10.4 Interaction paradigms

Agents exhibit distinct strategies: text-centric approaches (Browser Use (Claude Sonnet 4), Anthropic CUA) versus mouse-active strategies (OpenAI CUA), with Browser Use (GPT-4o Mini) employing a comprehensive but potentially inefficient approach.

10.5 Precision vs. persistence

OpenAI CUA's coordinate-persistence behavior constitutes a major failure mode in precision-required tasks; Anthropic CUA's micro-scrolling reflects a speed–accuracy trade-off.

10.6 Execution variance as a signal

High execution variance (GPT-4o Mini) may indicate either broad exploration or inefficiency, whereas lower variance suggests stable but potentially inflexible strategies.

10.7 Scrolling strategies

Anthropic CUA's frequent small scrolls contrast with other agents' minimal scrolling, indicating different viewport management policies with differing effectiveness.

10.8 Efficiency disparities

Average actions per execution differ by 2.3× between the most and least efficient agents, underscoring significant optimization headroom.

10.9 Task suitability

Text-dominant agents appear favorable for form-heavy workflows; precision limitations in some agents reduce reliability for complex pointer-driven navigation.

11 Reinforcement Learning on Web environments

Training autonomous agents to perform complex tasks on real-world web applications poses significant challenges. In open web environments, the content and state of the interface can change unpredictably, making it difficult for an agent to receive consistent feedback. Prior benchmarks have noted that real websites "lack determinism, with constantly changing data and content... making reproducible evaluation nearly impossible," and that it is difficult to define clear reward signals for agent training in such settings [13]. To address this, we propose a framework where **reproducible complex environments** (e.g., web applications like Apollo.io, LinkedIn, Salesforce CRM, Gmail, Google Calendar, Google Docs, Google Sheets) are instrumented with **deterministic, rule-based annotation evaluations**. In other words, each task in these environments comes with an automated *validator* that checks the outcome against specified success criteria. This yields a reliable reward signal for the agent, enabling effective reinforcement learning (RL).

By having a deterministic and frozen environment (using foundry's offerings) and programmatically evaluating outcomes, we create a controlled setting for training web agents. The agent can interact with a web application (clicking buttons, filling forms, etc.), and after each attempt, the validator deterministically judges success or failure based on rules. This consistent feedback allows the agent to learn from experience in a manner similar to traditional RL environments like games or simulators, but now applied to high-level web tasks. Crucially, the deterministic nature of the environment and reward function mitigates the noise that would otherwise plague learning in a live, unpredictable web setting.

Our contributions in this section are summarized as follows:

1. **Deterministic Web Task Environment:** We present a methodology to transform real web-based applications into reproducible environments with fixed initial states and rule-based outcome evaluation. By providing programmatic validators for each task, we ensure *robust, reproducible evaluation* of agent performance [13]. This framework eliminates external noise and enables fair comparisons across runs.

2. **Effective RL with Rule-Based Rewards:** We demonstrate that reliable binary or scalar rewards from our annotation heuristics enable effective RL algorithms—from simple hill-climbing optimization to advanced policy gradient methods—to train agents on complex web tasks. The agent's goal is formalized as maximizing a reward (e.g., success rate), and even basic search methods can iteratively improve the policy given the consistent reward signal. Our approach thus bridges the gap between high-level web tasks and *reinforcement learning* formulations.

3. **Theoretical Integration of Efficient RL (GRPO):** We discuss how state-of-the-art RL techniques like *Group Relative Policy Optimization (GRPO)* can leverage our setup. Since our reward model is deterministic and well-defined, algorithms that rely on relative comparisons of outcomes (like GRPO) are a natural fit. We highlight that GRPO—a recent policy optimization approach that foregoes a traditional value baseline in favor of group-wise reward ranking—can further improve sample efficiency [15]. In fact, GRPO has been adopted in large-scale LLM training to cut RL training costs [15], and we theorize its benefits would similarly accrue in our rule-based setting by stabilizing training with low-variance advantage estimates.

Together, we aim to show that *deterministic, rule-based reward annotation* is a powerful enabler for training agents on realistic web tasks. In the following, we survey related work, detail our framework for creating reproducible web environments, and illustrate how various RL algorithms (hill climbing, policy gradients, GRPO) can be applied effectively in this context.

11.1 Web Environments

As large language models (LLMs) and other agents are increasingly tasked with operating web interfaces, various researchers and founders (such as Foundry) have begun creating dedicated web-based RL environments. *WebArena* is one example of such environment that provides "fully functional websites" in four domains (e-commerce, forums, etc.) with an emphasis on realism and reproducibility [1]. WebArena introduces tasks and uses *programmatic validators* to check functional correctness of task completion [1]. This ensures that an agent's actions are evaluated based on whether the desired outcome was achieved (for example, whether an item was successfully added to a shopping cart), rather than merely comparing against a recorded sequence of actions [1].

Similarly, Foundry's effort on *SDRBench* goes further by building deterministic high-fidelity environments of sales domain websites. In AWE, similar to REAL benchmark, all data and timestamps are fixed to remove randomness, and researchers can inspect state changes at any step to define reward signals [13]. The evaluation in Foundry's system combines **programmatic state checks** for all tasks and scores them based on high-quality rule-based heuristics that apply to the task. Due to the granularity of the captured application state, it is remarkably easy to define such deterministic heuristic checks and score models.

These efforts underscore a key point: By controlling the environment and using rule-based evaluations, we obtain **reliable rewards and reproducible conditions** that are essential for effective RL training. Our work builds on this principle, applying it to real web applications like Gmail or Salesforce by creating sandboxed deterministic scenarios for each task.

11.2 Challenges in Uncontrolled Environments

Without determinism, training an agent on web tasks becomes extremely difficult. If the page content changes (due to live updates or shifting data) or if network latency causes flaky behavior, the agent could receive inconsistent rewards for the same actions, impeding learning. Prior works have noted that production websites cannot easily be used for RL because you cannot reset them or ensure the same conditions each time, and an agent's actions might irreversibly change the state (buying items, sending messages) [13].

By contrast, a controlled environment avoids these issues. The importance of this is evident in the performance gap: even the strongest GPT-4 based agents achieve relatively low success rates (e.g., approximately 14% on WebArena tasks) when evaluated in realistic settings [1]. The brittleness of agents in uncontrolled web environments is precisely what our framework aims to overcome by providing a *training ground* where consistency and repeatability are guaranteed.

11.3 Rule-Based Annotation in Reproducible Web Environments

Our methodology has two primary components: **(a) Environment Reproducibility**, and **(b) Rule-Based Reward Annotations**. These combine to form a Markov decision process (MDP) for each web task, enabling the direct application of RL algorithms.

11.4 Environment Design and Reset Mechanism

For each target application (Apollo.io, LinkedIn, Salesforce, Gmail, Google Calendar, Google Docs, Google Sheets), we construct a dedicated environment that can be reliably reset to a known initial state. In some cases, this involves using sandbox or developer modes of the applications; in others, it involves recreating the application with a high-fidelity mock, as described by 7. Inspired by the REAL benchmark's use of "deterministic replicas" of websites [13], we take care to fix all sources of randomness. This includes seeding any date/time fields, using test data that remains constant between runs of that particular task.

For example:

- **Salesforce (CRM platforms):** We use developer sandboxes (Scratch Orgs [12]) populated by a deterministic set of entities e.g. leads, accounts, contacts, campaigns and so on. Before each episode, any changes made by the agent (such as updating a lead status) are rolled back or the sandbox is reinitialized. This ensures that the next run sees the exact same list of (let's say) leads with the same properties, allowing fair evaluation.
- **LinkedIn (professional network):** For tasks like sending a connection request or messaging a user, we utilize a mock network of profiles, based on patterns of actual data on Linkedin. Each profile's state (connection lists, message inbox) is reset to baseline for each trial.
- **Google Calendar:** We give agents a google calendar clone with controlled data (e.g., a calendar with no events). By using the clone's abstraction for state manipulations, we can clear all events between episodes. Thus, if the task is "schedule a meeting and send an invite," we guarantee that prior to the agent's attempt, there are no pre-existing events that could confound the outcome.

The **reset mechanism** is critical. We implement it either as part of the environment, or in case of sandbox versions, automated scripts (e.g., API calls to clear data, or containerized services that are re-deployed). The end result is that the agent always faces a *known initial state*.

Formally, for each task we define an initial state s_0 and a deterministic state transition function $T(s, a) = s'$ dictated by the web interface logic (since we remove external nondeterminism, T is as predictable as a game engine). The agent's actions a are low-level (click, type) or high-level (semantic commands that our system maps to low-level events), and yield a new state s'. An episode ends when the task is completed or a time/step limit is reached.

11.5 Rule-Based Reward Annotation

Parallel to environment design, we develop a **rule-based evaluator** for each task. This is essentially a script or set of checks that inspects the final state (and possibly intermediate states) to decide whether the task was successfully accomplished. Our evaluators are analogous to unit tests for the task outcome.

For example:

- In a Gmail sending task, the evaluator will check the "Sent" folder (or an outgoing email log) for the presence of an email with the expected subject and recipient. If found, the task is marked successful (reward = 1); if not, it's a failure (reward = 0).
- In a Salesforce record update task, the evaluator similarly checks the state representation of Salesforce for the record after the agent's actions. If the field (e.g., "Lead Status") matches the target value specified by the task, it returns success. If the agent entered an incorrect value or failed to save, it returns failure.

- In a Google Sheets computation task (say the task is "calculate the sum of a column"), the evaluator will retrieve the cell (from csv) that should contain the sum and compare its value to the ground truth sum of that column's entries. The reward function could even assign a partial score based on how close the answer is, though in tasks for RL, we intend to use binary success/failure for clarity.

These checks are deterministic programs, often straightforward if-else conditions or string/number comparisons. Thanks to the controlled environment, there is no ambiguity: the correct outcome is known and any deviation can be detected. We emphasize that our evaluation rules focus on **functional correctness** rather than how the task was done. This philosophy is shared by WebArena's benchmark which validates outcomes rather than action sequences [1].

Formally, we can define a reward function $r(s_t, a_t, s_{t+1})$ that is zero for all intermediate steps unless the task concludes, and yields a terminal reward at the end: for final state s_T, $r(s_T) = 1$ if success criteria are met, else 0. The total return $R(\tau)$ for an episode τ is simply this success indicator. Training the agent then boils down to maximizing the expected success rate:

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\big[R(\tau)\big] = \Pr(\text{success}|\pi_\theta). \tag{1}$$

Here π_θ is the policy (with parameters θ) controlling the agent's actions. Because the environment is episodic and deterministic, $\Pr(\text{success}|\pi_\theta)$ is directly influenced by whether π_θ chooses the correct action sequence.

We ensure that evaluating the reward function is efficient. In many cases, our validator runs in negligible time (checking a few fields of the application state). This means we can afford to evaluate many episodes, which is important for RL training. The consistency of the reward is guaranteed by the deterministic environment i.e. the same action sequence from s_0 will always produce the same outcome and hence the same reward.

12 Reinforcement Learning Approaches

With the above MDP formulation in place, we can apply a spectrum of RL and black-box optimization techniques. In this section, we discuss two categories: (1) simple heuristic optimization (hill climbing), and (2) advanced policy gradient methods (with a focus on GRPO).

12.1 Hill Climbing and Evolutionary Strategies

One of the most intuitive approaches to improve an agent is **hill climbing**. In the context of training a policy, a hill climbing algorithm would start with an initial policy (which could be random or a baseline like a supervised model) and then repeatedly make small adjustments, accepting those that improve performance and discarding those that do not. Because our reward signal is deterministic and indicative of success rate, we can measure the performance of a given policy reliably by running a batch of episodes and computing the average reward.

In practice, a hill-climbing RL procedure might work as follows: we sample a set of random perturbations to the policy parameters θ (or to the policy output if using some fixed function approximator), evaluate each variant on a number of episodes, and then move θ in the direction of the best-performing variant. This is akin to an *evolutionary strategy* or cross-entropy method, where the "fitness" of each policy variant is the success rate on the task.

Hill climbing is essentially doing gradient-free optimization on the expected reward $J(\pi_\theta)$. While it may be less sample-efficient than gradient-based methods, its simplicity is attractive, and it can

work surprisingly well when rewards are sparse but consistent. Notably, Guo et al. found that large language models could themselves carry out hill-climbing optimization of solutions when provided consistent metrics [16].

12.2 Policy Gradient and GRPO

Beyond simple heuristics, we recommend leveraging policy gradient algorithms to directly optimize $J(\pi_\theta)$. In policy gradient methods, we compute an estimate of $\nabla_\theta J(\theta)$ and use it to update the policy parameters in a gradient ascent framework. A classic example is the REINFORCE algorithm, which in our case would use episodes τ and the reward $R(\tau)$ to push the policy toward higher-reward trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T_i} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \, R(\tau^{(i)}), \tag{2}$$

3: where $\tau^{(i)}$ is the i-th sampled trajectory.

While REINFORCE and simple policy gradient work, they often suffer from high variance in complex tasks. Traditionally, one would use a **critic network** or value function to provide a baseline (leading to algorithms like Actor-Critic or PPO) to stabilize training. However, training a value function in a sparse-reward environment can be tricky—it needs to learn to predict the probability of success from a given state.

This is where **Group Relative Policy Optimization (GRPO)** comes into play. GRPO is a newer variant of policy optimization that *replaces the learned value baseline with a group-based empirical baseline*. Instead of relying on a value network to estimate how good a state is, GRPO draws multiple action samples for the same state and uses their *relative* rewards to determine which actions were better [15].

In practice, for each state encountered during training, we sample a small group of possible actions (or continuations) from the current policy. We then evaluate all of those actions using our reward model and rank the actions by reward. We compute advantages by comparing each action's reward to the mean reward in the group [15]. The policy is then updated to favor actions with above-average reward and disfavor those below average.

Our deterministic evaluation is an ideal setting for GRPO. Since we can *truly measure which action is better*, we fulfill the requirement of GRPO that only needs "a means to verify correctness and order" of outcomes rather than ground-truth numeric rewards from human labels [15]. In the DeepSeek-R1 project, the authors adopted GRPO to improve training efficiency [15].

In our context, we can imagine using GRPO as follows: when the agent is in a particular state (e.g., it has filled some fields in a form and is deciding on the next action), we can sample multiple possible next actions from the policy. We then simulate each to the end of the task and collect the success/failure outcome. The policy update would then push the policy to increase probability of successful actions in that state. Foundry's deterministic, resettable and rewindable state system through AWE makes it possible to do so.

Mathematically, GRPO introduces a modified objective. If we denote by $\{a^{(k)}\}_{k=1}^{N}$ the group of N actions sampled for a state s, with rewards $R^{(k)}$ for each, then one simple form of the advantage for action k is:

$$A^{(k)} = R^{(k)} - \frac{1}{N} \sum_{j=1}^{N} R^{(j)}, \tag{4}$$

i.e., reward minus the group average [15].

12.3 Discussion: Why Determinism Matters for RL

It is worth reflecting on why all these methods; from hill climbing to GRPO are especially effective given our deterministic, rule-based setup. Traditional RL struggles when rewards are sparse and stochastic. High variance in returns means an agent needs many trials to discern whether one policy is better than another. By contrast, in our framework:

- If an agent policy has a 40% success rate, any evaluation of enough episodes will reflect that exactly, with very little noise (aside from the binomial variance which is quickly averaged out over multiple runs). There are no random failures unrelated to the agent.
- Credit assignment is clearer because we can instrument the environment, we could even provide intermediate rewards if needed. This is only possible because we have full knowledge of the task structure via our rules.
- We can run A/B tests on policies with confidence. If policy π_A outperforms π_B in our evaluation, we know π_A is truly better for the task.
- Since tasks are annotated and modular, we can train on simpler tasks and then use those skills on harder tasks, with the assurance that the evaluation for each task remains consistent.

In essence, our approach turns the problem of web automation via RL into something much more akin to solving a deterministic game or puzzle. The complexity of the web interface is still there, but we remove extraneous uncertainty.
In summary, deterministic rule-based annotation for web tasks is a powerful tool for reinforcement learning. It provides the necessary foundation for *reliable reward modeling* [13] and efficient algorithms like GRPO to shine. We hope that this approach will spur further research into autonomous agents that can safely and effectively interact with the software tools and web services that permeate modern life.

13 Discussion and Limitations

SDRBench was designed to balance two competing goals: (1) high-fidelity enterprise realism, and (2) controlled, reproducible evaluation. Our design choices reflect this tradeoff. The mocked environments allow for deterministic runs and measurable outcomes, but they inevitably simplify the production setting. For example, our email/calendar simulations even though high fidelity are pared-down versions of full enterprise capabilities, and our task set, while representative, cannot fully capture the diversity of real-world SDR workflows. This section discusses the challenges we encountered while building SDRBench, methodological limitations in our approach, and directions for future work.

13.1 Technical Challenges

Building the agent execution pipeline and running this benchmark surfaced several non-trivial issues that shaped the final design.

13.1.1 Navigation & Environment Containment

Early agent runs frequently attempted to access live websites (e.g., Google) to reach the target environments or manually typed URLs into the page, occasionally triggering CAPTCHAs or ending up on non-mocked environments. We prevented this via the following changes:

- Crafted explicit system prompts specifying allowed environments and disallowing arbitrary navigation.
- Provided structured navigation tools (`goto()`, `back()`, `forward()`) to enforce deterministic transitions.

13.1.2 Slow Page Loads & Agent Timeouts

Environments such as Salesforce exhibited slow load times, causing Playwright `goto()` calls to exceed default timeouts and terminate runs. This was countered by:

- Adding lightweight *loading pages* in the mocked environments to give Playwright a stable DOM target and ensure that navigation only completed once the target page was ready.
- Modifying agent SDKs to include a `max_timeout` parameter for navigation calls.

13.1.3 SDK Gaps & Model Integration

Out-of-the-box agent SDKs were not designed for a controlled, reproducible benchmark harness and required heavy modification:

- **Amazon Nova Act:** The Nova SDK lacked robust CDP session handling, frequently losing browser context between acts. Navigation calls often timed out without retrying, causing premature run terminations.

  Fix: We forked and patched the SDK to retain CDP sessions across acts, add retry logic for `goto()` and other navigation functions, and DOM queries, and gracefully handle context loss. Even after patching, Nova Act remained brittle — unhandled timeouts and CDP disconnects still caused occasional fatal errors.

- **Skyvern:** Skyvern was originally designed to run as a service with its own Dockerized backend and database for storing runs and workflows, which was incompatible with our lightweight, ephemeral setup. Additionally, it could not connect to an externally managed Playwright CDP session.

  Fix: We modified the SDK to bypass its backend service entirely, reuse our benchmark database and ports, and connect directly to our local Playwright session. We also resolved multiple dependency conflicts and patched import errors. Despite these efforts, Skyvern's step-by-step DOM scraping remained slow and often re-triggered already-completed actions, reducing overall performance.

- **Browser Use:** Browser Use was unstable on newer Playwright versions. Playwright had begun to natively handle certain events (e.g., unload dialog boxes), but Browser Use attempted to intercept them manually, causing race conditions and frequent agent crashes.

  Fix: We pinned Playwright to a stable version and patched Browser Use's event-handling logic to avoid redundant interception.

13.1.4 State Reset & Environment Management

Ensuring that each run was fully deterministic required resetting both browser state and backend state before every episode. This was particularly challenging with Salesforce, since we were not mocking its HTML but using live scratch orgs.
Challenges:

- Resetting Salesforce state was slow, requiring multiple SOQL queries and record mutations.
- We had to respect Salesforce governor limits, or risk exhausting API quotas mid-benchmark.

Solutions:

- Implemented scratch-org pooling, allowing reuse of pre-provisioned orgs instead of provisioning a new org for every run.
- Cached relevant records and performed targeted SOQL resets, minimizing the number of API calls required per episode.

13.2 Limitations in Our Approach

While we aimed to create a comprehensive and rigorous benchmark for evaluating browser agents on SDR workflows, we acknowledge several methodological shortcomings in our development process that may have impacted the quality and completeness of SDRBench.

13.2.1 Insufficient Automated Quality Assurance Processes

Our initial development approach lacked systematic automated quality checks for ground truth data, heuristics, and state verification logic. This led to substantial time loss when discrepancies between prompts, expected behaviors, and validation heuristics were discovered retroactively during agent testing. Several evaluation predicates required patching after benchmark runs revealed inconsistencies, potentially introducing bias toward agents that were tested with the corrected versions. A more rigorous upfront validation protocol with automated consistency checks across task descriptions, ground truth annotations, and evaluation logic would have prevented these costly iterations through manual interventions.

13.2.2 Incomplete Experimental Scope

Due to development timeline constraints, several critical components of real SDR workflows were deferred for future releases. Most notably, LinkedIn Sales Navigator—a cornerstone tool for professional SDR work—was not included in the initial benchmark version. While we have the capability to integrate premium enterprise tools through our Agent Web Engine architecture, prioritizing core infrastructure development and baseline task coverage meant postponing these advanced integrations. This pragmatic scoping decision, driven by release timeline pressures creates a gap between our current simulated environment and the full spectrum of production SDR operations. The absence of such premium tools in this initial release may simplify certain aspects of real-world prospecting workflow complexity, though our modular architecture explicitly enables their integration in subsequent versions.

13.2.3 Resource-Driven Design Decisions

Our evaluation methodology prioritizes deterministic, programmatic verification of state changes over qualitative assessments of agent behavior. While this choice enhances reproducibility, it was partly driven by resource constraints that prevented implementing more sophisticated evaluation protocols (e.g., LLM-based assessment of email quality, UI interaction efficiency metrics, or human evaluation studies). This may lead to overestimating performance for agents that technically complete tasks while exhibiting inefficient or unnatural behaviors.

Our initial evaluation covers only three representative agent families, which, while providing valuable initial insights, does not capture the full spectrum of approaches in the browser automation landscape. The absence of specialized RL-trained agents, commercial automation systems, and human baseline comparisons limits our ability to contextualize the benchmark's difficulty and the relative performance of different architectural choices.

13.2.4 Implications and Mitigation Strategies

These methodological shortcomings suggest several important caveats for interpreting SDRBench results. First, reported agent performance may be optimistic relative to real-world deployment scenarios due to our simplified environment and evaluation approach. Second, the benchmark may not yet provide sufficient signal for certain research directions, particularly reinforcement learning approaches. Third, task-specific performance variations should be interpreted cautiously given our incomplete quality validation.

To address these issues, the future work includes: (1) comprehensive automated testing frameworks for ground truth validation, (2) systematic comparison with human performance baselines, (3) integration of premium tools and services where feasible, (4) expanded baseline coverage including RL-trained models. We view SDRBench as a living benchmark that will evolve through community contributions to address these initial shortcomings while maintaining its core value as a reproducible testbed for long-horizon browser automation research.

13.3 Future Improvements

Building on *SDRBench* as a controlled and reproducible evaluation platform for enterprise browser agents, we prioritize three focused themes for the next iteration: (i) *Validation & Quality Assurance*, (ii) *Expanded Coverage (Tools & Tasks)*, and (iii) *RL Training & Infrastructure*. These directions sharpen rigor, broaden scope, and strengthen the benchmark's utility for both evaluation and training research.

13.3.1 Validation & Quality Assurance

We will harden evaluation fidelity and release discipline through automated and human-in-the-loop validation:

- Enforce end-to-end consistency between task specifications, ground-truth annotations, and evaluation predicates via automated checks at authoring time and regression gates on every update.
- Build robust, deterministic test suites—including targeted edge-case and mutation tests—with change-impact analysis to reveal brittle predicates prior to release.
- Establish human reference trajectories and conduct inter-rater reliability audits to quantify annotation quality and provide realistic performance ceilings for contextualizing agent results.
- Maintain versioned datasets and evaluators with semantic changelogs and a clear deprecation policy to preserve longitudinal comparability.

13.3.2 Expanded Coverage (Tools & Tasks)

We will broaden enterprise realism while preserving determinism and reproducibility:

- Integrate high-value enterprise tools (e.g., LinkedIn Sales Navigator, advanced Salesforce features) when licensing permits, or supply high-fidelity mocks that preserve workflow complexity and constraints when direct access is infeasible.
- Author richer, long-horizon, cross-application workflows that include multi-threaded email exchanges, calendar conflict resolution, spreadsheet transformations, and CRM automations with cascading state dependencies.
- Scale the task suite toward 1000+ items—targeting 150–200 long-horizon tasks—while retaining deterministic state-delta evaluation to ensure fair and reproducible comparisons.

13.3.3 RL Training & Infrastructure

We will establish *SDRBench* as a credible training ground for policy learning with reproducible, scalable experiments:

- Run empirical baselines that characterize learning curves and sample efficiency across on-policy and off-policy algorithms, including group-based reinforcement strategies (e.g., GRPO) with rule-derived reward heuristics from deterministic predicates.
- Design curricula that exploit snapshot/rewind for staged skill acquisition and progressive task difficulty, using fixed seeds and replayable trajectories for rigorous ablations.
- Provision training-at-scale through parallel environment execution, scheduler-aware orchestration, and standardized logging/observability that capture action rationales and structured error taxonomies.
- Publish canonical metrics beyond success rate—such as interaction efficiency and navigation optimality—alongside unified experiment schemas and evaluator-versioned leaderboards for transparent comparison.

14 Conclusion

SDRBench provides a realistic, resettable RL gym for SDR workflows with deterministic, state-based evaluation and dense reward support. It complements existing web benchmarks by targeting multi-app enterprise tasks and enabling both *evaluation* and *training*. We hope SDRBench will catalyze research on planning, memory, error recovery, and robust tool use for real-world autonomy.

References

[1] WebArena: A Realistic Web Environment for Building Autonomous Agents. (Simulated multi-site web benchmark).

[2] WebVoyager: A Multimodal Agent for Real Websites. (Live-web, GPT-4V evaluator).

[3] Mind2Web: Towards a Generalist Agent for the Web. (Live sites/snapshots dataset).

[4] Skyvern/Halluminate. WebBench: Large-scale read/write tasks across 100s of live websites.

[5] WebShop: Towards Scalable Web-based Shopping Agents. (Single-site, programmatic eval).

[6] MiniWoB++: A Minimalist Web Benchmark for RL.

[7] Similar AI. Agent S2: Modular agent pipeline for computer use.

[8] OpenAI CUA / GPT-4 Computer-Use Agent.

[9] Anthropic Claude Computer Use API.

[10] Amazon Nova Act: Web action model and SDK.

[11] RRWeb: Web recording and replay library

[12] Salesforce Scratch Orgs: Stateless developer environments for testing CRMs

[13] REAL: Benchmarking autonomous agents on deterministic simulations of real websites.

[14] WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning.

[15] DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning.

[16] A systematic survey on large language models for evolutionary optimization: From modeling to solving.