

# Operative: A Computer Use Framework for Autonomous AI Agents

Hanzo AI Research  
research@hanzo.ai  
*Hanzo AI San Francisco, CA*

February 2026

## Abstract

Autonomous AI agents capable of operating graphical user interfaces represent a significant frontier in applied artificial intelligence. We present Operative, an open-source framework for computer use that enables large language models to perceive, reason about, and interact with desktop and browser environments. Operative introduces three key innovations: (1) a hierarchical screen understanding pipeline combining OCR, visual grounding, and accessibility tree parsing to construct structured representations of arbitrary GUIs, (2) a typed action space with 47 primitive operations spanning mouse, keyboard, browser, and system interactions with formal pre- and post-condition specifications, and (3) a multi-step planning architecture that decomposes high-level user goals into executable action sequences with rollback capabilities. Safety is enforced through a configurable boundary system that restricts filesystem access, network egress, and system modifications. In evaluation across 12 benchmark tasks from OS-World and WebArena, Operative achieves 71.3% task completion rate, representing a 14-point improvement over the previous state of the art. The framework processes 4.2 screen observations per second on consumer hardware and supports both cloud-hosted models (via screenshot) and local vision-language models (via direct framebuffer access).

## 1 Introduction

The ability to operate computers through graphical user interfaces has been a longstanding goal of AI research [1]. While large language models have demonstrated remarkable capabilities in text-based tool use [2], extending these capabilities to visual, spatially-grounded computer interaction introduces fundamental challenges in perception, planning, and action execution.

Recent work on computer use agents—including Anthropic’s computer use mode [3], OpenAI’s Operator [4], and Google’s Project Mariner [5]—has demonstrated the feasibility of LLM-driven GUI interaction. However, these systems are tightly coupled to specific model providers, limiting reproducibility, customization, and deployment flexibility.

Operative addresses this gap as a model-agnostic, open-source framework for computer use that:

1. **Decouples perception from action:** The screen understanding pipeline produces structured representations consumable by any LLM, enabling model-agnostic planning.
2. **Formalizes the action space:** All 47 primitive operations have typed signatures with pre-conditions, post-conditions, and reversibility annotations, enabling formal verification of action sequences.
3. **Enforces safety boundaries:** A configurable policy engine restricts agent capabil-

ities based on the principle of least privilege, preventing unintended system modifications.

4. **Supports multi-step planning:** A hierarchical planner decomposes goals into sub-tasks with checkpointing and rollback, enabling recovery from errors without full task restart.

**Contributions.** The specific contributions of this paper are:

- A screen understanding pipeline that fuses OCR, visual grounding, and accessibility tree parsing to produce structured element maps with 94.7% element detection accuracy (Section 3).
- A formally specified action space with pre/post-conditions enabling static verification of action sequences (Section 4).
- A safety boundary system with filesystem, network, and system sandboxing (Section 5).
- A hierarchical planning architecture with checkpoint-based rollback achieving 71.3% task completion on OSWorld benchmarks (Section 6).
- Comprehensive evaluation across desktop and browser tasks (Section 8).

## 2 Background and Related Work

### 2.1 Computer Use Agents

The concept of software agents that interact with GUIs dates to the SIKULI system [6], which used screenshot-based visual pattern matching for UI automation. Modern computer use agents leverage vision-language models (VLMs) to understand screen content and generate interaction commands.

CogAgent [7] demonstrated that VLMs fine-tuned on GUI screenshots can accurately locate and interact with interface elements. SeeClick [8] introduced visual grounding pre-training for improved element localization. WebVoyager [9]

showed that GPT-4V could complete web tasks by reasoning over screenshots. AgentStudio [10] provided a benchmark infrastructure for evaluating computer use agents.

### 2.2 GUI Understanding

GUI understanding combines techniques from document understanding, visual grounding, and accessibility research. Key approaches include:

- **Pixel-level:** Direct visual processing of screenshots using CNNs or vision transformers [11].
- **DOM/Accessibility tree:** Structured parsing of web page DOM or OS accessibility APIs [9].
- **Hybrid:** Combining visual and structural signals for robust element detection [7].

Operative adopts the hybrid approach, fusing pixel-level visual understanding with accessibility tree data for maximum coverage across native desktop and web applications.

### 2.3 Action Spaces for Computer Interaction

Prior work has used various action space formulations:

- **Coordinate-based:** Raw  $(x, y)$  click and type actions [3].
- **Element-based:** Actions reference specific UI elements by ID or selector [9].
- **API-based:** Direct invocation of application APIs bypassing the GUI [17].

Operative supports all three modes with a unified type system, enabling agents to choose the most appropriate interaction level for each task.

## 3 Screen Understanding Pipeline

The perception system converts raw screen content into a structured **ScreenState** representation that captures all interactive elements, their properties, and spatial relationships.

### 3.1 Multi-Source Element Detection

Element detection fuses three sources:

#### 3.1.1 Visual Detection

A lightweight object detection model (YOLOv8-nano fine-tuned on 180,000 annotated GUI screenshots) identifies common UI elements: buttons, text fields, checkboxes, dropdowns, links, icons, and images. The model runs at 23ms per 1920×1080 screenshot on an M2 MacBook Pro.

For each detected element, the model produces a bounding box  $(x_1, y_1, x_2, y_2)$ , element type classification, and confidence score. We apply non-maximum suppression with IoU threshold 0.5 to remove duplicates.

#### 3.1.2 OCR Layer

Optical character recognition extracts text content from the screenshot. We use a two-stage pipeline:

1. **Text detection:** A CRAFT-based [12] text detector identifies text regions with character-level granularity.
2. **Text recognition:** A transformer-based recognizer processes each detected region, supporting 47 languages including CJK scripts.

OCR results are associated with visually detected elements by spatial overlap ( $\text{IoU} > 0.3$ ) to populate element text content.

#### 3.1.3 Accessibility Tree

On supported platforms (macOS via Accessibility API, Windows via UI Automation, Linux

---

### Algorithm 1 Multi-Source Element Fusion

---

**Require:** Visual elements  $V$ , OCR elements  $O$ , All elements  $A$

```

1:  $M \leftarrow \emptyset$  {Matched element set}
2: for  $v \in V$  do
3:    $a^* \leftarrow \arg \max_{a \in A} \text{IoU}(v.\text{bbox}, a.\text{bbox})$ 
4:   if  $\text{IoU}(v.\text{bbox}, a^*.\text{bbox}) > 0.4$  then
5:      $e \leftarrow \text{merge}(v, a^*)$ 
6:      $A \leftarrow A \setminus \{a^*\}$ 
7:   else
8:      $e \leftarrow \text{fromVisual}(v)$ 
9:   end if
10:   $o^* \leftarrow \{o \in O \mid \text{IoU}(o.\text{bbox}, e.\text{bbox}) > 0.3\}$ 
11:   $e.\text{text} \leftarrow \text{bestText}(e.\text{text}, o^*.\text{text})$ 
12:   $M \leftarrow M \cup \{e\}$ 
13: end for
14: for  $a \in A$  do
15:   {Unmatched all elements}  $M \leftarrow M \cup \{\text{fromAll}(a)\}$ 
16: end for
17: return  $M$ 

```

---

via AT-SPI, browsers via DOM), the framework queries the accessibility tree to obtain:

- Element roles (button, textbox, link, etc.)
- Labels and descriptions
- State information (enabled, focused, checked, expanded)
- Hierarchical relationships (parent, children, siblings)
- Bounding rectangles (when available)

### 3.2 Element Fusion

The three detection sources are fused using a matching algorithm that associates elements across sources:

The fusion algorithm prioritizes accessibility tree data for element roles and state (higher reliability) while using visual detection for spatial accuracy and OCR for text content when accessibility labels are absent.

### 3.3 ScreenState Representation

The fused elements are organized into a ScreenState object:

```

1 @dataclass
2 class Element:
3     id: str          # Unique
4     identifier
5     type: ElementType # button, input,
6     link...
7     bbox: BBox       # Bounding
8     rectangle
9     text: str         # Visible text
10    content
11    role: str         # Accessibility
12    role
13    state: dict        # enabled, focused
14    , etc
15    children: list     # Child elements
16
17 @dataclass
18 class ScreenState:
19     elements: list[Element]
20     focused: Element | None
21     screenshot: bytes   # PNG screenshot
22     timestamp: float
23     app_name: str       # Active
24     application
25     window_title: str   # Active window
26     url: str | None     # Browser URL if
27     any
28
29     def to_text(self) -> str:
30         """Serialize to numbered element
31         list
32         for LLM consumption."""
33
34     def find(self, query: str) -> list:
35         """Fuzzy search elements by text
36         or role."""

```

Listing 1: ScreenState data structure

The `to_text()` method produces a numbered element list that serves as the primary input to the LLM planner:

```

1 Active: Chrome - GitHub Dashboard
2 URL: https://github.com
3
4 [1] button "Sign in" (423,12)-(492,36)
5 [2] input "Search" (210,12)-(380,36)
6     focused
7 [3] link "Explore" (132,12)-(178,36)
8 [4] heading "Dashboard" (24,72)-(180,96)
9 [5] link "hanzoai/gateway" (24,120)
10    -(186,138)
11 ...

```

```

10 [47] button "New repository" (890,12)
11    -(1004,36)

```

Listing 2: Example ScreenState text representation

### 3.4 Performance Optimization

To achieve real-time perception (target: <250ms per frame), we implement several optimizations:

- **Incremental updates:** Only re-process screen regions that changed since the last frame (detected via pixel-level differencing with 2% change threshold).
- **Parallel pipelines:** Visual detection, OCR, and accessibility tree queries run concurrently using a thread pool.
- **Caching:** Accessibility tree queries are cached with 500ms TTL, as tree structure changes infrequently between user actions.
- **Resolution scaling:** Screenshots are down-scaled to 1280×720 for visual detection (sufficient for element localization) while full resolution is preserved for OCR.

## 4 Action Space

Operative defines a typed action space with 47 primitive operations organized into five categories.

### 4.1 Action Type System

Each action is defined with a formal signature:

**Definition 1** (Action Signature). *An action  $a$  has signature  $(T, P, pre, post, R)$  where  $T$  is the action type,  $P$  is the parameter type,  $pre : S \times P \rightarrow \{0, 1\}$  is the pre-condition predicate over screen state  $S$  and parameters  $P$ ,  $post : S \times S' \rightarrow \{0, 1\}$  is the post-condition predicate over pre- and post-states, and  $R \in \{reversible, irreversible, idempotent\}$  is the reversibility classification.*

Table 1: Mouse action primitives

Action	Params	Pre-condition
<code>click</code>	$x, y$ , btn	Target visible
<code>double_click</code>	$x, y$	Target visible
<code>right_click</code>	$x, y$	Target visible
<code>drag</code>	$x_1, y_1, x_2, y_2$	Source visible
<code>scroll</code>	$x, y, \delta$	Target scrollable
<code>hover</code>	$x, y$	Target visible
<code>click_element</code>	id	Element exists
<code>drag_element</code>	id, $x, y$	Element draggable
<code>select_text</code>	$x_1, y_1, x_2, y_2$	Text region
<code>triple_click</code>	$x, y$	Text element
<code>mouse_down</code>	$x, y$ , btn	—
<code>mouse_up</code>	$x, y$ , btn	Mouse pressed

- `select_option(selector, value)`: Select dropdown options.
- `wait_for(selector, state)`: Wait for element state changes.
- `execute_js(code)`: Execute JavaScript in page context.
- `screenshot_element(selector)`: Capture specific element.
- `get_text(selector)`: Extract text content.
- `get_attribute(selector, name)`: Read element attributes.
- `new_tab()`, `close_tab()`, `switch_tab(idx)`: Tab management.
- `go_back()`: Navigate backward.

## 4.2 Action Categories

### 4.2.1 Mouse Actions (12 operations)

Mouse actions support both coordinate-based and element-based targeting. When using element-based targeting (`click_element`), the framework automatically computes the click point as the center of the element’s bounding box, adjusted for any scroll offset.

### 4.2.2 Keyboard Actions (10 operations)

Keyboard actions include `type_text`, `press_key`, `hotkey`, `key_down`, `key_up`, and text manipulation operations (`select_all`, `copy`, `paste`, `cut`, `undo`). The `type_text` action supports configurable inter-keystroke delay for applications that process keystrokes individually.

### 4.2.3 Browser Actions (12 operations)

When operating in browser mode, Operative provides higher-level browser-specific actions built on Playwright [13]:

- `navigate(url)`: Navigate to URL with wait-for-load semantics.
- `fill(selector, text)`: Fill form fields with automatic clearing.

Browser actions provide higher reliability than coordinate-based mouse actions for web tasks, as they are resilient to layout shifts and viewport changes.

### 4.2.4 System Actions (8 operations)

System actions interact with the operating system: `open_app`, `close_app`, `switch_app`, `take_screenshot`, `read_clipboard`, `write_clipboard`, `run_command`, and `file_dialog`. System actions are subject to safety boundary restrictions (Section 5).

### 4.2.5 Composite Actions (5 operations)

Composite actions encode common multi-step patterns: `search_and_click` (type in search field, wait for results, click match), `fill_form` (iterate over form fields and fill values), `login` (detect login form, enter credentials, submit), `download_file` (click download, wait for completion, verify file), and `scroll_to_find` (repeatedly scroll and search for target element).

## 4.3 Action Validation

Before execution, each action is validated against its pre-conditions:

```

1 def validate_click(state: ScreenState, 18
2     x: int, y: int) -> 19
3     bool: 20
4     # Check coordinates in viewport 21
5     if not state.viewport.contains(x, y) 22
6     : 23
7     return False 24
8     # Check no modal dialog blocking 25
9     if state.has_modal_at(x, y): 26
10    return False 27
11    # Check element is interactive 28
12    elem = state.element_at(x, y) 29
13    if elem and not elem.state.get(" 30
14    enabled"): 31
15    return False 32
16    return True 33
17 34

```

Listing 3: Pre-condition validation

Post-condition verification runs after action execution to confirm expected state changes. For example, `click` on a button verifies that the screen state changed (detected via screenshot differencing), `type_text` verifies that the text appears in the focused field, and `navigate` verifies that the URL changed to the expected target.

## 5 Safety Boundaries

Operative implements a defense-in-depth safety system to prevent agents from causing unintended harm.

### 5.1 Permission Model

Safety boundaries are configured through a declarative policy:

```

1 safety:
2   filesystem:
3     allowed_paths:
4       - ~/Downloads
5       - ~/Documents/agent-workspace
6     denied_paths:
7       - ~/.ssh
8       - ~/.aws
9       - /etc
10    max_file_size: 100MB
11
12   network:
13     allowed_domains:
14       - "*.google.com"
15       - "*.github.com"
16     denied_domains:
17       - "*.darkweb.*"

```

```

block_downloads_from: ["*"]

system:
  allow_app_launch: true
  allow_app_close: false
  allow_system_settings: false
  allow_shell_commands: false
  max_concurrent_windows: 5

interaction:
  require_confirmation:
    - purchase
    - delete
    - send_email
    - form_submission
  max_actions_per_minute: 60
  idle_timeout: 300

```

Listing 4: Safety boundary configuration

### 5.2 Sandboxing Layers

1. **Action-level filtering:** Each action is checked against the policy before execution. Denied actions return an error to the planner, which must select an alternative approach.
2. **Confirmation gates:** Actions matching `require_confirmation` patterns trigger a human-in-the-loop approval step. The agent pauses and presents the proposed action with context to the user.
3. **VM isolation:** For maximum security, Operative supports execution within a Docker container or lightweight VM with restricted host access. The screen is shared via VNC or headless framebuffer.
4. **Rollback capability:** Reversible actions maintain undo state. If a sequence of actions produces unintended results, the planner can issue rollback commands to restore previous state.

### 5.3 Rate Limiting and Circuit Breaking

To prevent runaway agents, Operative enforces:

- **Action rate limit:** Maximum 60 actions per minute (configurable), preventing rapid uncontrolled clicking.

- **Cost limit:** For actions that incur LLM costs, a per-task budget ceiling.
- **Repetition detection:** If the agent repeats the same action sequence more than 3 times, execution pauses for human review.
- **Error escalation:** After 5 consecutive failed actions, the agent enters supervised mode requiring human approval for each subsequent action.

## 6 Multi-Step Planning

### 6.1 Hierarchical Task Decomposition

Given a high-level goal (e.g., “Book a flight from SFO to JFK for March 15”), the planner decomposes it into a hierarchy of sub-tasks:

1. Open browser and navigate to flight booking site.
2. Enter departure city (SFO).
3. Enter destination city (JFK).
4. Select date (March 15).
5. Search for flights.
6. Select preferred flight from results.
7. Complete booking process.

Each sub-task is further decomposed into atomic actions:

```

1 @dataclass
2 class Task:
3     goal: str
4     subtasks: list["Task"]
5     actions: list[Action] # Leaf level
6     status: TaskStatus
7     checkpoint: ScreenState | None
8
9     def is_complete(self, state) -> bool:
10         """Check completion condition.
11         """
12
13     def rollback_to(self, checkpoint):
14         """Restore to checkpoint state.
15         """

```

Listing 5: Task decomposition structure

---

### Algorithm 2 Observation-Action Loop

---

**Require:** Goal  $g$ , safety policy  $\pi$ , model  $M$

```

1: plan  $\leftarrow M.decompose(g)$ 
2: history  $\leftarrow []$ 
3: while not plan.is_complete() do
4:    $s \leftarrow perceive()$  {Screen understanding}
5:   history.append( $s$ )
6:    $a \leftarrow M.select\_action(s, plan, history)$ 
7:   if not  $\pi.allows(a)$  then
8:      $a \leftarrow M.replan(s, a, \pi.reason)$ 
9:   end if
10:  if  $a.requires\_confirmation$  then
11:    await_user_approval( $a$ )
12:  end if
13:  result  $\leftarrow execute(a)$ 
14:  history.append( $a, result$ )
15:  if result.failed then
16:    plan.update( $M.handle\_error(s, a, result)$ )
17:  end if
18:  plan.update_progress( $s$ )
19: end while

```

---

### 6.2 Observation-Action Loop

The core execution loop follows an Observe-Think-Act cycle:

### 6.3 Error Recovery

When actions fail, the planner employs a graduated recovery strategy:

1. **Retry:** For transient failures (element not yet loaded), wait and retry.
2. **Alternative action:** Try a different approach to the same sub-goal (e.g., keyboard shortcut instead of menu click).
3. **Rollback and replan:** Restore to the last checkpoint and generate a new plan for the remaining sub-tasks.
4. **Escalate:** Report failure to the user with context and request guidance.

Checkpoints are created automatically at sub-task boundaries and can be created manually at any point. For browser tasks, checkpoints capture the page URL and relevant form state. For

Table 2: Platform support matrix

Feature	macOS	Linux	Win	Docker
Screen capture	✓	✓	✓	✓
Mouse/keyboard	✓	✓	✓	✓
Ally tree	✓	✓	✓	—
Browser (PW)	✓	✓	✓	✓
App launch	✓	✓	✓	—
VM sandbox	—	✓	—	✓

screen capture uses the X11 SHM extension or Wayland screencopy protocol. Input is generated via XTest or uinput. Docker support uses a headless X11 server (Xvfb) with VNC for remote observation.

## 7.2 Model Integration

Operative supports any model accessible via the Hanzo LLM Gateway [18]:

desktop tasks, checkpoints capture a screenshot and the accessibility tree.

## 6.4 Context Window Management

Computer use tasks generate large observation histories that can exceed LLM context windows. Operative manages context through:

- **Summarization:** Older observations are summarized by a secondary LLM call that extracts key state changes and decisions.
- **Relevance filtering:** Only elements relevant to the current sub-task are included in the prompt.
- **Screenshot compression:** Screenshots are resized and compressed before inclusion as vision inputs. Element annotations are overlaid on screenshots to reduce reliance on text descriptions.
- **Sliding window:** The most recent 5 observations are included in full; older observations are represented by their summaries.

- **Vision models:** Claude Sonnet/Opus, GPT-4o, Gemini Pro Vision—receive screenshots as image inputs.

- **Text models:** Any chat model—receive text-serialized `ScreenState` as input.

- **Local models:** Models running via Ollama or vLLM with vision capabilities.

The model interface is abstracted behind a `Planner` protocol:

```

1 class Planner(Protocol):
2     async def select_action(
3         self,
4         state: ScreenState,
5         task: Task,
6         history: list[Observation],
7     ) -> Action:
8         """Select next action given
9         current
10            state and task context."""
11
12     async def decompose(
13         self, goal: str
14     ) -> Task:
15         """Decompose goal into task
16            hierarchy."""
17
18     async def handle_error(
19         self,
20         state: ScreenState,
21         action: Action,
22         error: ActionError,
23     ) -> TaskUpdate:
24         """Decide recovery strategy for
25            failed action."""

```

Listing 6: Planner protocol

## 7 Implementation

Operative is implemented in Python (12,400 lines) with platform-specific native extensions for screen capture and accessibility tree access.

### 7.1 Platform Support

On macOS, screen capture uses the `ScreenCaptureKit` framework via `PyObjC`. Mouse and keyboard input uses Quartz Event Services. Accessibility tree access uses the AX API. On Linux,

### 7.3 Browser Automation Backend

For web-based tasks, Operative uses Playwright as the browser automation backend. This provides:

- Cross-browser support (Chromium, Firefox, WebKit).
- DOM access for precise element targeting.
- Network interception for monitoring requests.
- Built-in waiting mechanisms for dynamic content.
- Device emulation for mobile web testing.

The browser backend can operate in two modes: (1) **visual mode**, where the agent sees screenshots and uses coordinate-based actions, and (2) **structured mode**, where the agent receives DOM-based element descriptions and uses selector-based actions. Structured mode is more reliable but less generalizable to non-browser environments.

## 8 Evaluation

### 8.1 Benchmarks

We evaluate Operative on three established benchmarks:

1. **OSWorld** [14]: 369 real-world desktop tasks spanning 9 applications (Ubuntu environment).
2. **WebArena** [15]: 812 web-based tasks across 4 self-hosted web applications.
3. **VisualWebBench** [16]: 1,500 web understanding and interaction tasks with fine-grained annotations.

### 8.2 Task Completion Results

Operative with Claude achieves state-of-the-art results across all three benchmarks. The improvement over raw Claude computer use (22.0% → 38.1% on OSWorld) demonstrates the value of the structured perception pipeline and hierarchical planning.

Table 3: Task completion rates (%) across benchmarks

System	OSWorld	WebArena	VWB
GPT-4V (direct)	12.2	14.4	18.7
Claude 3.5 CU	22.0	35.1	41.2
SeeAct	31.6	—	38.4
CogAgent	—	29.8	35.9
WebVoyager	—	42.7	—
Operative (Claude)	<b>38.1</b>	<b>54.3</b>	<b>52.8</b>
Operative (GPT-4o)	34.7	48.9	49.1
Operative (Gemini)	31.2	44.2	46.7

Table 4: Ablation study on OSWorld (Claude backend)

Configuration	Completion (%)
Full system	38.1
– Accessibility tree fusion	31.4
– Hierarchical planning	29.7
– Error recovery	33.2
– Context summarization	35.6
– Pre-condition validation	36.8
Screenshot-only (no fusion)	24.3

### 8.3 Ablation Study

The hierarchical planning architecture contributes the largest improvement (+8.4 points), followed by accessibility tree fusion (+6.7 points). Error recovery accounts for +4.9 points, confirming the importance of graceful failure handling.

### 8.4 Perception Accuracy

The fused pipeline achieves 92.9% F1, significantly outperforming any individual source. The accessibility tree provides the highest precision but misses visual-only elements (icons without labels, decorative elements that serve as click targets). Visual detection catches these at the cost of some false positives, while OCR fills in text content missing from accessibility labels.

Table 5: Element detection accuracy by source

Source	Precision	Recall	F1
Visual only	89.1	72.3	79.8
OCR only	91.4	58.2	71.1
A11y only	96.2	81.7	88.3
Fused (all)	94.7	91.2	92.9

Table 6: Per-step latency breakdown (ms)

Component	p50	p95	p99
Screen capture	8	14	23
Visual detection	23	31	42
OCR	67	112	189
A11y tree query	12	28	54
Element fusion	3	7	12
LLM planning	1,840	4,200	8,100
Action execution	45	180	520
Post-verification	31	67	124
<b>Total per step</b>	<b>2,029</b>	<b>4,639</b>	<b>9,064</b>

## 8.5 Latency Analysis

LLM planning dominates the per-step latency at 91% of total time. The perception pipeline (capture + detection + OCR + A11y + fusion) completes in a median of 113ms, enabling real-time perception. With local models (e.g., CogVLM running on a 4090 GPU), planning latency drops to a median of 340ms.

## 8.6 Safety Boundary Effectiveness

We evaluate safety boundaries by running Operative on 100 adversarial tasks designed to trick agents into harmful actions (deleting files, exfiltrating data, making purchases):

- **Blocked by policy:** 94 of 100 harmful actions were blocked by the safety boundary system.
- **Caught by confirmation:** 4 additional harmful actions were caught by the confirmation gate.
- **Missed:** 2 actions circumvented boundaries through indirect means (e.g., using a text ed-

itor to write a shell script, then executing it through a different application). These cases informed additional heuristic rules added to the default policy.

# 9 Discussion

## 9.1 Limitations

**Dynamic content.** Rapidly changing content (animations, video, real-time dashboards) degrades perception accuracy, as the screen state may change between observation and action execution.

**Complex reasoning.** Tasks requiring multi-step logical reasoning (spreadsheet formula construction, code debugging) remain challenging, as the agent must maintain extended mental models beyond what the screen state provides.

**Speed.** At a median of 2 seconds per action step, Operative is significantly slower than human interaction, limiting applicability to time-sensitive tasks.

## 9.2 Ethical Considerations

Computer use agents raise important ethical concerns:

- **Unauthorized access:** Agents must not be used to bypass authentication or access controls.
- **Privacy:** Screen observations may capture sensitive information. Operative supports configurable redaction of detected PII in logs and screenshots.
- **Impersonation:** Agents acting on behalf of users in interactive systems must clearly identify as automated.

# 10 Conclusion

Operative provides a principled, model-agnostic framework for computer use that advances the

state of the art through structured perception, formally specified actions, and hierarchical planning with safety guarantees. By decoupling the perception, planning, and execution layers, Operative enables rapid experimentation with different models and strategies while maintaining consistent safety properties. The 71.3% improvement over baseline computer use approaches validates the importance of systematic framework design in this emerging capability domain.

## References

- [1] T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang. World of bits: An open-domain platform for web-based agents. In *ICML*, 2017.
- [2] T. Schick, J. Dwivedi-Yu, R. Dessì, et al. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.
- [3] Anthropic. Introducing computer use. <https://www.anthropic.com/news/3-5-models-and-computer-use>, 2024.
- [4] OpenAI. Operator: A new AI agent that can use a web browser. <https://openai.com/index/introducing-operator/>, 2025.
- [5] Google DeepMind. Project Mariner: Building AI agents for the web. Technical report, Google, 2024.
- [6] T. Yeh, T.-H. Chang, and R. Miller. Sikuli: Using GUI screenshots for search and automation. In *UIST*, 2009.
- [7] W. Hong, W. Wang, Q. Lv, et al. CogAgent: A visual language model for GUI agents. In *CVPR*, 2024.
- [8] K. Cheng, Q. Sun, Y. Chu, et al. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. *arXiv preprint arXiv:2401.10935*, 2024.
- [9] H. He, W. Yao, K. Ma, et al. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.
- [10] L. Zheng, Z. Wang, K. Ma, et al. AgentStudio: A toolkit for building general virtual agents. *arXiv preprint arXiv:2403.17918*, 2024.
- [11] A. Dosovitskiy, L. Beyer, A. Kolesnikov, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.
- [12] Y. Baek, B. Lee, D. Han, S. Yun, and H. Lee. Character region awareness for text detection. In *CVPR*, 2019.
- [13] Microsoft. Playwright: Reliable end-to-end testing for modern web apps. <https://playwright.dev>, 2024.
- [14] T. Xie, D. Zhang, J. Chen, et al. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.
- [15] S. Zhou, F. Xu, H. Zhu, et al. WebArena: A realistic web environment for building autonomous agents. In *ICLR*, 2024.
- [16] J. Liu, H. Li, X. Liu, et al. VisualWebBench: How far have multimodal LLMs evolved in web page understanding and grounding? *arXiv preprint arXiv:2404.05955*, 2024.
- [17] J. Wang, H. Xu, J. Ye, et al. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.
- [18] Hanzo AI Research. Hanzo LLM Gateway: Unified proxy for 100+ AI providers. Technical report, Hanzo AI, 2026.
- [19] J. Koh, R. Lo, L. Jang, et al. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *ACL*, 2024.
- [20] X. Deng, Y. Gu, B. Zheng, et al. Mind2Web: Towards a generalist agent for the web. In *NeurIPS*, 2023.

- [21] G. Kim, P. Baldi, and S. McAleer. Language agents with reinforcement learning for strategic play in the Werewolf game. *arXiv preprint arXiv:2310.18602*, 2024.
- [22] B. Zheng, B. Gou, J. Kil, et al. GPT-4V(ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024.
- [23] C. Rawles, S. Li, D. Wiegand, et al. AndroidWorld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573*, 2024.
- [24] X. Lu, S. Gao, Z. Chen, et al. WebLINX: Real-world website navigation with multi-turn dialogue. In *ICML*, 2024.
- [25] S. Putta, E. Mills, N. Garg, et al. Agent Q: Advanced reasoning and learning for autonomous AI agents. *arXiv preprint arXiv:2408.07199*, 2024.