AutoRPA: Efficient GUI Automation through LLM-Driven Code Synthesis from Interactions

ICML 2026

Minghao Chen, Xinyi Hu, Zhou Yu, Yufei Yi

Hangzhou Dianzi University

Background: Real-World GUI Automation Scenarios

Finance
Expense report entry, invoice verification

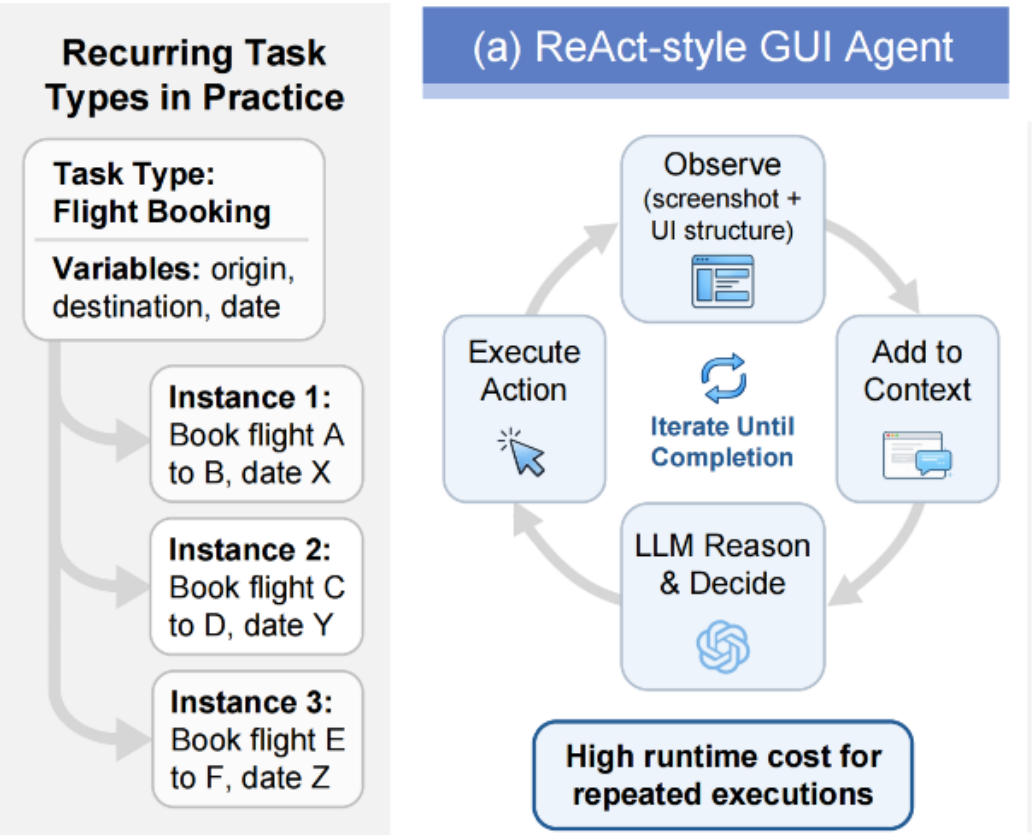
Office
Email sorting, report generation

E-commerce
Order processing, product listing

Administration
Attendance statistics, travel booking

Limitation of ReAct

Requires step-by-step LLM inference for every task instance, leading to prohibitive token costs for repetitive tasks



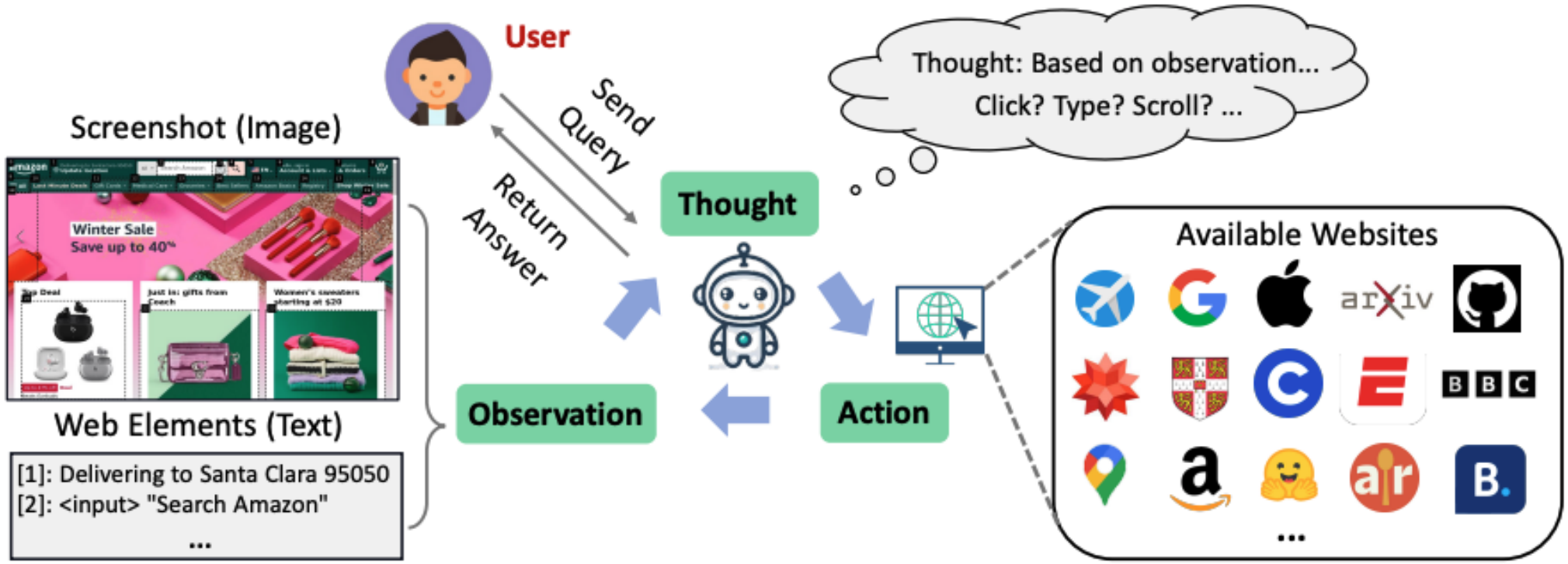
ReAct: Synergizing reasoning and acting in language models. *ICLR* 2023

Multimodal GUI Agents

Multimodal LLMs enabled agents to interact with real-world GUIs via screenshots

Core: These works demonstrated that LLM agents could reliably handle practical GUI tasks like web browsing, email management, and mobile app operation

Limitation: reason and execute one step at a time



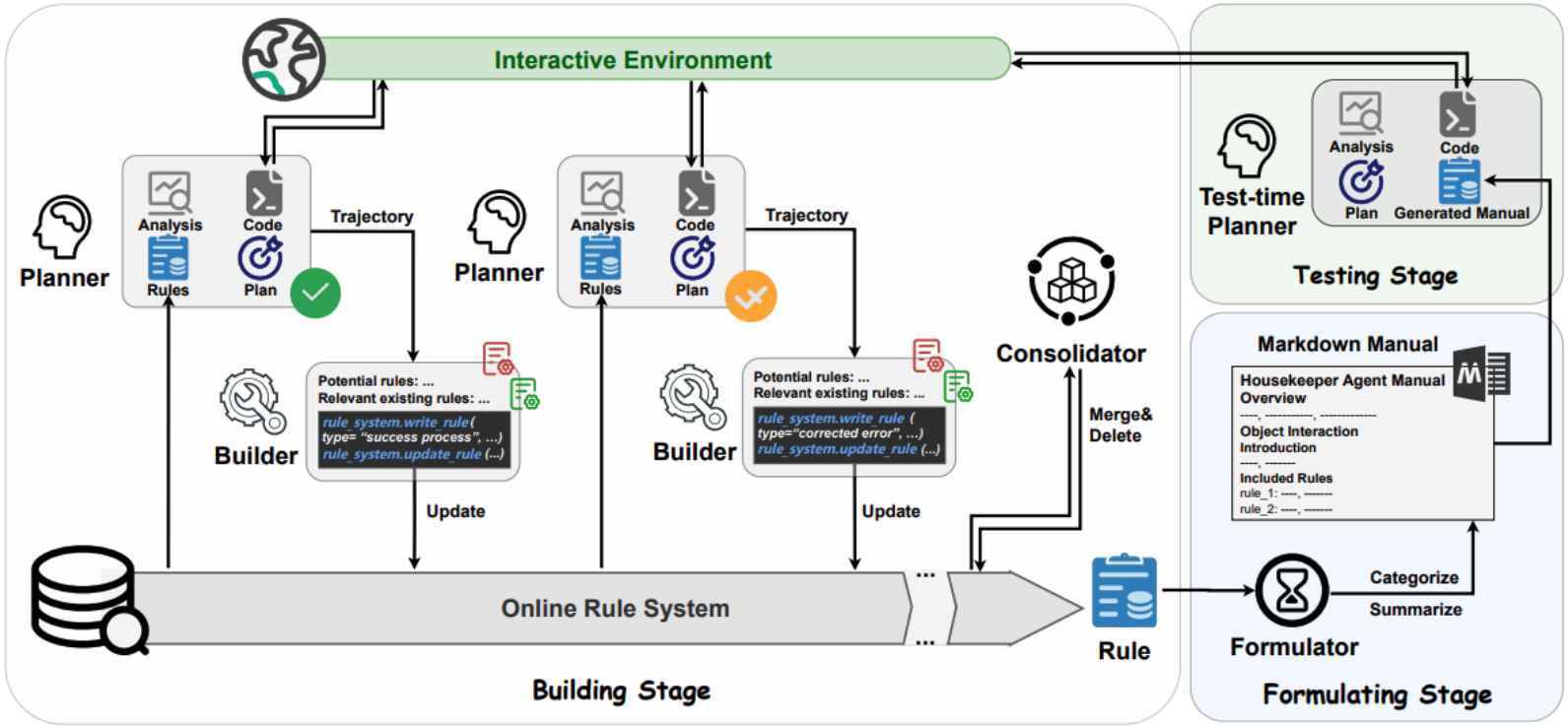
Experience-Driven Self-Optimization

Reflexion (2023): Agents reflect on failures to improve future attempts

ExpeL (2024): Extract cross-task rules from offline trajectories

AutoManual (NeurIPS 2024): Our prior work that generates human-readable instruction manuals from interactions

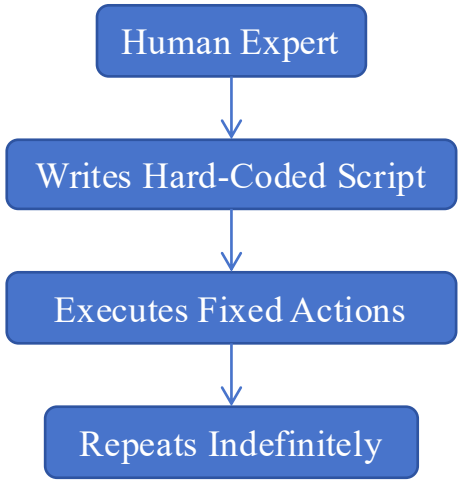
Core Limitation: The manual is used as a prompt for LLM at test time, not as executable code



AutoManual: Constructing Instruction Manuals by LLM Agents via Interactive Environmental Learning, NeurIPS 2024

What is Traditional Robotic Process Automation (RPA)?

RPA workflow diagram



Example of brittle RPA code

```
# Traditional RPA Login Script (1920x1080 ONLY)
# Breaks on ANY UI change or resolution difference

import pyautogui, time

time.sleep(2)
pyautogui.click(x=450, y=280)  # Click username field
pyautogui.typewrite("admin")
pyautogui.click(x=450, y=320)  # Click password field
pyautogui.typewrite("password123")
pyautogui.click(x=450, y=380)  # Click Login button
time.sleep(3)
pyautogui.click(x=120, y=180)  # Click Dashboard menu
```

Three Fatal Flaws of Traditional RPA

- ❌ High Manual Development Cost
- ❌ Extreme Brittleness
- ❌ High Maintenance Overhead

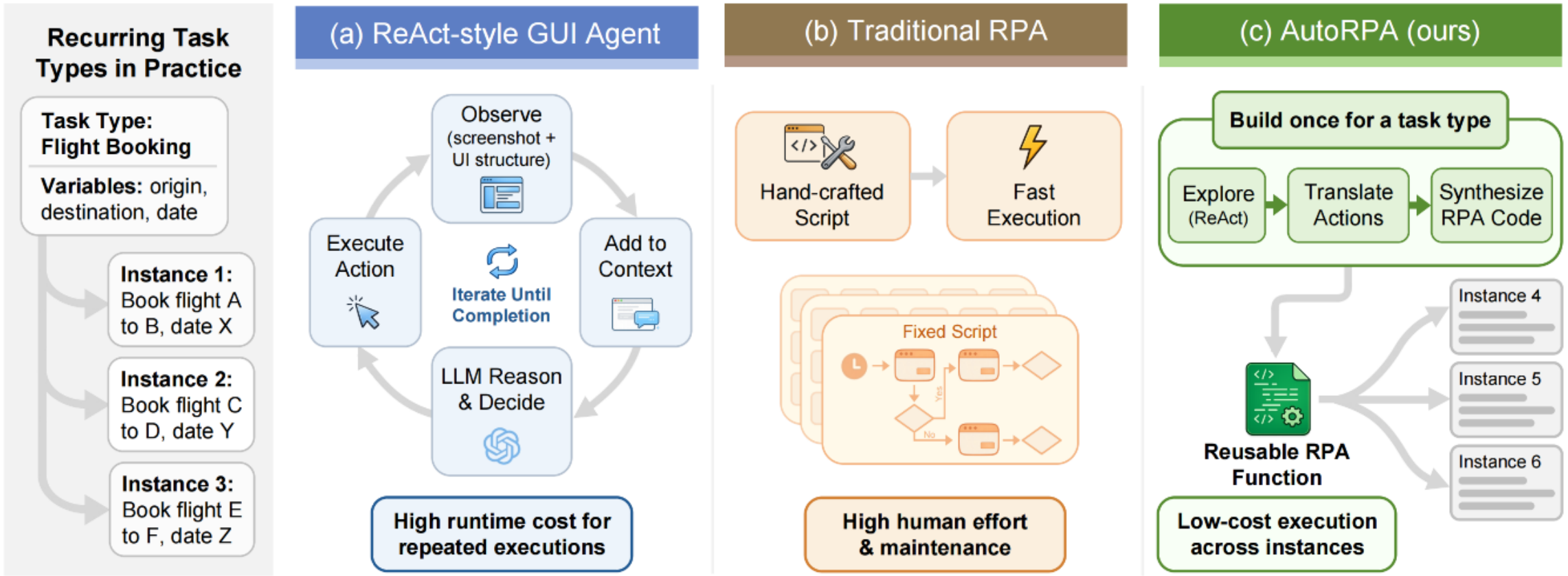
The Unaddressed Gap: Efficiency for Repetitive Tasks

All existing methods optimize for novel task performance, not repetitive task efficiency.

AutoRPA filled this gap!

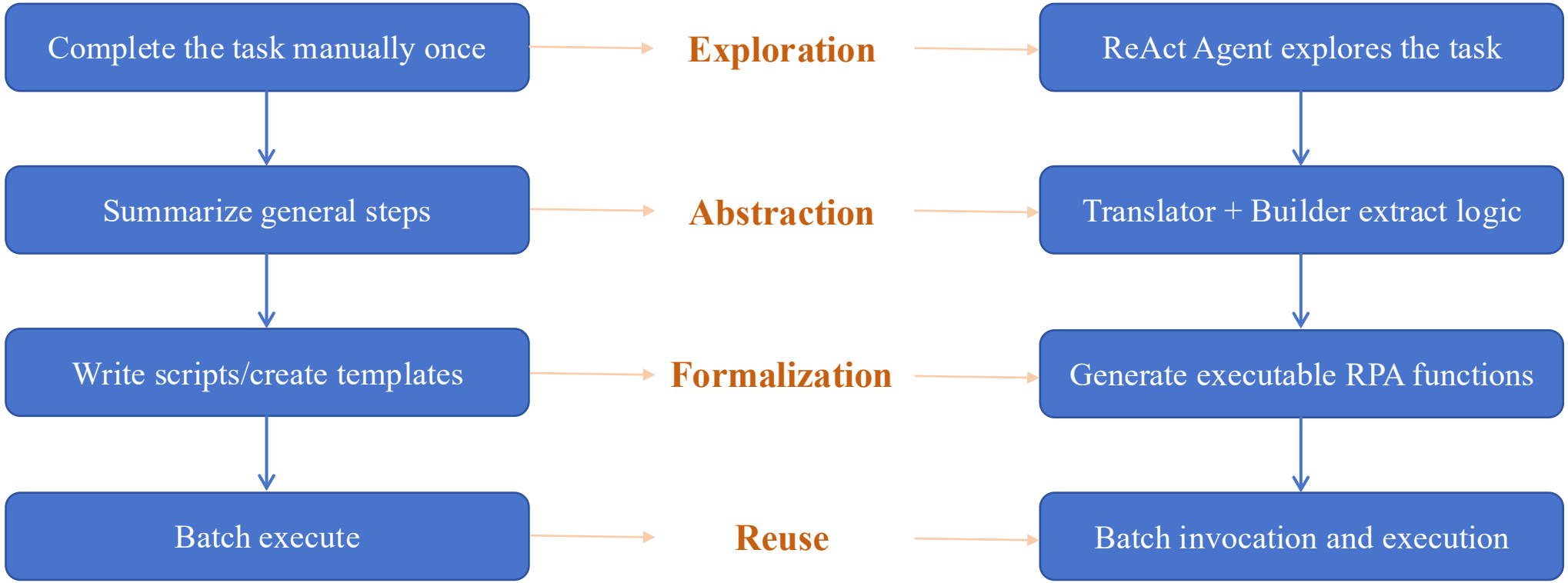
Method	Human effort	Reuse of Experience	Test-Time LLM Dependency	Token Cost for Repetitive Tasks
ReAct	✅ Medium	❌ None	❌ Full	❌ Very High
SeeAct/WebVoyager	✅ Medium	❌ None	❌ Full	❌ Very High
AutoManual	✅ Low	✅ Rules/Manuals	❌ Full	❌ High
Tranditional RPA	❌ High	❌ None	✅ None	✅ Extremely Low
AutoRPA (Ours)	✅ Low	✅ Executable Code	✅ Minimal	✅ Extremely Low

Pain Point: The Dilemma of Existing GUI Automation Paradigms

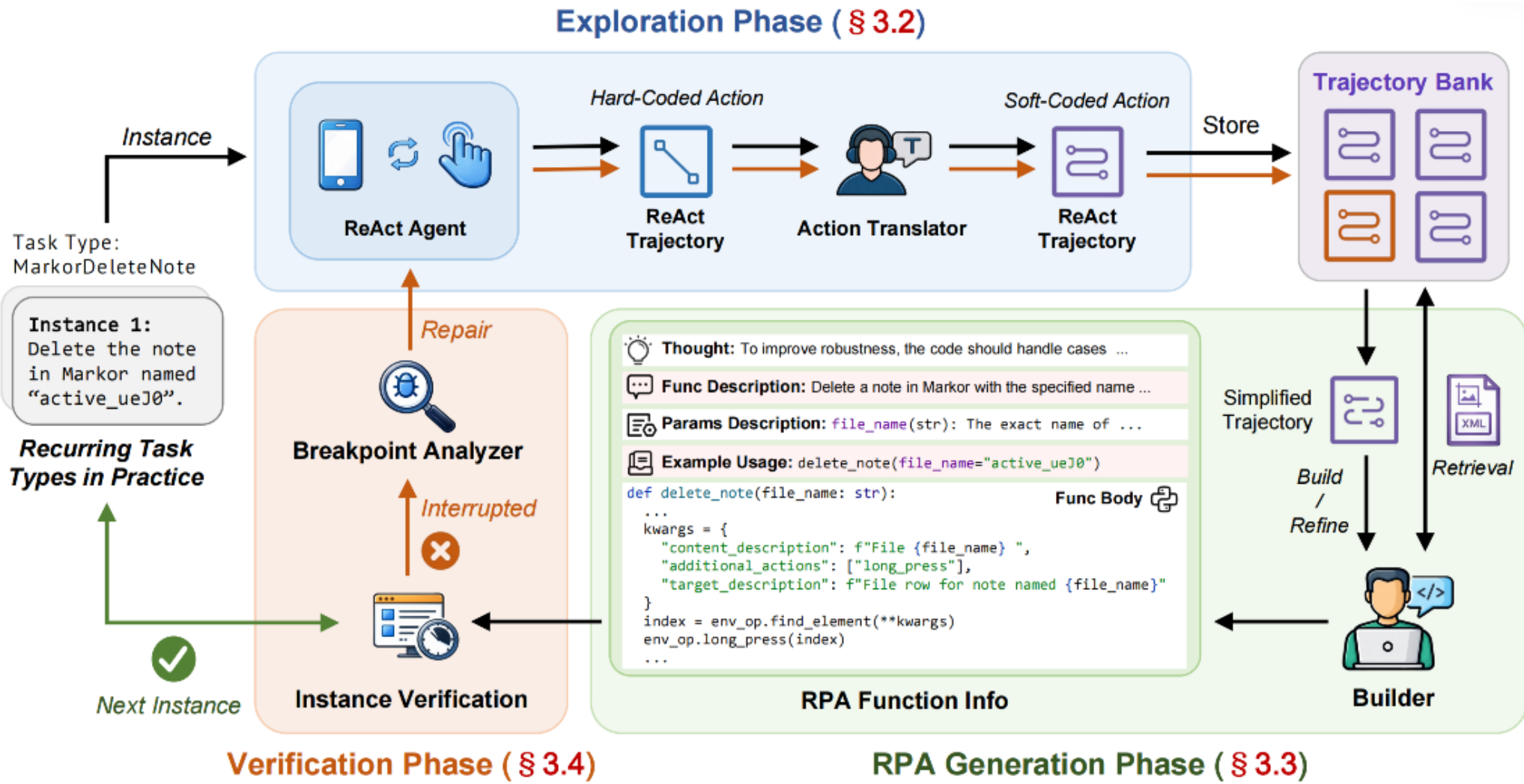


Core Idea: Learn from Human Automation Thinking

Human approach to repetitive tasks: **Explore once, summarize, then execute repeatedly**



AutoRPA Overall Architecture



Core Method 1: Exploration Phase -- Hard-Coded to Soft-Coded Translation

Solve the poor robustness of ReAct actions

- **ReAct hard-coded actions**

```
# Depends on element index, breaks on layout changes
agent.click(index=2)
agent.input_text(index=5, "123456")
```

- **Soft-coded actions generated by Translator**

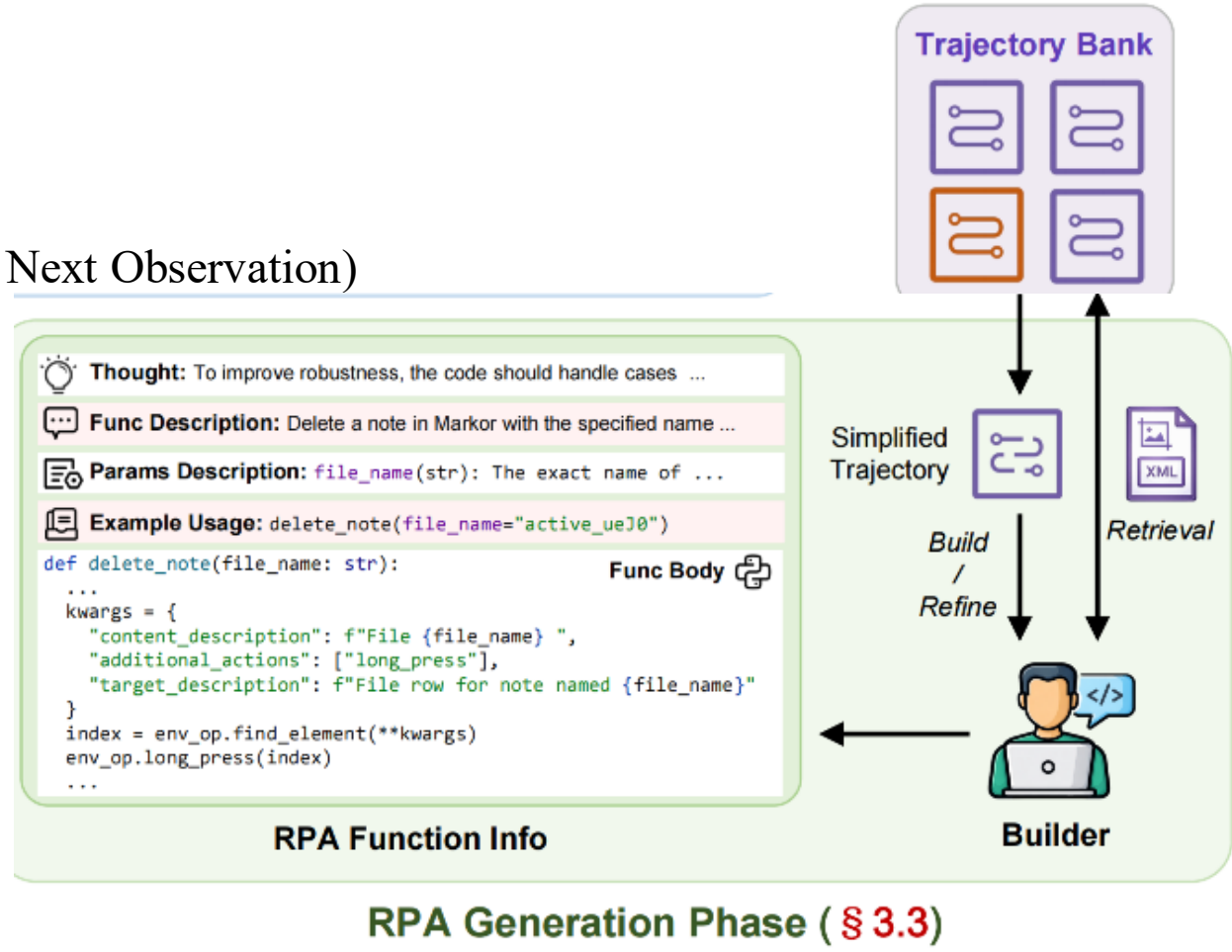
```
# Semantic attribute-based localization, highly robust
kwargs = {"text": "Password", "target_description": "Central password input field"}
pwd_index = env_op.find_element(**kwargs)
assert pwd_index != -1, "Password field not found"
env_op.input_text(pwd_index, "123456")
```

Core Method 2: Generation Phase -- Tree-Structured RAG for Code Synthesis

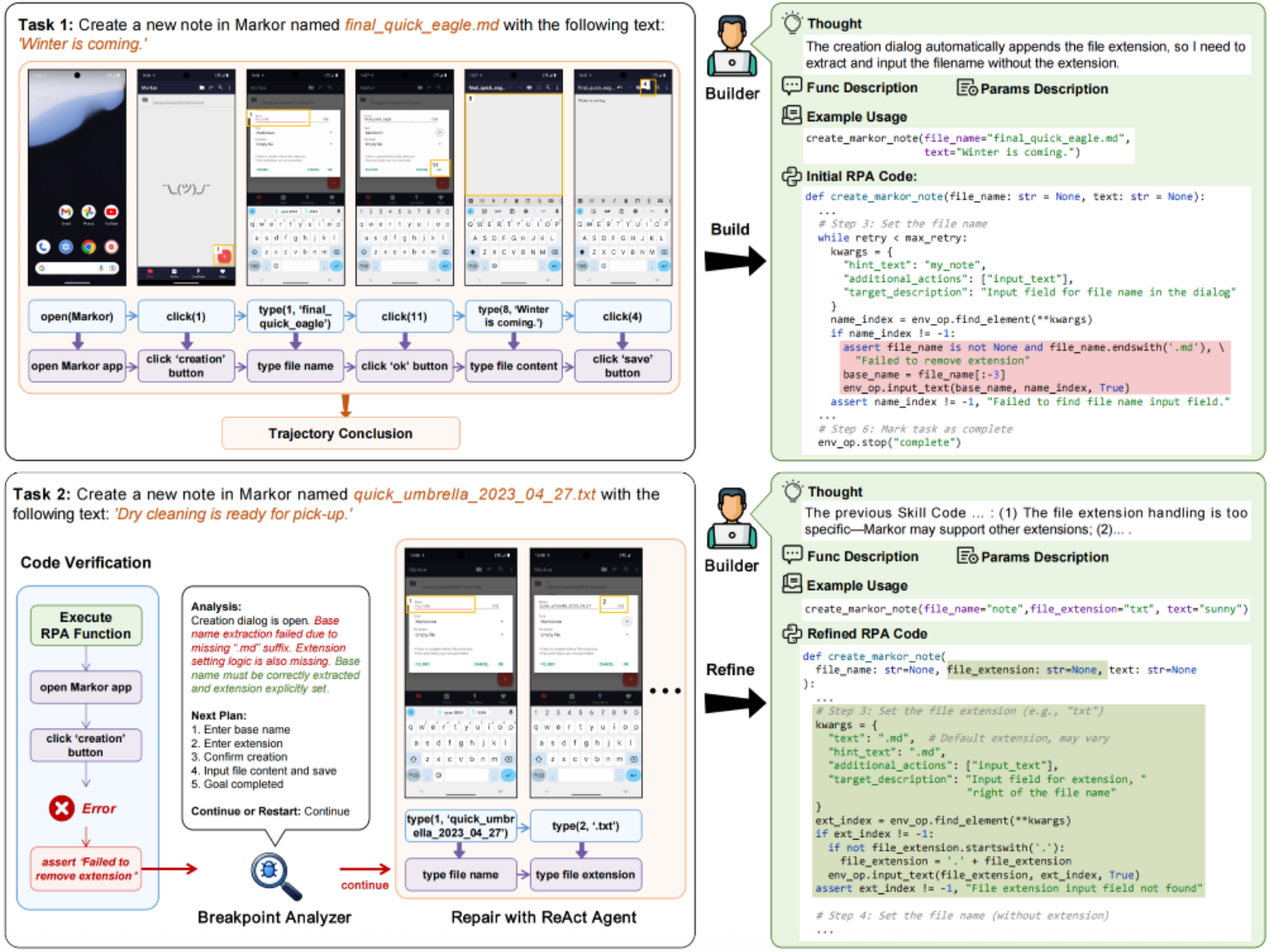
Top layer:
Trajectory conclusions (Overall task summary)

Middle layer:
Simplified trajectories (Actions + Results)

Bottom layer:
Interaction blocks (Observation + Action + Result + Next Observation)



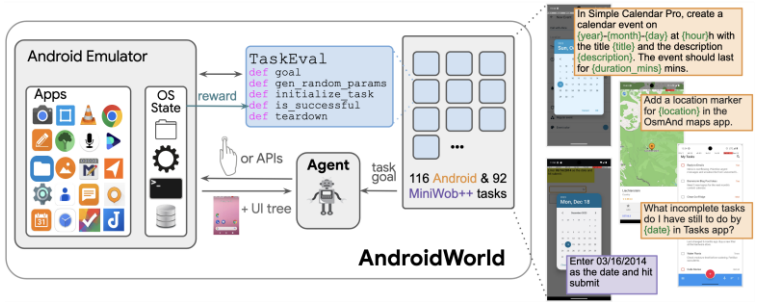
Core Method 3: Verification Phase -- Hybrid Repair Strategy



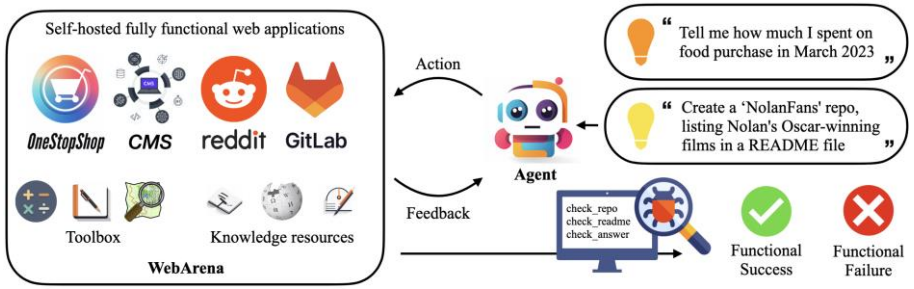
Experimental Setup

Benchmarks

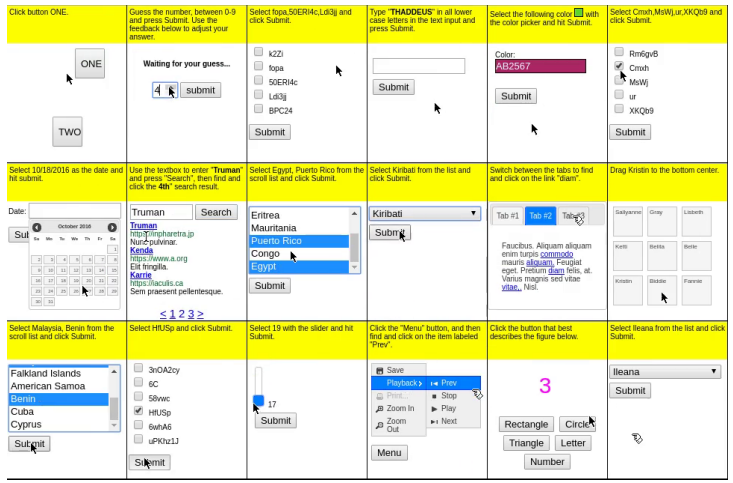
AndroidWorld
Real Android applications, 116 task types (Most realistic)



WebArena (Reddit Domain)
Real web environment, 19 task types (Most complex)



MiniWoB++
Simulated web environment, 53 task types (Most widely used benchmark)



Compared methods

ReAct paradigm
ReAct†, SeeAct, M3A, SteP

Skill learning
AdaPlanner, AutoManual, AutoGuide

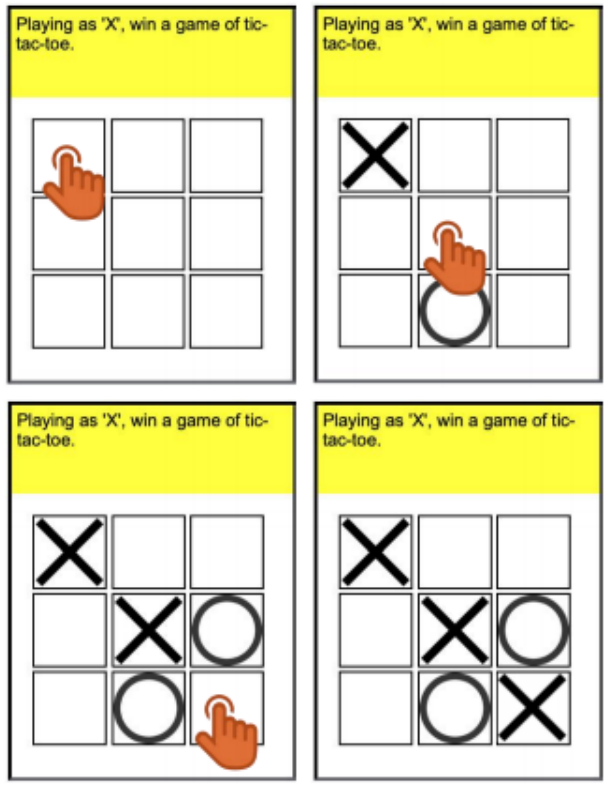
Main Result 1: AndroidWorld (SOTA + 81% Token Reduction)

Method	Model	Time (min) ↓	Tokens (*k*) ↓	Success (%) ↑
SeeAct (Zheng et al., 2024)	GPT-4o	6.38	68.4	15.3
M3A (Rawles et al., 2025)	GPT-4o	2.83	120.2	34.5
ReAct†	GPT-4o	4.97	79.9	35.2
AutoRPA (code only)	GPT-4o	**1.08**	**2.6**	34.3
AutoRPA	GPT-4o	1.76	14.7	**37.0**
SeeAct (Zheng et al., 2024)	GPT-4.1	5.14	58.8	25.4
M3A (Rawles et al., 2025)	GPT-4.1	2.23	103.4	48.3
ReAct†	GPT-4.1	3.91	68.7	50.0
AutoRPA (code only)	GPT-4.1	**1.42**	**2.7**	47.2
AutoRPA	GPT-4.1	1.81	12.8	**51.7**
SeeAct (Zheng et al., 2024)	GPT-5	12.43	134.5	40.3
M3A (Rawles et al., 2025)	GPT-5	5.56	164.6	57.0
ReAct†	GPT-5	8.57	142.5	74.1
AutoRPA (code only)	GPT-5	**2.72**	**6.2**	70.7
AutoRPA	GPT-5	4.35	30.6	**75.9**

Main Result 2: MiniWoB++ (94% Token Reduction + Par with AutoManual)

Methods	Hard (9 types)		All (53 types)	
	Tokens (k) ↓	Success (%) ↑	Tokens (k) ↓	Success (%) ↑
RCI (Kim et al., 2023)	19.7	57.8	10.2	87.2
AdaPlanner (Sun et al., 2023)	15.1	74.1	6.1	90.3
AdaPlanner (one demo)	13.1	24.4	4.5	74.3
AutoManual (Chen et al., 2024)	23.2	**91.1**	4.6	<u>95.2</u>
ReAct†	16.2	84.4	9.2	92.8
AutoRPA (code only)	**1.0**	80.0	**0.9**	92.5
AutoRPA	<u>1.4</u>	**91.1**	<u>1.4</u>	**95.4**

Case Study: Tic-Tac-Toe Code Comparison



AdaPlanner

```
def solution(agent, start_from=1):
    # General plan: I need to play as 'X' and win the tic-tac-toe game. The board is empty except for an 'O' at position 7 (bottom row, middle). I will play to create a winning line for 'X'. I will play top-left (0), center (4), and bottom-right (8) for a diagonal win.

    # [Step 1] Click the top-left cell (id="ttt-0") to place 'X'
    html = agent.click_xpath("//span[@id='ttt-0']")
    assert 'mark-x' in html or 'ttt-0' in html, "Step 1: X should be placed in top-left cell"

    # [Step 2] Click the center cell (id="ttt-4") to place 'X'
    html = agent.click_xpath("//span[@id='ttt-4']")
    assert 'mark-x' in html or 'ttt-4' in html, "Step 2: X should be placed in center cell"

    # [Step 3] Click the bottom-right cell (id="ttt-8") to place 'X' and win
    html = agent.click_xpath("//span[@id='ttt-8']")
    assert 'mark-x' in html or 'ttt-8' in html, "Step 3: X should be placed in bottom-right cell"
```

Case Study: Tic-Tac-Toe Code Comparison

AutoManual

```
# Helper method to click on a cell in the tic-tac-toe grid
def click_cell(agent, cell_id):
    cell_xpath = f"//*[@id='{cell_id}']"
    return agent.click_xpath(cell_xpath)

# [Step 1] Click the center cell (ttt-4) if it's empty.
html_string = click_cell(agent, 'ttt-4')
# *Pause here* to observe the opponent's move and the updated HTML.

# [Step 2] Click one of the corners.
# Choosing the bottom-left corner (ttt-6) as it's part of multiple winning combinations.
html_string = click_cell(agent, 'ttt-6')
# *Pause here* to observe the opponent's move and the updated HTML.

# [Step 3] Click the top-left corner (ttt-0) to continue the winning strategy.
html_string = click_cell(agent, 'ttt-0')
# *Pause here* to observe the opponent's move and the updated HTML.

# [Step 4] Click the left-middle cell (ttt-3) to create a two-way winning opportunity.
html_string = click_cell(agent, 'ttt-3')
# *Pause here* to observe the opponent's move and the updated HTML.
```

AutoRPA

```
def play_tic_tac_toe(my_symbol: str = "X", opponent_symbol: str = "O"):
    WIN  = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
    PREF = [4,0,2,6,8,1,3,5,7]          # center → corners → edges

    def read_board():
        html  = env_op.get_html_body()
        bd = ['']*9
        for i,cls in re.findall(r'id="ttt-(\d)".+?class="([^"]*)"', html):
            bd[int(i)] = 'X' if 'mark-x' in cls else 'O' if 'mark-o' in cls else ''
        return bd

    def fin_move(bd,sym):
        for a,b,c in WIN:
            line=[bd[a],bd[b],bd[c]]
            if line.count(sym)==2 and '' in line:
                return (a,b,c)[line.index('')]
        return None

    def choose_move(bd):
        if (m:=fin_move(bd, my_symbol)) is not None: return m, "Winning move"
        if (m:=fin_move(bd, opponent_symbol)) is not None: return m, "Blocking move"
        m = next(i for i in PREF if bd[i]=='')
        if m==4: desc="Center move"
        elif m in (0,2,6,8): desc="Corner move"
        else: desc="Edge move"
        return m, desc

    # ------- Main loop (≤5 moves) ----------
    for _ in range(5):
        board = read_board()
        if any(all(board[i]==p for i in w) for p in 'XO' for w in WIN) or '' not in board:
            break
        move, desc = choose_move(board)
        kwargs = {"self_dom": {'tag':'span','id':f'ttt-{move}'}, "target_description": desc}
        xpath = env_op.find_element(**kwargs)
        env_op.click_xpath(xpath)
        time.sleep(0.2)
```

Thanks for watching!

Funding Support: National Natural Science Foundation of China, Zhejiang Provincial Natural Science Foundation

Contact: zhouyu@hdu.edu.cn