Self-Evolving MCP-GUI Agents via Automated Environment Generation and Experience Learning

Tiantian He[1], Yihang Chen[1], Keyue Jiang[1], Ka Yiu Lee[2], Kaiwen Zhou[3], Kun Shao[4†], and Shuai Wang[5†‡]

[1] University College London, United Kingdom
[2] Independent Researcher, United Kingdom
[3] The Chinese University of Hong Kong, Hong Kong SAR, China
[4] Independent Researcher, China
[5] Yale University, United States

Abstract. Computer-use agents that combine GUI interaction with structured API calls via the Model Context Protocol (MCP) show promise for automating software tasks. However, existing approaches lack a principled understanding of how agents should balance these two modalities and how to enable iterative self-improvement across diverse applications. We formulate MCP-GUI interplay as a **unified hybrid policy learning problem** where the agent learns *when* each modality provides complementary advantages, and show that distillation and experience augmentation target fundamentally different failure modes—requiring application-aware mechanism selection. Built on this formulation, we propose a **self-evolving framework** with a fully automatic pipeline that orchestrates *automatic environment generation and validation*, trajectory collection, gap-driven task synthesis, and quality-filtered training—all without manual intervention. A key innovation is our **experience bank**, which accumulates LLM-learned rules from trajectory comparison, enabling inference-time improvement without fine-tuning. Systematic **cross-application analysis** across three desktop applications reveals that the optimal strategy depends on MCP–GUI composition and toolchain complexity: distillation achieves 77.8% pass rate on MCP-dominant tasks (+17.8pp), while the experience bank excels on GUI-intensive tasks (+10.0pp). Code available at: `https://github.com/Tiantian-H/EE-MCP`

Keywords: Computer use agents · Model Context Protocol · Self-evolution

1 Introduction

Computer-use agents (CUAs) powered by large language models have emerged as a practical paradigm for automating complex software tasks. These agents interact

† Corresponding authors: Shuai Wang (`shuai.wang.sw2572@yale.edu`), Kun Shao (`shaokun1991@gmail.com`).
‡ Project leader.

with applications through multiple modalities—visual perception of graphical user interfaces, keyboard/mouse actions and structured API calls via tool protocols. Such hybrid interaction patterns are becoming increasingly common across diverse domains, from web automation to desktop productivity tools. The challenge lies not only in mastering individual modalities, but in learning *when* to leverage each for complementary advantages.

The recent introduction of Model Context Protocol (MCP) [1] exemplifies this trend, enabling agents to combine structured API calls with traditional GUI manipulation. This raises a fundamental question:

How should agents learn to balance MCP tool calls and GUI actions, and what mechanisms enable effective self-improvement across diverse applications?

Recent benchmarks such as MCPWorld [16] and OSWorld-MCP [8] highlighted the importance of this question. MCPWorld demonstrates that hybrid agents combining MCP and GUI outperform both GUI-only and MCP-only approaches. OSWorld-MCP further shows that MCP tools can significantly improve agent accuracy, while also revealing that tool invocation rates remain low even for state-of-the-art models. These findings underscore the need for methods that help agents learn *when* and *how* to use each modality effectively.

Existing training methods rely on either one-shot supervised fine-tuning (SFT) from expert demonstrations or online reinforcement learning (RL) with environment rewards. Both share key limitations: they treat all samples or rewards equally regardless of *which* weaknesses most need correction, do not diagnose systematic failure patterns to generate targeted training data, and do not reveal how evolution mechanisms interact with application-specific MCP–GUI task compositions.

We propose a **self-evolving policy learning** framework (Figure 1) that, rather than treating MCP tools and GUI actions as independent modalities, formulates their interplay as a **unified hybrid policy learning problem** where the agent learns *when* each modality provides complementary advantages. The framework runs as a fully automatic closed loop: multi-dimensional profiling identifies systematic weaknesses that drive targeted task and environment generation, while an experience bank accumulates LLM-learned rules for inference-time improvement—all without manual intervention. A central finding is that the effectiveness of these mechanisms depends on each application's **MCP–GUI task composition**: Chrome benefits most from distillation (+17.8pp), while GUI-intensive VS Code benefits primarily from the experience bank (+10pp), requiring *application-aware* mechanism selection.

Contributions. We make the following contributions:

- A **hybrid MCP-GUI policy learning formulation** that models structured API calls and visual GUI actions as a unified sequential decision problem where the agent learns *when* to use each modality; we show distillation and experience augmentation are complementary rather than interchangeable, requiring application-aware mechanism selection driven by both MCP–GUI composition and tool-chain complexity.

- A **self-evolving framework** with a fully automatic pipeline orchestrating expert trajectory collection, environment generation and validation, experience-bank construction, LLM-judge evaluation, gap analysis, and adaptive task generation—without manual intervention.
- An **experience bank** that accumulates concise, actionable rules from LLM-based trajectory comparison for inference-time improvement without fine-tuning, organized by skill category with capacity limits and application-type filtering.
- Systematic **cross-application analysis** across Chrome, VS Code, and LibreOffice Calc showing the optimal evolution strategy depends on each application's MCP–GUI task composition, with practical guidance for deployment.

2 Related Work

Computer Use Agents and Benchmarks. GUI-agent progress has been driven by benchmarks: OSWorld [13] for real desktop applications, WebArena [18] and VisualWebArena [9] for web tasks, and Mind2Web [5] for large-scale web interaction data. Most relevant, MCPWorld [16] and OSWorld-MCP [8] target hybrid MCP-GUI agents, showing that hybrid approaches outperform single-modality ones while tool-invocation rates stay low even for strong models. Whereas these benchmarks characterize *what* agents can do, we focus on *how* agents learn to balance modalities through self-evolution.

Learning from Demonstrations. Agent training from demonstrations has been explored extensively. AgentTrek [14] uses web-based trajectory collection. DigiRL [2] combines offline and online learning for device control. Agent Workflow Memory [12] extracts reusable procedures from demonstrations. Our approach differs by using Claude as an expert demonstrator and incorporating iterative refinement based on multi-dimensional performance feedback.

Self-Improvement and Iterative Learning. Self-improvement methods for LLMs have shown promise across domains: STaR [17] bootstraps reasoning from self-generated rationales, Self-Instruct [11] generates training data from model outputs, and ReST [6] applies reinforced self-training on the model's own high-reward generations. Our framework adapts these ideas to computer-use agents via structured performance profiles that drive targeted data generation; our self-improvement variant (Section 4.5) uses rejection sampling—training only on the model's own successful outputs to avoid distribution shift.

3 Problem Formulation

3.1 Computer Use as Sequential Decision Making

We formulate computer use as a Markov Decision Process (MDP) where an agent interacts with applications through both MCP tools and GUI actions. The central

challenge is learning a **hybrid policy** that selects the optimal modality at each step—a decision that depends on the application, task type, and current state.

State Space. Each state $s \in \mathcal{S}$ comprises the current screenshot $s_{\text{visual}} \in \mathbb{R}^{H \times W \times 3}$, available MCP tool descriptions s_{mcp}, and action history s_{history}.

Action Space. The agent selects from two modalities at each step:

MCP Actions $\mathcal{A}_{\text{MCP}}$: Structured API calls following the Model Context Protocol format:

$$a_{\text{mcp}} = \langle \texttt{tool\_name}, \texttt{arguments} \rangle \tag{1}$$

GUI Actions $\mathcal{A}_{\text{GUI}}$: Visual interface interactions including click, type, scroll, and keyboard shortcuts:

$$a_{\text{gui}} = \langle \texttt{action\_type}, \texttt{coordinates}, \texttt{parameters} \rangle \tag{2}$$

Task Specification and Evaluation. A task $\tau = \langle g, s_0, d, \mathbf{c}, \phi_{\text{judge}} \rangle$ consists of a natural language goal g, initial state s_0, difficulty level $d \in \{\text{easy}, \text{medium}, \text{hard}\}$, required skill categories $\mathbf{c} \subseteq \mathcal{C}$, and an LLM-judge evaluation function $\phi_{\text{judge}} : \xi \to [0, 1]$.

3.2 Skill Categories

We define six **application-agnostic** skill categories $\mathcal{C}$ that characterize different aspects of computer use: **Data Retrieval** (c_{retrieve})—reading files, querying databases; **Data Manipulation** ($c_{\text{manipulate}}$)—writing, editing, creating data; **Search & Query** (c_{search})—finding patterns, filtering results; **Execution & Automation** (c_{exec})—running scripts, triggering actions; **Navigation & Browsing** (c_{nav})—moving between views and UI elements; **Configuration & Settings** (c_{config})—modifying preferences and parameters. These categories apply across all applications in our benchmark (Table 2); detailed definitions with cross-application examples are provided in the supplementary material.

3.3 Two-Stage Policy Iteration Framework

We formulate agent learning as a two-stage iterative framework rather than one-shot imitation. Let π_k denote the agent policy at iteration k. Each iteration consists of two stages:

Stage 1: Policy Evaluation via Multi-Dimensional Profiling. The current policy π_{k-1} is evaluated on a task set $\mathcal{G}$ using an LLM judge ϕ_{judge}, producing a performance profile $\mathcal{P}_k$ that tracks capabilities across modality balance, difficulty levels, and skill categories (Section 4.3). This profile plays a role analogous to a value function, identifying where the policy is weak.

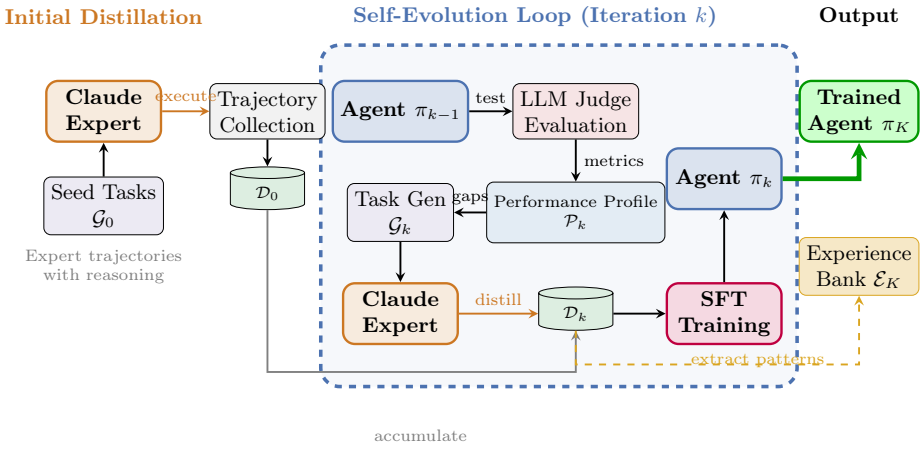

Fig. 1: Self-Evolving Policy Learning Framework. Left: Expert trajectories are collected from Claude on seed tasks $\mathcal{G}_0$. **Center**: The self-evolution loop—the agent is evaluated via LLM judge to produce a multi-dimensional performance profile $\mathcal{P}_k$ (tracking MCP/GUI ratio, difficulty, and skills), which drives targeted task and environment generation. New trajectories are accumulated with existing data for SFT training. **Right**: Two complementary outputs: (1) a fine-tuned agent π_K (effective for MCP-dominant tasks), and (2) an experience bank $\mathcal{E}_K$ that enables inference-time improvement (effective for GUI-intensive tasks).

Stage 2: Policy Refinement. Based on the profile $\mathcal{P}_k$, gap analysis identifies weaknesses and generates new tasks $\mathcal{G}_k$. The policy is then improved through trajectory distillation and experience accumulation (detailed in Section 4). The training objective is:

$$\pi_{k+1} = \arg\max_{\pi} \mathbb{E}_{\xi \sim \mathcal{D}_k}[\log \pi(a^*|s, g)] \tag{3}$$

where $\mathcal{D}_k$ is the accumulated training distribution and a^* denotes demonstrated actions—from the expert in distillation, or from the student's own successes in self-improvement (Section 4.5). The training distribution evolves across iterations:

$$\mathcal{D}_{k+1} = \mathcal{D}_k \cup \mathcal{D}_{\text{new}}(\mathcal{G}_k, \mathcal{P}_k) \tag{4}$$

where $\mathcal{D}_{\text{new}}$ contains new expert trajectories collected on tasks $\mathcal{G}_k$ generated based on performance profile $\mathcal{P}_k$.

4 Method: Self-Evolving Policy Learning

Our framework operates as an iterative improvement loop (Figure 1), with each component addressing a distinct aspect of hybrid policy learning. The two-stage structure (evaluation via profiling, improvement via SFT) mirrors policy iteration but operates offline with expert demonstrations rather than online exploration.

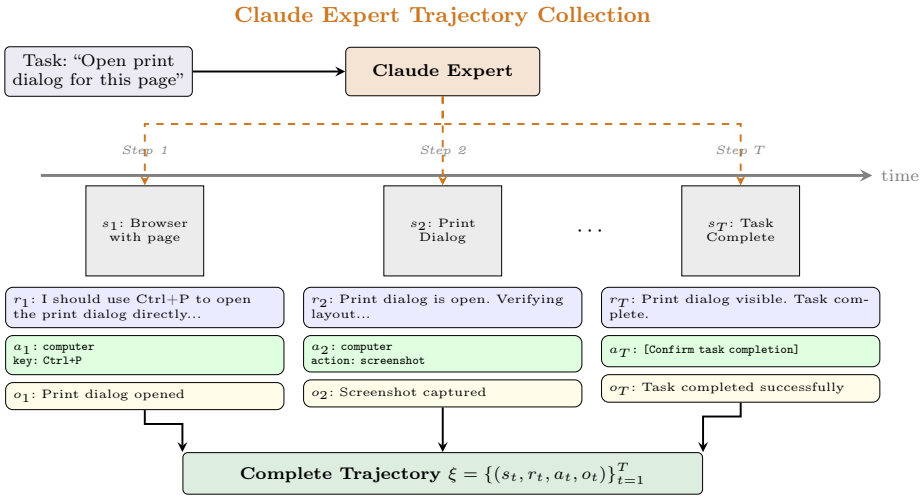

Fig. 2: Expert Trajectory Distillation. Claude executes tasks step-by-step, producing trajectories with screenshots s_t, reasoning r_t, actions a_t (MCP or GUI), and observations o_t. These serve as training data for the student model.

4.1 Expert Trajectory Collection

We use Claude Sonnet 4 as the expert policy to collect trajectories that capture the complete decision-making process (Figure 2). Each trajectory $\xi = \{(s_t, r_t, a_t, o_t)\}_{t=1}^{T}$ records the screenshot s_t, reasoning r_t, action a_t (MCP or GUI), and observation o_t. Tasks are attempted multiple times, retaining the highest-scoring trajectory. All trajectories are standardized to the model's native `<tool_call>` format.

4.2 Offline Policy Improvement

This is the *policy improvement* stage: given the profile $\mathcal{P}_k$, we improve the policy via supervised fine-tuning on expert and self-generated trajectories (this section and Section 4.5) and inference-time experience augmentation (Section 4.5); we focus here on trajectory distillation.

Training Objective. Given accumulated trajectories $\mathcal{D}_k = \{\xi_i\}_{i=1}^{N_k}$, we minimize:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{\xi \in \mathcal{D}_k} \sum_{t=1}^{|\xi|} \log p_\theta(r_t, a_t | s_{\leq t}, g) \tag{5}$$

Data Accumulation Strategy. We accumulate trajectories across iterations to prevent catastrophic forgetting:

$$\mathcal{D}_k = \mathcal{D}_0 \cup \bigcup_{i=1}^{k-1} \mathcal{D}_{\text{new}}^{(i)} \tag{6}$$

Table 1: Multi-dimensional performance profile $\mathcal{P}_k$ tracks agent capabilities across five dimensions. These metrics feed into gap analysis for targeted task generation.

Dimension	**Metrics**
1. Modality Balance	ρ_{mcp}, ρ_{gui}, ρ_{hybrid}
2. Difficulty Level	σ_{easy}, σ_{medium}, σ_{hard}
3. Skill Categories	σ_{retrieve}, σ_{manip}, σ_{search}, σ_{exec}, σ_{nav}, σ_{config}
4. Format Quality	ϕ_{format}, ϕ_{parse}, ϕ_{args}
5. Efficiency	$\bar{n}_{\text{steps}}$, ϕ_{complete}, ϕ_{timeout}
Output: Gap Analysis → Task Generation	

Critically, we reset to the base model θ_0 at each iteration rather than continuing from π_{k-1}, preventing error accumulation. To further mitigate forgetting, we mix successful trajectories from previous iterations with newly collected data via memory replay.

4.3 Policy Evaluation via Multi-Dimensional Profiling

This stage corresponds to the *policy evaluation* step in our iterative refinement framework. Rather than computing a value function as in standard RL, we use an LLM judge to evaluate the current policy across multiple dimensions, identifying systematic weaknesses that guide the next improvement iteration (Table 1).

The performance profile $\mathcal{P}_k = \langle \boldsymbol{\rho}, \boldsymbol{\sigma}_d, \boldsymbol{\sigma}_c, \boldsymbol{\phi}, \boldsymbol{\eta} \rangle$ captures modality balance, difficulty-level success rates, per-skill success rates, format quality, and efficiency metrics (Table 1).

4.4 Profile-Guided Task and Environment Generation

Based on the performance profile, we generate targeted tasks to address identified weaknesses.

Gap Analysis. We identify gaps by comparing profile metrics against target thresholds:

$$\Delta_{\text{mcp}} = \rho_{\text{target}} - \rho_{\text{mcp}} \qquad \text{(modality gap)} \tag{7}$$

$$\Delta_d = \gamma_d - \sigma_d \qquad \text{(difficulty gap)} \tag{8}$$

$$\Delta_c = \gamma_c - \sigma_c \qquad \text{(skill gap)} \tag{9}$$

where ρ_{target}, γ_d, γ_c are configurable thresholds. Tasks are generated weighted toward identified weaknesses using LLM-based synthesis. For example, at iteration 2 on Chrome the gap analysis flagged the agent as GUI-heavy with underused tools (`bookmark_page`, `delete_browsing_data`) and generated targeted tasks to close these MCP gaps. Generated task examples are in the supplementary material (Section L).

Automatic Environment Generation. A critical enabler of fully automatic evolution is **environment generation**: each generated task requires a valid application state (e.g., specific browser tabs, spreadsheet data, or open files) before evaluation can begin. Our pipeline automatically generates environment setup scripts from the task description using an LLM, then validates them by checking that the target application reaches the expected initial state. Failed environments are retried with error feedback. This eliminates the manual environment authoring bottleneck that limits scaling in existing benchmarks—each new task produced by gap analysis is immediately executable without human configuration. Details on the environment validation pipeline are in the supplementary material (Section I).

4.5 Self-Improvement and Experience Bank

A key challenge with expert distillation is the **capability gap**: the expert (Claude) can successfully execute actions that the student model cannot reliably reproduce. Training on such trajectories introduces distribution shift and may cause the student to attempt actions beyond its capabilities.

We address this with a **self-improvement paradigm** that trains exclusively on the model's own successful trajectories:

Rejection Sampling. At each iteration, we run the current model π_{k-1} on each task up to $N = 3$ attempts, retain trajectories exceeding a success threshold $\tau_{\text{success}} = 0.5$, select the best trajectory per task by LLM-judge score, and fine-tune on these self-generated trajectories.

Experience Bank: Cross-Iteration Knowledge Transfer. A key innovation is the **experience bank**—a structured knowledge base that accumulates LLM-learned patterns across iterations. Unlike simple trajectory replay, the experience bank uses an LLM to *compare* successful and failed trajectories, extracting transferable insights about what strategies work.

The experience bank is organized by **skill category** (Section 3.2), with each category maintaining four types of knowledge: (1) *task-specific strategies*—patterns that led to success (e.g., "take a screenshot first to identify the current view"); (2) *environment knowledge*—LLM-learned observations about UI layout and application behavior; (3) *tool usage patterns*—effective MCP tool sequences for common operations; and (4) *error recovery strategies*—approaches that helped recover from failures.

Experience Bank Construction. At the end of each iteration, we build the experience bank by: (1) collecting successful trajectories from both Claude and Qwen; (2) grouping by skill category; (3) using an LLM to compare successful vs. failed trajectories; (4) synthesizing concise, transferable patterns; and (5) filtering by application type to prevent cross-app contamination. Formally, for each skill category $c \in \mathcal{C}$, we store $\mathcal{E}_c = \langle \mathbf{s}_c, \mathbf{e}_c, \mathbf{t}_c \rangle$ (strategies, environment knowledge, tool patterns). Given successful trajectory ξ^+ and failed trajectory ξ^-, we extract differentiating factors via $(\mathbf{s}, \mathbf{e}, \mathbf{t}) = \text{LLM}_{\text{extract}}(\xi^+, \xi^-, c)$. When

Algorithm 1 Self-Evolution Pipeline

Input: Seed tasks $\mathcal{G}_0$, base model θ_0, iterations K
Output: Trained agent π_K, experience bank $\mathcal{E}_K$
$\mathcal{D}_0 \leftarrow \text{CollectExpert}(\text{Claude}, \mathcal{G}_0)$ ▷ Phase 1
$\pi_0 \leftarrow \text{SFT}(\theta_0, \mathcal{D}_0)$ ▷ Phase 2 (optional)
$\mathcal{E}_0 \leftarrow \text{BuildBank}(\mathcal{D}_0)$ ▷ Initial bank
for $k = 1$ to K **do**
  $\mathcal{P}_k \leftarrow \text{Evaluate}(\pi_{k-1}, \mathcal{G}_0, \mathcal{E}_{k-1})$ ▷ Policy Evaluation
  $\mathcal{G}_k \leftarrow \text{GenerateTasks}(\mathcal{P}_k)$ ▷ Gap-driven
  $\mathcal{D}_k^{\text{new}} \leftarrow \text{CollectExpert}(\text{Claude}, \mathcal{G}_k)$
  $\mathcal{D}_k \leftarrow \mathcal{D}_{k-1} \cup \mathcal{D}_k^{\text{new}}$
  $\pi_k \leftarrow \text{SFT}(\theta_0, \mathcal{D}_k)$ ▷ Policy Improvement
  $\mathcal{E}_k \leftarrow \text{BuildBank}(\mathcal{D}_k, \mathcal{P}_k)$
end for

the bank exceeds a per-category capacity limit, we merge via LLM summarization: $\mathcal{E}_c^{\text{new}} = \text{LLM}_{\text{merge}}(\mathcal{E}_c^{\text{old}}, \mathcal{E}_c^{\text{fresh}})$.

Experience-Enhanced Prompts. At inference time, prompts are augmented with relevant experience:

$$\text{prompt}_k = \text{prompt}_{\text{base}} \oplus \text{Experience}(\mathcal{E}_{k-1}, c_{\text{task}}) \tag{10}$$

where $\oplus$ denotes prompt concatenation, $\mathcal{E}_{k-1}$ is the experience bank, and c_{task} is the skill category of the current task. This enables **inference-time improvement**—agents can benefit from accumulated experience without requiring additional fine-tuning.

Evolution Modes. We evaluate two modes: **Distillation + Experience** (SFT with experience-enhanced prompts) and **Experience-Only** (experience bank augmentation without fine-tuning, suitable when compute is limited).

4.6 Self-Evolution Algorithm

Algorithm 1 summarizes the complete self-evolution pipeline. Infrastructure details including environment validation and distributed execution are in the supplementary material (Section I).

5 Experimental Setup

5.1 Environment and Applications

We evaluate across multiple desktop applications with varying MCP tool complexity (Table 2).

The applications span diverse domains: code editing (VS Code, 30 tasks), spreadsheet processing (LibreOffice Calc, 45 tasks), and web browsing (Chrome,

Table 2: Application benchmark overview. Each application has a distinct set of MCP tools, requiring the agent to adapt its modality selection strategy.

Application	Seed Tasks	MCP Tools
VS Code	30	9
LibreOffice Calc	45	16
Chrome	45	11
Total	**120**	**36**

45 tasks). Task counts reflect the available evaluation benchmarks for each domain. MCP tool counts vary across applications, requiring the agent to learn application-specific modality strategies.

5.2 Models and Training

Base Model. We use **Qwen3-VL-8B** [3] as our primary model, which supports native `<tool_call>` format with high format accuracy. Cross-model analysis with Qwen2.5-VL-7B [4] is provided in the supplementary material. All models are served with vLLM (tensor-parallel size 1, fp16) at temperature 0.2.

Training and Evolution. We use LoRA [7] for parameter-efficient fine-tuning (rank 8, LR 2×10^{-5}). The evolution loop runs $K \in \{1, 3\}$ iterations with $N = 3$ attempts per task and success threshold $\tau_{\text{success}} = 0.5$. Full training configuration details are provided in the supplementary material.

5.3 Evaluation Protocol

LLM-Judge Evaluation. We use Claude Sonnet 4 as the LLM judge: given a task goal, the initial and final screenshots, and the full trajectory, it produces a binary pass/fail decision and an LLM Score in $[0, 1]$ reflecting task completion quality. To validate the judge, we re-ran it on an identical set of trajectories and observed 93% inter-pass agreement, indicating stable verdicts. Each task is attempted with a 10-step budget at 1024×768 resolution.

Metrics. We report **Pass Rate** (binary success from the LLM judge, determined independently of the continuous score), **LLM Score** (trajectory quality in $[0, 1]$), and **MCP Ratio** (MCP vs. total actions). Our main results characterise how capability evolves on a *fixed task pool* across iterations; generalisation is assessed separately via the hold-out designs in Section 7.3.

6 Results

We evaluate our framework across three desktop applications (Chrome, VS Code, LibreOffice Calc) with 120 tasks total (Table 2), using Qwen3-VL-8B as the student model and Claude as the LLM judge.

Table 3: Chrome results after one iteration (Qwen3-VL-8B, 45 tasks). Multi-iteration trends in Table 5.

Strategy	Pass	Score	Eff.	MCP%
Baseline	60.0	0.654	0.662	43.7
+ Exp. Bank	62.2	0.640	0.646	45.0
+ Distill. + Exp.	**77.8**	**0.770**	**0.724**	36.3

Table 4: Chrome 45-task comparison with modality ablations and recent 7B peer GUI agents (Qwen3-VL-8B unless noted); same protocol as Table 3.

Method	Pass (n=45)
Hybrid, full method (Distill. + Exp.)	**77.8% (35/45)**
Hybrid, single-iteration baseline (MCP+GUI)	60.0% (27/45)
MCP-only ablation	51.1% (23/45)
GUI-only ablation	48.9% (22/45)
UI-TARS-1.5-7B [10]	24.4% (11/45)
Aguvis-7B-720p [15]	15.6% (7/45)

6.1 Single-Iteration Results on Chrome

Chrome represents an ideal setting for studying MCP–GUI synergy: tasks frequently map to direct MCP tool calls (e.g., `bookmark_page`, `navigate_to`), providing clean signal for tool invocation learning. Table 3 compares policy learning strategies in a single iteration.

Trajectory distillation yields substantial gains across all metrics. Distillation reaches 77.8% pass rate (+17.8pp over baseline), with its largest gains on hard tasks (53.3% vs. 26.7%), and attains the best LLM Score (0.770) and efficiency (0.724); the experience bank alone adds a complementary training-free +2.2pp with no fine-tuning.

The gain comes from MCP–GUI synergy, not raw modality access. To isolate this, Table 4 reports two single-modality ablations (*MCP-only* disables GUI actions; *GUI-only* disables MCP tool calls) and two recent 7B GUI agents—UI-TARS-1.5-7B [10] and Aguvis-7B-720p [15]—under the same protocol. The hybrid baseline (60.0%) exceeds GUI-only by +11.1pp and MCP-only by +8.9pp, showing the gain stems from combining the two modalities rather than from either alone; both ablations in turn surpass the pure-vision peers UI-TARS (24.4%) and Aguvis (15.6%), consistent with prior findings that GUI-native agents struggle with structured navigation that MCP tool access makes reliable.

6.2 Self-Evolution Across Iterations

A key contribution of our framework is the iterative self-evolution loop: at each iteration, the agent is first evaluated to produce a performance profile $\mathcal{P}_k$ (Section 4.3), which then drives gap analysis to automatically generate new

Table 5: Self-evolution results across 3 iterations on Chrome (Qwen3-VL-8B, 45 tasks). Pass rate (%) and LLM Score shown per iteration. Distillation achieves 77.8% in a single iteration; the experience-only variant improves steadily to 64.4% without fine-tuning. Iter 0 is the shared pre-evolution baseline; Best Iter is the peak over k iterations.

	Iter 0		**Best Iter**	
Strategy	**Pass**	**Score**	**Pass**	**Score**
Exp. Bank Only	60.0	0.654	64.4 (iter 2)	0.674
Distill. + Exp.	60.0	0.654	**77.8** (iter 1)	**0.770**

Table 6: Cross-application comparison (pass rate %; VS Code 30, Chrome/Calc 45 tasks). Distillation excels for Chrome (+17.8pp); the experience bank is more effective for GUI-intensive VS Code (+10.0pp) and Calc (+2.3pp).

	Chrome (45)	**VS Code (30)**	**Calc (45)**
Baseline	60.0	53.3	44.4
+ Exp. Bank	62.2	**63.3** (+10.0)	**46.7** (+2.3)
+ Distill. + Exp.	**77.8** (+17.8)	43.3	42.2

targeted tasks $\mathcal{G}_k$ addressing identified weaknesses. Expert trajectories are then collected on these new tasks, the experience bank is refined, and successful student trajectories are added to the training pool via rejection sampling. Table 5 presents iteration-by-iteration results on Chrome (45 tasks).

Both variants improve through complementary mechanisms. The experience-only variant improves steadily to 64.4% without fine-tuning, while distillation achieves a stronger peak (77.8% at iter 1). Each mechanism targets different failure modes: distillation teaches correct tool invocation format and syntax, while the experience bank provides strategic shortcuts such as keyboard alternatives to unreliable GUI sequences. See supplementary Sections F (Calc) and M (per-difficulty breakdown) for detailed results.

6.3 Cross-Application Analysis

Table 6 compares strategies across applications with varying MCP–GUI composition, revealing the central insight of our work.

Optimal strategy depends on task characteristics. The results validate our central claim (Contribution 1): the optimal mechanism depends on each application's task characteristics. Chrome, where many tasks map to direct MCP calls, benefits most from distillation (+17.8pp); GUI-intensive VS Code benefits from the experience bank's keyboard-shortcut rules that bypass unreliable visual grounding (+10.0pp); and Calc, despite high baseline MCP usage (61.5%), gains modestly from the experience bank (+2.3pp), as its compositional multi-step tool sequences are

harder to capture as concise rules than the single-step shortcuts (e.g., Ctrl+P) that benefit Chrome and VS Code.

Why distillation hurts certain applications. On VS Code, distillation *decreases* pass rate by −10.0pp, and on Calc MCP usage drops from 61.5% to 47.8% after SFT—evidence of **distribution mismatch**, where expert trajectories shift the student away from its already-MCP-heavy policy. The experience bank instead encodes *abstract strategies* (e.g., "use `Ctrl+Shift+P`") that augment rather than override the existing policy; thus when expert and student distributions diverge, inference-time augmentation can outperform imitation. Consistent with this, the experience bank improves *every* application training-free (+10.0pp VS Code, +2.3pp Calc, +2.2pp Chrome).

7 Analysis

7.1 Complementarity of Evolution Mechanisms

The two mechanisms address **complementary failure modes** (Contribution 1): distillation teaches *how* to invoke MCP tools (format, syntax), while the experience bank teaches *when* to use each modality. Per-iteration profiling confirms this—distillation drives hard-task gains (+26.6pp) with MCP adoption rising 43.7%→55.8%, while the experience bank drives medium-task gains (+13.4pp), and the training pool grows 44→79 via self-generated data—the loop autonomously expands its own training set. Per-difficulty breakdowns and full profiles are in supplementary Section M.

7.2 Qualitative Improvement Example

We illustrate inference-time improvement from the experience bank (Contribution 3). On a Chrome print-dialog task, the baseline called an incorrect MCP tool (`google_chrome.print`, dot syntax) and clicked randomly for 17 steps before failing; after one evolution iteration it solved the task in a single step by pressing `Ctrl+P`—a shortcut auto-discovered via trajectory comparison (Figure 3). Before/after screenshots are in the supplementary material.

7.3 Robustness across Hold-out Task Designs

To assess generalisation beyond the fixed task pool, we construct three independent 12-task hold-out designs (HO-1/HO-2/HO-3): structurally matched but independently instantiated Chrome-task variants (same MCP-tool coverage, but different websites, navigation paths, and targets), excluded from all evolution iterations and used only for post-hoc evaluation. As shown in Table 7, the ordering (Hybrid > MCP-only > GUI-only > peer agents) holds on every design with non-overlapping mean±std between adjacent methods—stronger evidence than a within-set confidence interval, as it reflects design-to-design variation, not just task-sampling noise.

Table 7: Three independent Chrome hold-out designs (12 tasks each). Method ordering is consistent across all three.

Method	HO-1	HO-2	HO-3	Mean±Std
Hybrid	8/12	7/12	9/12	**66.7±8.3**
MCP-only	7/12	5/12	7/12	52.8±9.6
GUI-only	4/12	5/12	7/12	44.4±12.7
Aguvis-7B-720p [15]	4/12	1/12	3/12	22.2±12.7
UI-TARS-1.5-7B [10]	2/12	2/12	3/12	19.4±4.8

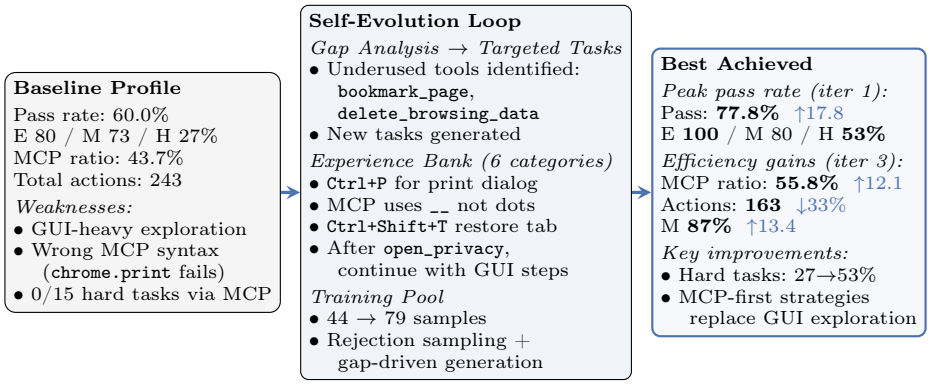

Fig. 3: Self-evolution on Chrome (Qwen3-VL-8B, Distillation + Experience Bank). **Left**: baseline profile and weaknesses. **Center**: evolution mechanisms (gap analysis, experience bank, training-pool growth). **Right**: best metrics by source iteration—peak pass at iter 1 (+17.8pp), efficiency by iter 3 (MCP +12.1pp, actions −33%). Full profiles in supplementary Section M.

7.4 What Does Self-Evolution Learn?

Figure 3 summarizes the end-to-end improvement (Contribution 2): from a baseline relying on GUI exploration with incorrect MCP syntax, the pipeline autonomously identifies weaknesses, generates targeted tasks, and accumulates experience rules—all auto-discovered by LLM trajectory comparison, none manually programmed. Three improvements emerge: (i) **MCP adoption** rises from 43.7% to 55.8% by iteration 3 (with a temporary iter-1 dip to 36.3% as the model first learns tool syntax); (ii) **total actions drop 33%** (243→163) as the agent avoids exploratory GUI sequences; and (iii) **gap analysis adapts**, moving from format errors (dot vs. underscore) early to underused tools (e.g., bookmark management) later.

7.5 Cross-Model Comparison

We compare Qwen3-VL-8B and Qwen2.5-VL-7B on Chrome to identify prerequisites for self-evolution. Qwen2.5 starts at 48.9% but *degrades* with experience

augmentation (−8.9pp), whereas Qwen3 improves (+2.2pp experience, +17.8pp distillation). Among several differences between the two models, native tool-calling appears the most directly relevant: Qwen3's `<tool_call>` format enables MCP invocations, while Qwen2.5 treats tool syntax as noise (6.6% vs. 43.7% MCP ratio). Self-evolution thus amplifies performance primarily when the base model has capability in *both* modalities; see details in supplementary Section J.

8 Discussion

Application-Aware Mechanism Selection. A central practical takeaway is that the effective training mechanism is application-dependent rather than universal. Trajectory distillation excels when the student can faithfully reproduce the expert's tool calls—as on Chrome, where it yields the largest gain (+17.8pp)—but it can *degrade* performance on GUI-intensive applications such as VS Code, where visual grounding varies across environments and imitating expert pixel-level actions transfers poorly. There, the experience bank's abstract, symbolic strategies (e.g., keyboard shortcuts in place of brittle click sequences) augment the policy more reliably. Practitioners should therefore match the mechanism to the target application's MCP–GUI composition rather than applying distillation uniformly; our multi-dimensional profiling supplies a concrete signal for making this selection automatically within the loop.

Data Efficiency and Design Choices. The pipeline bootstraps from only 44 expert demonstrations, growing to 79 via rejection sampling and gap-driven generation—far fewer environment interactions than online RL [2]. Two design choices proved critical: **per-category capacity limits** prevent context overflow in smaller models, and **application-type filtering** prevents cross-app contamination.

Limitations and Future Work. We scope our automation claim: the pipeline runs without manual intervention *given a tool-capable base model* with native function-calling. Its effectiveness depends on the base model's ability. Promising directions include further training with advanced reinforcement learning; online exploration with process rewards and better distillation for GUI-intensive tasks.

9 Conclusion

We presented a self-evolving framework for hybrid MCP-GUI agents that combines trajectory distillation, inference-time experience augmentation, and automatic environment generation, iteratively identifying weaknesses and generating targeted tasks with gains that our peer-agent and hold-out comparisons show are robust across independent task designs.

10 Acknowledgements

K.J. is supported by the UKRI Engineering and Physical Sciences Research Council (EPSRC) [EP/R513143/1].

References

1. Anthropic: Model context protocol (2024), https://modelcontextprotocol.io/, accessed: 2026-03-05
2. Bai, H., Zhou, Y., Pan, J., Cemri, M., Suhr, A., Levine, S., Kumar, A.: Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. Advances in Neural Information Processing Systems **37**, 12461–12495 (2024)
3. Bai, S., Cai, Y., Chen, R., Chen, K., et al.: Qwen3-vl technical report (2025), https://arxiv.org/abs/2511.21631, accessed: 2026-03-05
4. Bai, S., Chen, K., Liu, X., Wang, J., et al.: Qwen2.5-vl technical report (2025), https://arxiv.org/abs/2502.13923, accessed: 2026-03-05
5. Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., Su, Y.: Mind2web: Towards a generalist agent for the web. Advances in Neural Information Processing Systems **36**, 28091–28114 (2023)
6. Gulcehre, C., Paine, T.L., Srinivasan, S., Konyushkova, K., Weerts, L., Sharma, A., Siddhant, A., Ahern, A., Wang, M., Gu, C., et al.: Reinforced self-training (rest) for language modeling. arXiv preprint arXiv:2308.08998 (2023)
7. Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models (2021), https://arxiv.org/abs/2106.09685, accessed: 2026-03-05
8. Jia, H., Liao, J., Zhang, X., Xu, H., Xie, T., Jiang, C., Yan, M., Liu, S., Ye, W., Huang, F.: Osworld-mcp: Benchmarking mcp tool invocation in computer-use agents. arXiv preprint arXiv:2510.24563 (2025)
9. Koh, J.Y., Lo, R., Jang, L., Duvvur, V., Lim, M., Huang, P.Y., Neubig, G., Zhou, S., Salakhutdinov, R., Fried, D.: Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 881–905 (2024)
10. Qin, Y., Ye, Y., Fang, J., Wang, H., Liang, S., Tian, S., Zhang, J., Li, J., Li, Y., Huang, S., et al.: Ui-tars: Pioneering automated gui interaction with native agents. arXiv preprint arXiv:2501.12326 (2025)
11. Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N.A., Khashabi, D., Hajishirzi, H.: Self-instruct: Aligning language models with self-generated instructions. In: Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers). pp. 13484–13508 (2023)
12. Wang, Z.Z., Mao, J., Fried, D., Neubig, G.: Agent workflow memory. arXiv preprint arXiv:2409.07429 (2024)
13. Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T.J., Cheng, Z., Shin, D., Lei, F., et al.: Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. Advances in Neural Information Processing Systems **37**, 52040–52094 (2024)
14. Xu, Y., Lu, D., Shen, Z., Wang, J., Wang, Z., Mao, Y., Xiong, C., Yu, T.: Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. In: International Conference on Learning Representations. vol. 2025, pp. 79822–79843 (2025)
15. Xu, Y., Wang, Z., Wang, J., Lu, D., Xie, T., Saha, A., Sahoo, D., Yu, T., Xiong, C.: Aguvis: Unified pure vision agents for autonomous gui interaction. arXiv preprint arXiv:2412.04454 (2024)
16. Yan, Y., Wang, S., Du, J., Yang, Y., Shan, Y., Qiu, Q., Jia, X., Wang, X., Yuan, X., Han, X., Qin, M., Chen, Y., Peng, C., Wang, S., Xu, M.: Mcpworld: A unified benchmarking testbed for api, gui, and hybrid computer use agents (2025), https://arxiv.org/abs/2506.07672, accessed: 2026-03-05

17. Zelikman, E., Wu, Y., Mu, J., Goodman, N.: Star: Bootstrapping reasoning with reasoning. Advances in Neural Information Processing Systems **35**, 15476–15488 (2022)
18. Zhou, S., Xu, F.F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., et al.: Webarena: A realistic web environment for building autonomous agents. In: International Conference on Learning Representations. vol. 2024, pp. 15585–15606 (2024)