ASIL: Replacing Screenshot-and-Click with Structured State and Semantic Actions

Rui Xie[1,4] **Lu Chen**[1,2,3]
[1]X-LANCE Lab, School of Computer Science,
Shanghai Jiao Tong University, Shanghai, China
[2]Shanghai Innovation Institute, Shanghai, China
[3]Jiangsu Key Lab of Language Computing, Suzhou, China
[4]State Key Laboratory for General Artificial Intelligence, BIGAI, Beijing, China
sharryXR@sjtu.edu.cn chenlusz@sjtu.edu.cn
Project page: https://sharryxr.github.io/ASIL/

Abstract

Powerful code agents can execute scripts, call tools, and manage files, yet many important applications remain accessible primarily through graphical user interfaces. We argue that screenshot-and-click is an inefficient interface for software-operating agents: screenshots are state-incomplete, and GUI actions are brittle, semantically weak, and poorly matched to long-horizon planning. We introduce ASIL (Agent-Software Interaction Layer), an agent-native interface that exposes software through structured JSON observations and code-executable semantic actions, realized through the deepest feasible access path for each application. We instantiate ASIL across 15 applications and a benchmark of 300 single-application and 80 multi-application tasks. ASIL reaches above 80 with closed models while executing fewer than five actions per task. Under a repaired runtime and a 50-step screenshot budget, the same tasks yield 6.6 and 26.6 strict success under screenshot-and-click control, rising to 15.0 and 53.3 on an easier OSWorld-comparable band. Against application-native interfaces on matched tasks, ASIL exceeds LibreOffice's UNO API by 28–38 strict points but only matches draw.io's MCP content contract. The structured modality also suits training: small-scale SFT raises Qwen3.5-2B from 58.0 to 72.1 and Qwen3.5-9B from 66.6 to 80.4, and resource-limited on-policy RL further raises them to 74.4 and 82.2.

1 Introduction

Code agents have rapidly evolved from code-completion assistants into execution-oriented software engineering agents. Systems such as Claude Code and Codex operate in terminals, IDEs, CI/CD pipelines, and repository workflows, where they read and modify codebases, run commands and tests, and verify changes through tests or state checks (Yang et al., 2024; Anthropic, 2026; OpenAI, 2026). These agents succeed because they consume machine-readable project context and act through executable tools. In many practical settings, however, the last barrier to fully agentic execution is not text or code reasoning but operation of graphical software: creative tools, productivity suites, desktop utilities, and service-backed applications still expose most of their functionality through GUIs, even when the underlying systems already contain rich internal state, structured artifacts, and executable logic.

Current computer-use agents approach this barrier by imitating a human operator: they observe screenshots, infer the rendered interface state, and emit low-level clicks, drags, key presses, and text entries (Xie et al., 2024; Agashe et al., 2025a,b; Zhang et al., 2024; Tan et al., 2025; Qin et al., 2025; Wang et al., 2025; Hong et al., 2024; Cheng et al., 2024; Lin et al., 2025). This screenshot-and-click loop bakes a human-native interface into the core of the agent. The observation side is misaligned because a screenshot is not software state but only its visible projection, omitting hidden panels, background processes, document structure, and internal metadata, and forcing repeated multimodal inference over a presentation layer. The action side is misaligned because GUI events are coordinate-sensitive motor primitives: a single semantic intent (rename a layer, modify a property, trigger an export) expands into long brittle sequences that depend heavily on grounding and break under layout or theme changes. Recent improvements to visual grounding and UI parsing (Yang et al., 2023; Cheng et al., 2024; Lu et al., 2024; Lin et al., 2025) still operate within this same low-semantic action space. The central question is therefore not how to make models better at looking at screens, but what obser-

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 39611–39629
October 24-29, 2026 ©2026 Association for Computational Linguistics

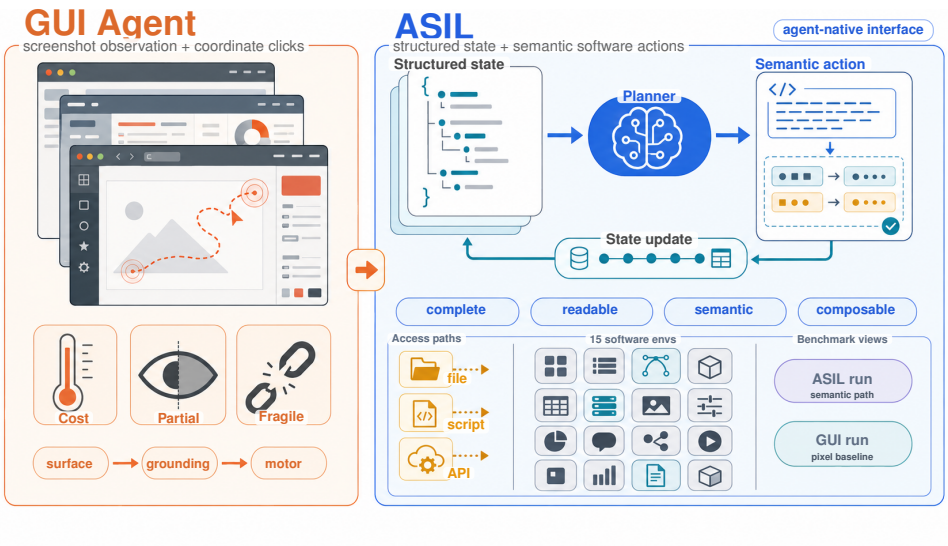

Figure 1: **Overview of ASIL.** Conventional GUI agents bind software operation to a human-native screenshot-and-click loop, requiring visual grounding over a partial pixel surface and brittle motor sequences. ASIL replaces this interface with structured observations of software state and code-executable semantic actions, realized through file-, script-, and service-level access paths across 15 software environments and 380 benchmark tasks.

vation and action interfaces a software-operating agent should natively use.

We address this mismatch with **ASIL** (Agent-Software Interaction Layer), an agent-native interface in which the native observation is a structured JSON representation of software state and the native action is a code-executable semantic operation. ASIL is not tied to a single backend: the same agent form is realized through whichever interface most directly exposes stable state and meaningful operations – structured file formats, native scripting runtimes, or service APIs. We instantiate ASIL across 15 applications, implement a benchmark with 300 single-application and 80 multi-application tasks, and render per-step GUI snapshots from the underlying software state so that ASIL and screenshot-driven runs can be compared under the same task definitions and validators. Figure 1 summarizes this transition and its realization pathways.

This paper makes five contributions:

- We formulate GUI-agent failure as an interface problem and define ASIL as a software-operating agent form that replaces screenshot observations and GUI events with structured JSON observations and code-executable semantic actions.
- We present a realization methodology that maps this form onto heterogeneous software through file, scripting, and service access paths, together with a semi-automatic onboarding framework that compiles reviewed interface profiles into validated adapter contracts.
- We provide an implemented 380-task benchmark with shared validators and visual renderings that supports direct ASIL/GUI comparison across 15 applications.
- We add repaired GUI / 50max results, matched native-interface baselines against LibreOffice UNO and draw.io MCP, and independent validity checks that bound the comparison.
- We show that the same modality is practical for training: small-scale SFT plus resource- and time-limited on-policy RL yield double-digit gains on Qwen3.5-2B and Qwen3.5-9B, and we confirm these findings under a curated 80-task hard suite and a realization-pattern ablation.

2 Related Work

GUI agents and computer-use benchmarks have demonstrated that large multimodal models can operate real software through human-facing interfaces (Nguyen et al., 2025a; Xie et al., 2024; Agashe et al., 2025a,b; Zhang et al., 2024; Tan et al., 2025; Qin et al., 2025; Wang et al., 2025; Xie et al., 2026). These systems typically frame software use as a screenshot-centered observation problem and a GUI-event action problem: the agent interprets rendered screens, grounds controls, and emits clicks, keystrokes, drags, or other human-like operations. Recent work also reduces domain bias in GUI agents by retrieving web tutorial videos and

injecting plug-and-play planning and grounding annotations (Xie et al., 2026). Our work shares the goal of making agents operate real software, but studies a different interface question: whether the screenshot-and-click loop should remain the default substrate for software-operating agents.

Several recent systems improve GUI-agent capability by adding scripts, tools, APIs, or compound action layers. OS-Copilot, UFO2, and DynaSaur introduce OS-level skills, hybrid GUI–API control, or dynamically generated programmatic actions (Wu et al., 2024; Zhang et al., 2025; Nguyen et al., 2025b). AXIS and API-based web agents show that routing intent through APIs rather than pixels improves efficiency in their domains, and declarative-interface analyses make the same argument for computer-use agents more broadly (Lu et al., 2025; Song et al., 2025; Wang et al., 2026). CLI-Anything and OpenCLI expose higher-level command or DOM interfaces through agent-facing CLIs and adapters (Yang et al., 2026; jackwener, 2026); application-specific MCPs and native interfaces such as draw.io MCP and LibreOffice UNO provide additional programmatic surfaces. ASIL shares this goal of making software callable, but its evaluated contract couples normalized observations, schema-constrained semantic actions, stable identifiers where exposed, final-state validators, serialized trajectories, and reusable traces/rewards across inference, SFT, and RL. Table 5 in Appendix A summarizes the contract-level distinction, and Section 5.2 reports matched baselines rather than claiming universal dominance over application-specific interfaces.

Code agents provide the closest positive precedent. SWE-agent shows that a purpose-built agent-computer interface improves repository navigation, file editing, and test execution (Yang et al., 2024), and CodeAct shows that executable Python can serve as a unified action space for LLM agents (Wang et al., 2024). Claude Code and Codex further show that agents become powerful inside environments with readable project state, executable commands, and verifiable feedback (Anthropic, 2026; OpenAI, 2026). ASIL transfers this lesson from code repositories to existing GUI software, reconstructing the agent-software interface around each application's file formats, scripting runtimes, service APIs, and verifiable state, and thereby lowering the barrier between agents and the large body of software functionality currently trapped behind human-oriented GUIs.

3 Interface Mismatch and ASIL

3.1 Why Screenshot-and-Click is Inefficient

The coupling of screenshot observation with low-level GUI events is doubly inefficient. Every step pays for visual encoding and multimodal reasoning before any task-level planning, while the low-level action space expands a single semantic goal into many atomic events. Current OSWorld-Verified results make this pressure visible: many simple tasks are evaluated with budgets above 50 steps and close to 100 steps (Xie et al., 2024; XLANG Lab, 2025). At inference time, high per-step latency multiplied by long trajectories makes even simple desktop tasks slow. At training time, RL rollouts repeatedly pay for screenshot acquisition, model calls, GUI execution, state waiting, and validation across many turns. Long trajectories also lower reliability: GUI actions are highly dependent across steps, so one grounding, focus, layout, or timing error corrupts every state that follows, and recovery requires re-grounding in a changed interface before repair.

3.2 ASIL Protocol Objects

ASIL replaces this loop with a structured environment. The agent receives an OBSERVATION object that organizes the current software state into task metadata, application state, interactive elements, environment context, navigation structure, and a concise textual summary; this preserves exactly the information that screenshot-based agents struggle to recover (which document is active, which entities are editable, what background conditions affect correctness). The agent then emits an ACTION object whose type, target, and parameters are defined by the application's action schema. Depending on the software, an action may modify a structured file, execute a native script, call a service endpoint, navigate within an internal topology, or trigger a batch operation – moving the action space from "click here" to "set this value" or "invoke this function." A single semantic action can encapsulate a long GUI sequence, and its post-action state is checked directly from the next structured observation.

We develop ASIL around four design principles. *Completeness* exposes relevant software state beyond the visible surface. *Semanticity* aligns actions with meaningful software operations rather than GUI mechanics. *Stability* keeps identifiers and targets valid under presentation changes. *Composability* expresses complex tasks through a small number

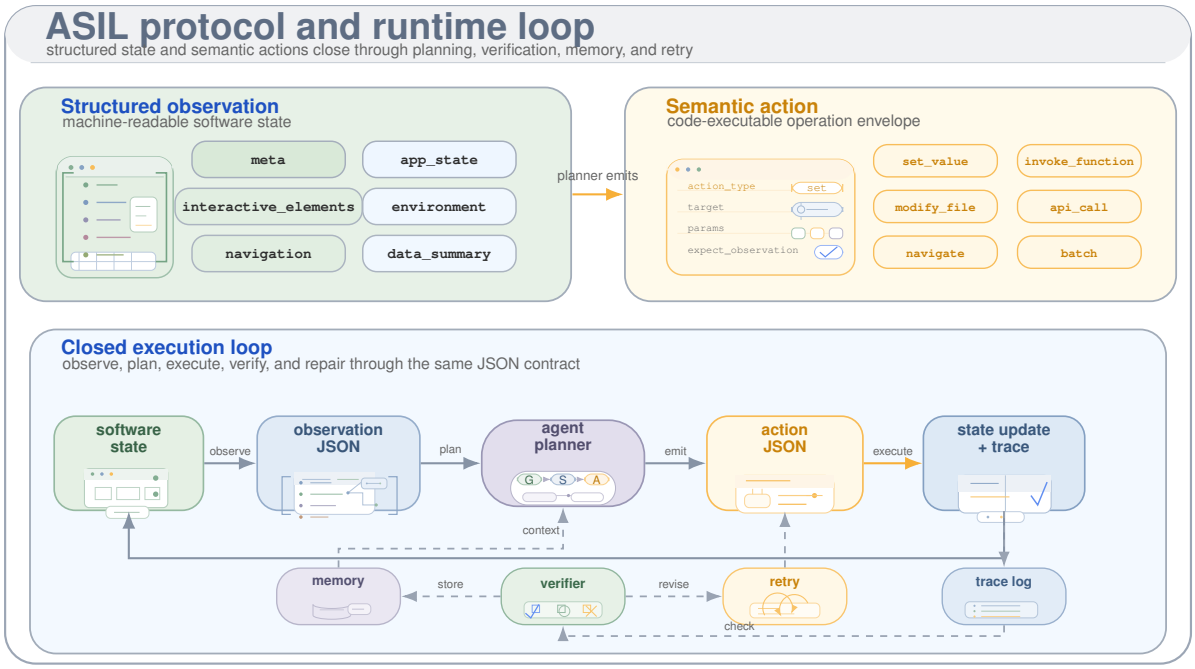

Figure 2: **ASIL protocol and runtime loop.** ASIL exposes software as structured observations and code-executable semantic actions, then closes the loop through planning, verification, memory, retry, and inspectable state updates under the same JSON contract.

of higher-level actions. Figure 2 shows the protocol objects and the resulting observe-act-check loop.

Formally, ASIL substitutes the screenshot-and-click interface $(\mathrm{Pix}, \mathcal{M}^k)$ with $(\Phi, \mathcal{A})$ in a single observation–action–transition loop:

$$o_t = \Phi(s_t), \quad a_t \sim \pi_\theta(\cdot \mid o_t), \quad s_{t+1} = T(s_t, a_t), \tag{1}$$

where $\mathcal{S}$ is the latent software state, $\mathcal{O}$ the structured observation space, and $\mathcal{A}$ the schema-constrained semantic action space. On the observation side, Pix collapses task-determining attributes outside the rendered viewport (hidden panels, document properties, background state) into indistinguishable equivalence classes, whereas Φ is engineered to be near-injective on the task-relevant subspace by construction. On the action side, one semantic $a \in \mathcal{A}$ realizes the same transition as a sequence $(m_1, \ldots, m_k) \in \mathcal{M}^k$ with $k \gg 1$, lowering per-step cost and shortening the RL credit-assignment horizon.

4 Realizing ASIL in Real Software

4.1 ASILization Pipeline and Realization Patterns

ASIL is realized through a semi-automatic ASILization pipeline. For each application, we first identify the deepest feasible access path: the interface that exposes stable state and semantic operations closest to the software's real transition system while remaining practical to run and evaluate. The resulting application adapter implements a shared observe-execute-validate contract, so heterogeneous applications expose the same ASIL interface. The current system uses three recurring realization patterns – file-backed execution (e.g., SVG, ODF, notebooks), native scripting execution (e.g., Blender Python), and service or API execution (e.g., REST, WebSocket) – as implementation pathways for the same protocol rather than separate agent types. JSON is the normalized agent-facing representation, not a requirement that software natively stores JSON: reviewed file parsers, scripts/commands, and APIs all map state into the same schema. Across the fifteen implementations, six are file-backed, four native-script, and five service/API realizations.

Eligibility and onboarding. An application qualifies when it exposes at least one open read path plus a semantic-action path. Coverage is therefore best read at the level of software *functions*: for most common functions there exists at least one qualifying application, even when a particular closed product does not qualify. Functions available only through opaque software with no

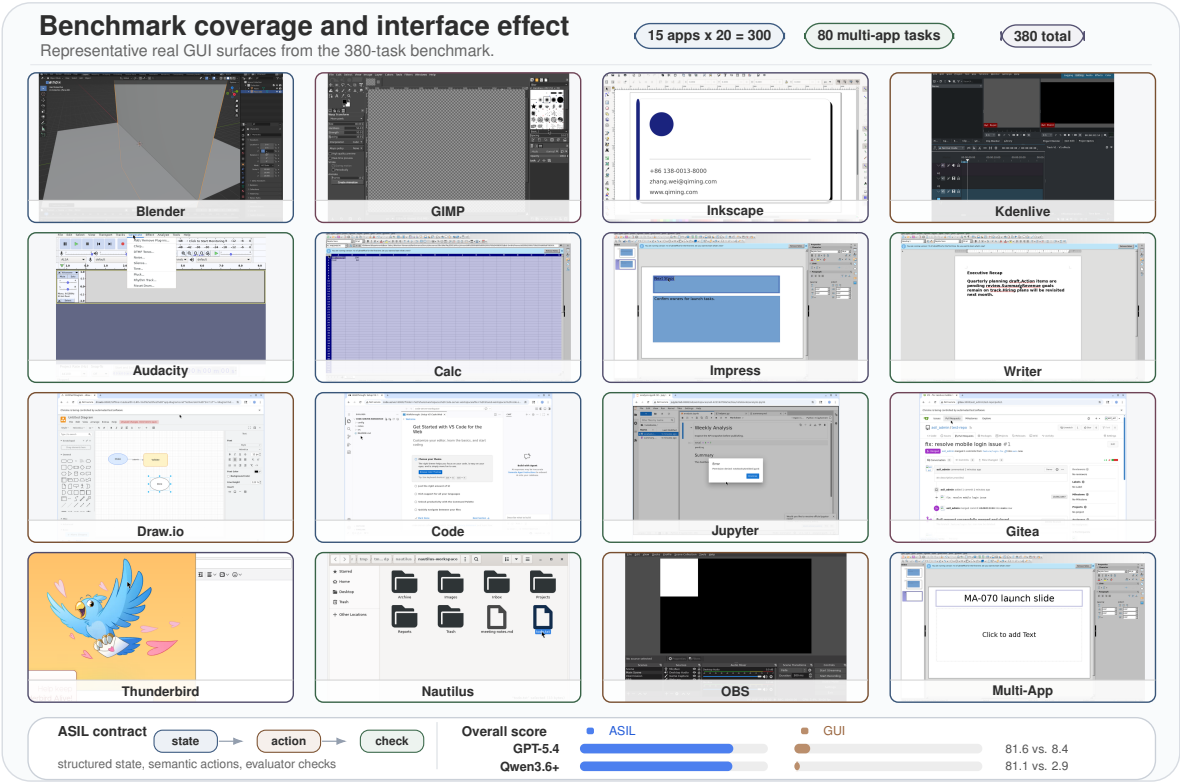

Figure 3: **Benchmark coverage and interface effect.** The benchmark spans 15 single-application environments and 80 multi-application workflows. The main panel shows representative real GUI screenshots from GPT-5.4 and Qwen3.6-plus benchmark result directories; the compact summary highlights the same-task ASIL-vs-GUI interface gap under the combined 380-task evaluation.

parseable file, scripting, structured-command, or service surface remain out of scope. The repeatable onboarding work is automated after a reviewed interface profile: one GPT-5.4 call compiled a 97-line Gitea API profile in 24.8 seconds into one observation view and two semantic actions with zero audit errors, then passed 3/3 host and 3/3 Docker probes. Interface discovery, task/evaluator design, application-specific bridges, and GUI synchronization remain human-reviewed; Appendix B separates these stages.

4.2 Benchmark Construction

Figure 3 summarizes the benchmark's application coverage and the observed ASIL-vs-GUI interface effect using representative real screenshots from the evaluated runs.

We implement a single benchmark that contains 300 single-application tasks and 80 multi-application tasks. The single-application portion covers 15 software domains with 20 tasks each, including creative tools, productivity applications, service-backed systems, code and workspace environments, and desktop utilities. Representative environments include Inkscape, Blender, GIMP, Audacity, Kdenlive, LibreOffice Calc, Writer, and Impress, together with OBS, Gitea, code-server, JupyterLab, Thunderbird, and Nautilus. The multi-application portion requires information or artifacts to move across application boundaries. This coverage tests whether ASIL is a general software-operating agent form rather than a technique specialized to one application family.

The benchmark is implemented as a shared evaluation system rather than as separate ASIL and GUI task suites. The same task definitions, initial artifacts, state validators, and result directories are used across interface modes. In ASIL mode, the participant receives structured observations and emits JSON semantic actions; in GUI mode, the participant receives screenshots and emits GUI-event actions in real software sessions. ASIL runs also render per-step GUI or page snapshots from the underlying software state, producing visual artifacts that can be inspected alongside screenshot-driven GUI-agent runs. File-level artifact details are given in Appendix B.

The evaluator checks final software state rather

than surface-level action histories. Because the same evaluator is used for deterministic execution, ASIL agent runs, GUI runs, SFT filtering, and RL rewards, the benchmark keeps task success criteria software-aware and replayable across all reported settings. ASIL prompts in the main benchmark receive evaluator-derived textual success hints, whereas GUI prompts receive only the task instruction; we therefore do not claim that the original main-table comparison isolates the interface variable alone. Every comparison added in this camera-ready version – matched native-interface baselines, the visual-supplement ablation, and the hint-off arm of the balanced audit – is run hint-off on both sides, and Section 5.2 quantifies the hint effect.

4.3 Training Setup under the ASIL Modality

The original motivation for ASIL also suggests a practical training direction: structured observations, semantic actions, and evaluator traces are easier to serialize, verify, and reuse than screenshot-and-click traces. We therefore run supervised and reinforcement-learning studies under the ASIL modality, using the same evaluator-backed runtime as the benchmark. Figure 4 summarizes this pipeline as a single closed loop: verified ASIL trajectories supply both the replay-based SFT datasets and the on-policy rollout data, the policy is updated against rewards produced by the same evaluator used at inference time, and the resulting checkpoints are then scored on the same 380-task benchmark, with the 9B family's training deltas amplifying by an additional 3–4 points on the hard suite of Section 5.4.

The setup has three components: a low-overlap training task pool generated separately from the evaluation benchmark (a learnable 320/80 RL subset is drawn from a final 512/128 v3 pool covering all 15 domains); step-level SFT traces from known-correct ASIL actions and verified GPT-5.4 rollouts; and evaluator-backed on-policy RL through an external ASIL AgentService, so rewards and rollouts share the same observations, schemas, adapters, and validators as inference-time evaluation.

Appendices C and D give the full SFT/RL settings and the task-generation overlap audit.

5 Experiments

5.1 Main Benchmark

Table 1 reports the main 380-task benchmark. The single-application columns group the 15 software environments (20 tasks each) by domain region, and Multi-App is reported as a peer region; Overall aggregates all 380 tasks. ASIL and GUI share the same task definitions, initial states, and validators. ASIL uses the default 15-step budget and averages fewer than five executed actions; the repaired GUI rows use a 50-step native-computer-use budget, with one additional row truncating the same sonnet4.6 trajectories at 15 actions.

The gap remains large after repair and after tripling the GUI budget. GPT-5.4 reaches 81.6 under ASIL versus 6.6 under GUI; sonnet4.6 reaches 81.2 versus 26.6, and 17.9 when the same GUI trajectories are restricted to ASIL's 15-step budget. The repaired GUI rows also differ sharply from each other, so the GUI side is not uniformly weak; a strong native computer-use model can recover many single-application tasks. The aggregate still supports the central claim: structured observations and semantic actions make software operation shorter, more stable, and more verifiable than screenshot-and-click control. Figure 5 in Appendix A illustrates this failure mode on a LibreOffice task, where GUI control spends its budget in a repeated keypress paste loop while ASIL writes the spreadsheet state with one `modify_file` action.

5.2 Repaired GUI Bands, Native Baselines, and Validity

A single GUI aggregate hides strong dependence on task shape. On *easy60*, a single-application band drawn from the same 380 tasks and calibrated by structural proxies against OSWorld-369, the repaired GUI / 50max rows rise from 6.6 to 15.0 strict for GPT-5.4 and from 26.6 to 53.3 for sonnet4.6 (mean scores 18.3 and 54.2). We treat this as an OSWorld-comparable reference band, not an exact difficulty match; the full benchmark is deliberately harder because it includes complete workflows with no pre-staged intermediate progress.

Some applications also expose native programmatic interfaces, so we compare against those directly in Table 2. On 60 LibreOffice tasks, ASIL exceeds native UNO by 28–38 strict points even though UNO exposes the full automation surface, indicating that a normalized observation and contracted action layer can be easier for agents to use than a low-level native API. On 20 draw.io tasks, ASIL is level with draw.io's MCP content contract for GPT-5.4 and behind it for sonnet4.6. The claim is therefore compositional rather than per-application dominance: ASIL contributes one

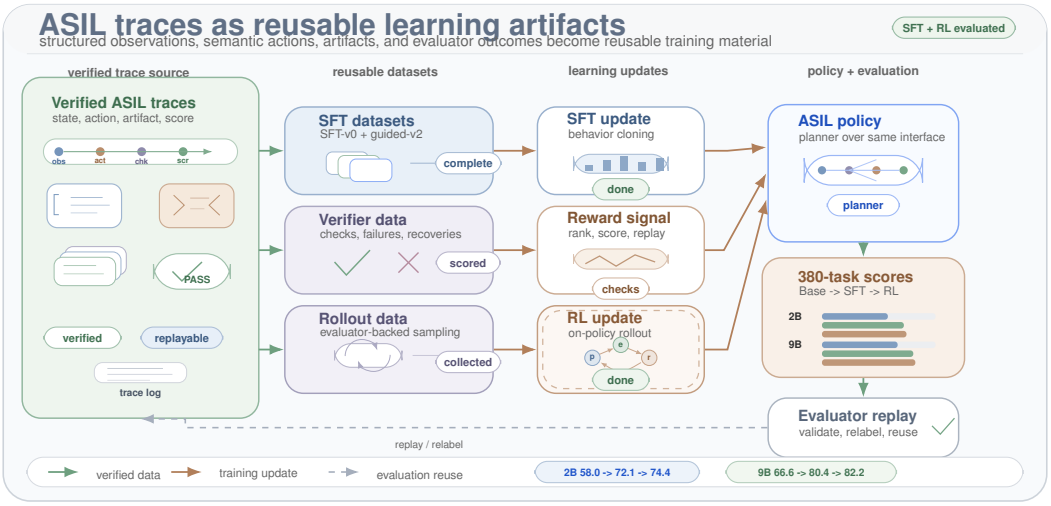

Figure 4: **ASIL traces as reusable learning artifacts.** Verified ASIL observations, actions, artifacts, and evaluator outcomes are reusable across supervised fine-tuning and evaluator-backed rollout training. The same loop drives both stages: SFT updates the policy from replayed and guided traces, while on-policy RL updates it from rollouts whose rewards come from the same ASIL evaluator. Both stages produce final 380-task gains on Qwen3.5-2B and Qwen3.5-9B, and the 9B family's training deltas further amplify by 3–4 points on the curated 80-task hard suite reported in Section 5.4.

cross-application contract that couples observation, action, validation, and reusable trajectories. Where a mature native agent interface already exists, the right use of ASIL is to ingest it as an access path.

We also check measurement validity. A fail-closed raw validator that does not call the ASIL observation builder agrees with the official evaluator on 60/60 balanced final states. The prompt success hint is measurable: on balanced-30, GPT-5.4 scores 29/30 strict with evaluator hints and 26/30 without them. A GIMP visual-supplement ablation (44 tasks, no hint, 50 steps) gives 44.1 mean for ASIL-only and 43.7 with per-step screenshots, suggesting the current bottleneck is perceptual action vocabulary rather than missing screenshot observations.

5.3 Training Results under ASIL

The open-model rows provide a second result: ASIL is not only a stronger inference interface, but also a more sample- and compute-efficient training modality. Empirically, GUI-agent training is comparatively harder and more expensive because screenshot observations and low-level GUI actions make each rollout pay for repeated visual processing, long action horizons, and brittle state recovery. Under ASIL, Qwen3.5-2B improves from 58.0 to 72.1 with SFT and to 74.4 with resource- and time-limited on-policy RL. Qwen3.5-9B improves from 66.6 to 80.4 with SFT and to 82.2 with RL. These double-digit gains are obtained from a deliberately small training setup: the final SFT stage uses only thousands of step-level ASIL samples, and the final RL runs use 320 training tasks and 80 validation tasks with 4 A800 GPUs for 2B and 8 A800 GPUs for 9B. Appendix C gives the full training settings.

The gains are meaningful but not uniform. For Qwen3.5-2B, SFT improves Files & Communication by 39.8 points but slightly trails the base model on Multi-App; RL recovers Multi-App and reaches the best 2B overall score. For Qwen3.5-9B, SFT contributes the largest improvements in Multi-App (+35.2) and Office & Diagrams (+27.2), while RL further improves the overall score to 82.2 and reaches the best Multi-App score among all reported rows. We therefore read the training results as evidence for the training advantage predicted by the interface design: small-scale SFT and RL become effective under ASIL with much less rollout burden than screenshot-and-click training would require.

5.4 Hard-Task Evaluation

We further evaluate the selected checkpoints on a held-out 80-task hard suite emphasizing long-horizon, multi-constraint, deliverable-grade workflows: real-photo editing and annotation in GIMP, cross-application artifact transfer, office and diagram deliverables, developer and service workflows, and file/email operations. Each task requires

Model setting	Mode	Creative (120)	Office (80)	Developer (60)	Files (40)	Multi-App (80)	Overall (380)
GPT-5.4	ASIL / 15max	75.8	**85.0**	91.7	92.5	73.8	81.6
	GUI / 50max†	11.7	1.2	6.7	12.5	1.2	6.6
Qwen3.6-plus	ASIL / 15max	82.5	78.8	91.7	87.5	70.0	81.1
Kimi K2.5	ASIL / 15max	**84.4**	79.5	**95.6**	86.2	81.7	**84.8**
sonnet4.6	ASIL / 15max	81.2	74.3	89.8	94.4	75.1	81.2
	GUI / 50max†	29.2	7.5	51.7	72.5	0.0	26.6
	GUI / 15max†	23.3	6.2	36.7	32.5	0.0	17.9
Qwen3.5-27B	ASIL / 15max	79.1	72.7	92.1	98.3	50.2	75.8
Qwen3.5-2B Base	ASIL / 15max	49.5	48.7	68.4	52.7	75.1	58.0
Qwen3.5-2B SFT	ASIL / 15max	67.8 (+18.3)	68.9 (+20.2)	70.6 (+2.2)	92.5 (+39.8)	72.5 (-2.6)	72.1 (+14.1)
Qwen3.5-2B RL	ASIL / 15max	69.7 (+20.2)	69.1 (+20.4)	79.8 (+11.4)	77.7 (+25.0)	81.0 (+5.9)	74.4 (+16.4)
Qwen3.5-9B Base	ASIL / 15max	67.5	48.3	87.3	97.5	52.7	66.6
Qwen3.5-9B SFT	ASIL / 15max	72.1 (+4.6)	75.5 (+27.2)	91.4 (+4.1)	83.8 (-13.8)	87.9 (+35.2)	80.4 (+13.8)
Qwen3.5-9B RL	ASIL / 15max	72.2 (+4.7)	73.5 (+25.3)	92.6 (+5.3)	**98.8** (+1.2)	**89.6** (+36.9)	82.2 (+15.5)

Table 1: Main 380-task benchmark results. Column heads show task counts; full region membership is in Appendix A. †GUI / 50max rows are this round's repaired native-computer-use re-test; GUI / 15max truncates the same trajectories. Parentheses are point changes over the corresponding ASIL base model; bold and underline mark the best and second-best scores per column.

Setting	Interface	Strict	Mean
GPT-5.4 full 380	GUI / 50max	6.6	11.6
sonnet4.6 full 380	GUI / 50max	26.6	30.3
GPT-5.4 easy60	GUI / 50max	15.0	18.3
sonnet4.6 easy60	GUI / 50max	**53.3**	**54.2**
GPT-5.4 LibreOffice	ASIL / UNO	95.0 / 66.7	97.5 / 72.8
sonnet4.6 LibreOffice	ASIL / UNO	98.3 / 60.0	99.5 / 69.2
GPT-5.4 draw.io	ASIL / MCP	55.0 / 55.0	80.9 / 80.7
sonnet4.6 draw.io	ASIL / MCP	25.0 / 55.0	46.7 / 81.9

Table 2: Camera-ready calibration results (%). Top: repaired GUI / 50max by task band. Bottom: matched native-interface baselines, run on identical tasks with the same evaluator, model, 15-step budget, and no success hint.

6 to 15 meaningful semantic actions. The suite is ASIL-only because Table 1 already shows repaired GUI / 50max control far below ASIL on the easier 380-task benchmark, and preliminary GPT-5.4 GUI trials confirmed the same collapse on the hard suite. Appendix A gives the task composition and evaluation setup.

The hard suite sharpens the training picture. For Qwen3.5-9B, the SFT and RL gains over the base model increase to +17.1 and +19.8 points, larger than the corresponding +13.8 and +15.5 gains on the main benchmark. The strongest improvement appears in Multi-App, which rises from 33.9 to 73.9 to 77.2 across Base, SFT, and RL. This is the regime where semantic actions should matter most: long workflows make low-level GUI traces fragile, while ASIL exposes higher-level operations and evaluator-backed rewards.

For Qwen3.5-2B, the picture is more limited. SFT improves the base model by +6.0 on the hard suite, while the selected RL checkpoint reaches only +3.6 and trails the 2B SFT checkpoint. This indicates that the 2B RL operating point that is best on the broader benchmark does not transfer uniformly to the difficulty tail. Creative real-photo editing remains the main bottleneck for all rows, suggesting a residual access-path and data-coverage limitation rather than a pure interface failure.

5.5 Realization-Pattern Ablation

Finally, we test whether the result depends on a single adapter style by probing the deepest-feasible access principle of Section 4. Table 4 reports two probes: a cross-application pattern audit over the 300 single-application tasks, and a within-application path-restriction study on LibreOffice.

The interface effect is large in every realization pattern: for GPT-5.4, ASIL beats repaired GUI / 50max by 89.5, 84.5, and 77.1 points in the file-backed, scripting, and service/API buckets. The sonnet4.6 GUI row shows that service/API tasks are the easiest band for screenshot control, but still below the ASIL row. Training gains are more pattern-dependent. Qwen3.5-9B RL improves most in the file-backed bucket (+18.8) and in service/API (+7.7), but is nearly unchanged in native scripting (-0.9), where the base model is already strong. We therefore do not claim that RL improves every access pattern equally.

The LibreOffice path-restriction study separates realization pattern from application identity. GPT-5.4 reaches exactly 98.6 under both the file-backed and script-dispatch paths, and Qwen3.5-9B RL dif-

Model setting	Mode	Creative (24)	Office (12)	Developer (8)	Files (6)	Multi-App (30)	Overall (80)
GPT-5.4	ASIL / 15max	14.7	**85.9**	**91.7**	**100.0**	73.9	**61.7**
Qwen3.5-2B Base	ASIL / 15max	13.3	40.8	41.1	20.8	72.8	43.1
Qwen3.5-2B SFT	ASIL / 15max	8.0 (-5.3)	50.3 (+9.5)	66.5 (+25.4)	**100.0** (+79.2)	66.7 (-6.1)	49.1 (+6.0)
Qwen3.5-2B RL	ASIL / 15max	10.1 (-3.2)	54.2 (+13.4)	64.0 (+22.9)	83.3 (+62.5)	61.1 (-11.7)	46.7 (+3.6)
Qwen3.5-9B Base	ASIL / 15max	12.4	28.3	84.4	**100.0**	33.9	36.6
Qwen3.5-9B SFT	ASIL / 15max	14.5 (+2.1)	57.5 (+29.2)	80.3 (-4.1)	66.7 (-33.3)	73.9 (+40.0)	53.7 (+17.1)
Qwen3.5-9B RL	ASIL / 15max	**15.2** (+2.8)	41.4 (+13.1)	**91.7** (+7.3)	**100.0** (+0.0)	**77.2** (+43.3)	56.4 (+19.8)

Table 3: Held-out 80-task hard-suite results under ASIL / 15max mode. Region breakdown (24 GIMP, 12 Draw.io+LibreOffice, 8 code-server+Gitea+JupyterLab, 6 Nautilus+Thunderbird, 30 multi-app) and full setup in Appendix A; parentheses are point changes over the corresponding ASIL base model.

(a) Cross-application pattern audit					**(b) LibreOffice path-restriction study**				
Participant	Mode	A (file)	B (script)	C (api)	Participant / realization	Calc	Writer	Impress	Overall
GPT-5.4	ASIL / 15max	**91.2**	**89.5**	**95.1**	GPT-5.4 / file	**100.0**	98.4	97.4	**98.6**
GPT-5.4	GUI / 50max	1.7	5.0	18.0	GPT-5.4 / script-dispatch	**100.0**	98.4	97.4	**98.6**
sonnet4.6	GUI / 50max	10.8	21.2	71.0	Qwen3.5-9B RL / file	35.6	**100.0**	**100.0**	78.5
Qwen3.5-9B Base	ASIL / 15max	53.0	78.8	84.4	Qwen3.5-9B RL / script-dispatch	40.6	**100.0**	99.0	79.9
Qwen3.5-9B RL	ASIL / 15max	71.8	77.9	92.1					

Table 4: Realization-pattern ablation. Panel (a) groups single-application tasks by Pattern A file-backed, Pattern B native-scripting, and Pattern C service/API realizations; GUI / 50max rows are recomputed from this round's repaired re-test. Panel (b) compares file-backed and script-dispatch access paths on the same 60 LibreOffice tasks.

fers by only 1.3 points. Since GUI control is near zero on the same LibreOffice family in Table 1, both deep access paths sit far above the GUI baseline. The evidence supports the deepest-feasible access principle without implying that one backend pattern is universally optimal.

5.6 Failure Analysis

The remaining failures fall into two classes. GUI runs mainly fail from missing hidden state, grounding errors, and brittle event sequences. These errors explain the low repaired GUI rows in Table 1 and the GPT-5.4 GUI / 50max scores of 1.7, 5.0, and 18.0 across realization patterns in Table 4. ASIL removes this class by changing the interface.

ASIL runs instead fail from residual modeling and coverage limits. The clearest example is Creative real-photo editing in the hard suite: even the strongest row averages only 15.2 on GIMP-heavy tasks, because some target conditions depend on pixel-level image semantics that the current semantic action vocabulary does not fully cover. Other failures come from incorrect semantic-action choice or from underrepresented application families in the training traces, and the Thunderbird regression shows that supervised training can also disturb a behavior pattern the base model already handles. Figure 6 in Appendix A illustrates these three failure modes. The practical implication is that after the interface bottleneck is removed, progress should come from richer adapters, broader trace coverage, and stronger training, not from further optimizing screenshot-and-click control.

6 Conclusion

This paper argues that the central obstacle in operating GUI software is not task difficulty but an interface mismatch between human-native screenshot-and-click and agent-native semantic operation. ASIL replaces this loop with structured software state and code-executable semantic actions, realized through the deepest feasible access path for each application – a more software-native agent form, not an extra tool layer.

On a 380-task benchmark across 15 applications, ASIL obtains strong inference results with short semantic trajectories, and the same modality enables practical training: small-scale SFT and on-policy RL yield double-digit gains on Qwen3.5-2B and Qwen3.5-9B. The comparative claim is deliberately bounded. A repaired 50-step GUI re-test reaches 6.6 and 26.6 strict success on the full benchmark and 15.0 and 53.3 on an easier OSWorld-comparable band; against native interfaces, ASIL clearly exceeds LibreOffice UNO but only matches draw.io MCP. Software operation is best treated as interaction over state and verifiable artifacts, not over pixels and motor traces.

Limitations

We discuss four limitations tied to the paper's core claims: prompt asymmetry in the original comparison, small-model RL stability on long-horizon tasks, realization coverage gaps for fully opaque applications, and tasks whose success criteria are intrinsically perceptual.

Prompt asymmetry and evaluator reuse. The submitted main ASIL prompts include evaluator-derived success hints, while GUI prompts do not. The hint has a measurable effect (29/30 versus 26/30 strict on balanced-30), and the same evaluator also filters SFT data and supplies RL rewards. We therefore do not claim that the original main table isolates only the interface variable. The camera-ready baselines added in Section 5.2 use hint-off comparisons and an independent raw-state validator, but broader independent validation across all 380 tasks remains future work.

2B training stability on long-horizon tasks. The Qwen3.5-2B training gradient amplifies cleanly on the main 380-task benchmark (+14.1 and +16.4 points for SFT and RL over the base model) but only partially on the 80-task hard suite, where the gains shrink to +6.0 and +3.6 and the selected 2B RL checkpoint falls 2.4 points behind the 2B SFT checkpoint. We read this as the small-model RL operating point rather than an interface problem: `global_step_8` maximizes the aggregate ASIL-380 score but does not consistently improve long-horizon behavior on Multi-App and Creative. Finding a 2B RL setting that retains the main-benchmark gain on long-horizon tasks – for example, a longer schedule, a hard-task-weighted curriculum, or a different checkpoint aggregation – remains future work. The Qwen3.5-9B family does not show this pattern.

Coverage gaps for fully opaque applications. ASIL is realized through the semi-automatic ASILization pipeline of Section 4, which substantially lowers per-application setup cost. The main remaining limitation is extending ASIL to applications that are simultaneously *closed-source*, use *file formats that are hard to unpack*, and *lack rich external scripting or service interfaces*: the three realization patterns (file, scripting, service) all require at least one such access door to be open, so applications with all three closed cannot expose a deep feasible access path under our current methodology. The 15-application instantiation reported here focuses on software where at least one access door is open; extending ASIL to fully opaque applications, which would require fundamentally different access strategies, is left to future work.

Residual perceptual tasks. ASIL exposes structured software state, but some task goals are intrinsically perceptual. The clearest example is the Creative region of the hard suite, where GIMP real-photo editing requires judgments about visual composition that the current semantic action vocabulary cannot fully express – even GPT-5.4 ASIL averages only 14.7 on this region and obtains zero full passes across the 24 tasks. A naive visual supplement does not fix this alone (44.1 mean ASIL-only versus 43.7 with screenshots on 44 no-hint GIMP tasks), suggesting that richer perceptual action primitives are also needed. ASIL is most valuable when success criteria are checkable against structured state; tasks dominated by aesthetic or perceptual criteria need a hybrid interface.

Ethical Considerations

ASIL is an interface and evaluation framework rather than a user-facing autonomous deployment. The benchmark tasks are synthetic software-operation tasks and do not contain personally identifying or offensive content. The main risks are misuse of more capable software-operating agents, overclaiming generality for software without open access paths, and hidden evaluator bias if downstream users treat ASIL scores as universal GUI capability. We mitigate these risks by releasing task definitions, validators, adapter code, training data, and failure summaries; by reporting repaired GUI and native-interface baselines; and by scoping claims to software with open file, scripting, structured-command, or service access paths. Deployments that connect ASIL-style agents to real accounts or files should add permission boundaries, audit logging, confirmation gates for destructive actions, and application-specific safety policies.

Acknowledgments

We thank the anonymous reviewers and area chairs for their constructive feedback. This work was supported by the China NSFC Projects (92370206, 62120106006, 62576212, U23B2057), the Shanghai Municipal Science and Technology Project (25X010202846), and the Frontier Technologies R&D Program of Jiangsu (BF2025029).

References

Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Wang. 2025a. Agent S: An open agentic framework that uses computers like a human. In *International Conference on Learning Representations*.

Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025b. Agent S2: A compositional generalist-specialist framework for computer use agents. *Preprint*, arXiv:2504.00906.

Anthropic. 2026. Claude Code documentation: Overview. https://code.claude.com/docs/en/overview. Accessed: 2026-05-26.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. 2024. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9313–9332, Bangkok, Thailand. Association for Computational Linguistics.

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

jackwener. 2026. OpenCLI. https://github.com/jackwener/opencli. Accessed: 2026-05-26.

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

Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. AXIS: Efficient human-agent-computer interaction with API-first LLM-based agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7711–7743, Vienna, Austria. Association for Computational Linguistics.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. OmniParser for pure vision based GUI agent. *Preprint*, arXiv:2408.00203.

Dang Nguyen, Jian Chen, Yu Wang, Gang Wu, Namyong Park, Zhengmian Hu, Hanjia Lyu, Junda Wu, Ryan Aponte, Yu Xia, Xintong Li, Jing Shi, Hongjie Chen, Viet Dac Lai, Zhouhang Xie, Sungchul Kim, Ruiyi Zhang, Tong Yu, Mehrab Tanjim, and 11 others. 2025a. GUI agents: A survey. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22522–22538, Vienna, Austria. Association for Computational Linguistics.

Dang Nguyen, Viet Dac Lai, Seunghyun Yoon, Ryan A. Rossi, Handong Zhao, Ruiyi Zhang, Puneet Mathur, Nedim Lipka, Yu Wang, Trung Bui, Franck Dernoncourt, and Tianyi Zhou. 2025b. DynaSaur: Large language agents beyond predefined actions. In *Conference on Language Modeling*.

OpenAI. 2026. Codex: OpenAI's coding agent for software development. https://developers.openai.com/api/docs/guides/code-generation#use-codex. Accessed: 2026-05-26.

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

Yueqi Song, Frank F. Xu, Shuyan Zhou, and Graham Neubig. 2025. Beyond browsing: API-based web agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 11066–11085, Vienna, Austria. Association for Computational Linguistics.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, Ruyi An, Molei Qin, Chuqiao Zong, Longtao Zheng, Yujie Wu, Xiaoqiang Chai, Yifei Bi, Tianbao Xie, Pengjie Gu, and 9 others. 2025. Cradle: Empowering foundation agents towards general computer control. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 58658–58725. PMLR.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better LLM agents. *Preprint*, arXiv:2402.01030.

Xinyuan Wang, Bowen Wang, Dunjie Lu, Junlin Yang, Tianbao Xie, Junli Wang, Jiaqi Deng, Xiaole Guo, Yiheng Xu, Chen Henry Wu, Zhennan Shen, Zhuokai Li, Ryan Li, Xiaochuan Li, Junda Chen, Boyuan Zheng, Peihang Li, Fangyu Lei, Ruisheng Cao, and 23 others. 2025. OpenCUA: Open foundations for computer-use agents. *Preprint*, arXiv:2508.09123.

Yuan Wang, Mingyu Li, and Haibo Chen. 2026. From imperative to declarative: Towards LLM-friendly OS interfaces for boosted computer-use agents. *Preprint*, arXiv:2510.04607.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. 2024. OS-Copilot: Towards generalist computer agents with self-improvement. *Preprint*, arXiv:2402.07456.

Rui Xie, Zhi Gao, Chenrui Shi, Zirui Shang, Lu Chen, and Qing Li. 2026. GUIDE: Resolving domain bias in GUI agents through real-time web video retrieval and plug-and-play annotation. *Preprint*, arXiv:2603.26266. Accepted to ECCV 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37. Datasets and Benchmarks Track.

XLANG Lab. 2025. Introducing OSWorld-Verified. https://xlang.ai/blog/osworld-verified. Accessed: 2026-05-26.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. 2023. Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V. *Preprint*, arXiv:2310.11441.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37.

Yuhao Yang, Tianyu Fan, and Chao Huang. 2026. CLI-Anything: Towards agent-native computer use. *arXiv preprint arXiv:2606.03854*.

Chaoyun Zhang, He Huang, Chiming Ni, Jian Mu, Si Qin, Shilin He, Lu Wang, Fangkai Yang, Pu Zhao, Chao Du, Liqun Li, Yu Kang, Zhao Jiang, Suzhen Zheng, Rujia Wang, Jiaxu Qian, Minghua Ma, Jian-Guang Lou, Qingwei Lin, and 2 others. 2025. UFO2: The desktop AgentOS. *Preprint*, arXiv:2504.14603.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2024. UFO: A UI-focused agent for Windows OS interaction. *Preprint*, arXiv:2402.07939.

A Evaluation Details

Main benchmark regions. The main benchmark contains 300 single-application tasks and 80 multi-application tasks. The 300 single-application tasks are grouped into four regions in Table 1: Creative Media contains Audacity, Blender, GIMP, Inkscape, Kdenlive, and OBS; Office & Diagrams contains Draw.io, LibreOffice Calc, LibreOffice Impress, and LibreOffice Writer; Developer Workflows contains code-server, Gitea, and JupyterLab; Files & Communication contains Nautilus and Thunderbird. Each single-application environment contributes 20 tasks. Multi-App is reported separately because these tasks require information or artifacts to move across application boundaries. The default paired budget is 15 interaction steps for both ASIL and GUI. This is a middle-ground setting: ASIL generally needs far fewer steps, whereas GUI-agent evaluations often use 50-step budgets or larger. For complex tasks whose GUI execution would otherwise have too little room for retries, the GUI run is allowed up to 50 steps; these exceptions give GUI extra recovery opportunities. Across reported ASIL runs, the average executed trajectory length remains below five steps.

Hard-task suite. The held-out hard suite contains 80 ASIL-only tasks designed to overweight long-horizon, multi-constraint, deliverable-grade workflows. It contains 24 GIMP real-photo editing and annotation tasks, 30 multi-application workflows, 12 office and diagram tasks, 8 developer tasks, and 6 files and communication tasks. Each task is 6 to 15 meaningful semantic actions long and is checked against final artifacts and visible application state. The suite is held out from the main 380-task benchmark and passes deterministic ASIL execution before agentic evaluation. We do not report formal GUI rows on this suite because repaired GUI / 50max control already remains far below ASIL on the easier main benchmark. We also ran preliminary GPT-5.4 GUI trials on the hard suite and observed the same collapse, so a full GUI sweep would mostly add expensive low-score rows rather than change the interpretation.

Interface-effect case study. Figure 5 grounds the aggregate interface effect of Section 5 in one concrete LibreOffice payroll task from the main benchmark. The figure is retained as a qualitative failure-mode illustration from the submitted paired run: GPT-5.4 GUI control exhausts 15 keypress

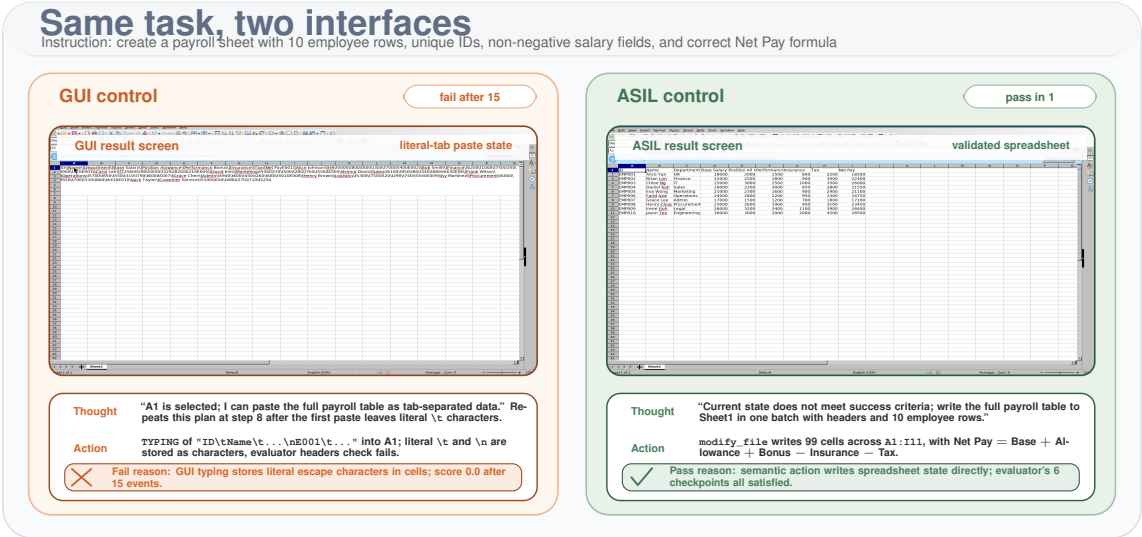

Figure 5: **Same task under GUI control and ASIL control.** GPT-5.4 is run on the libreoffice_19 payroll task in both modes; thoughts and actions are excerpted verbatim from the official GUI and ASIL evaluation runs. The GUI run identifies a literal-tab paste problem but cannot escape the low-level typing loop and fails after 15 events; ASIL writes the spreadsheet state directly with one `modify_file` action and passes.

events in a repeated literal-tab paste loop, whereas GPT-5.4 ASIL uses a single `modify_file` semantic action to satisfy all six evaluator checkpoints.

Failure case studies. Figure 6 visualizes the three residual failure modes named in Section 5.6 using representative tasks from the hard suite: an access-path limit on GIMP real-photo editing, a data-coverage gap on LibreOffice Calc multi-cell edits, and a Qwen3.5-9B SFT regression on Thunderbird that RL later recovers. Each panel pairs a task instruction with a canonical thought/action sketch and the aggregate pass counts observed in the seven-model hard-suite evaluation.

Evaluator design. The evaluator is a shared component used identically across deterministic execution, ASIL agent runs, GUI agent runs, SFT trace filtering, and RL reward computation. Each task defines one or more *success paths*, where each path is an ordered conjunction of typed *checkpoints* (e.g., spreadsheet cell value, file existence, regex match on document text, XPath structure on SVG, image pixel statistics, REST resource state). At evaluation time the evaluator inspects the current software state through the same adapter the agent uses, scores each checkpoint independently with a typed comparator, and returns the maximum path score in $[0, 1]$ as the task's continuous score; binary success requires every required checkpoint along the matched path to pass. Because the same evaluator is invoked for SFT filtering, RL reward, and benchmark scoring, training signal is identical to evaluation signal by construction; Section 5.2 reports an independent raw-state check on a balanced subset.

Realization-pattern audit. Pattern A is file-backed and covers Inkscape, LibreOffice Calc, LibreOffice Writer, LibreOffice Impress, Draw.io, and JupyterLab. Pattern B is native-scripting and covers Blender, GIMP, Kdenlive, and Audacity. Pattern C is service or API based and covers OBS, Gitea, code-server, Thunderbird, and Nautilus. The audit in Table 4 excludes multi-application tasks because they can traverse multiple patterns within one task.

LibreOffice path-restriction study. The default LibreOffice path uses file-backed ODF and spreadsheet edits. The script-dispatch variant routes the same semantic actions through a generated Python script process while reusing the same observation builder, task set, and evaluator. Both variants are evaluated on the same 60 LibreOffice tasks used by the main benchmark.

B ASIL Adapter and Trace Implementation Details

Figure 7 illustrates the three realization pathways as implementations of one shared protocol and shows

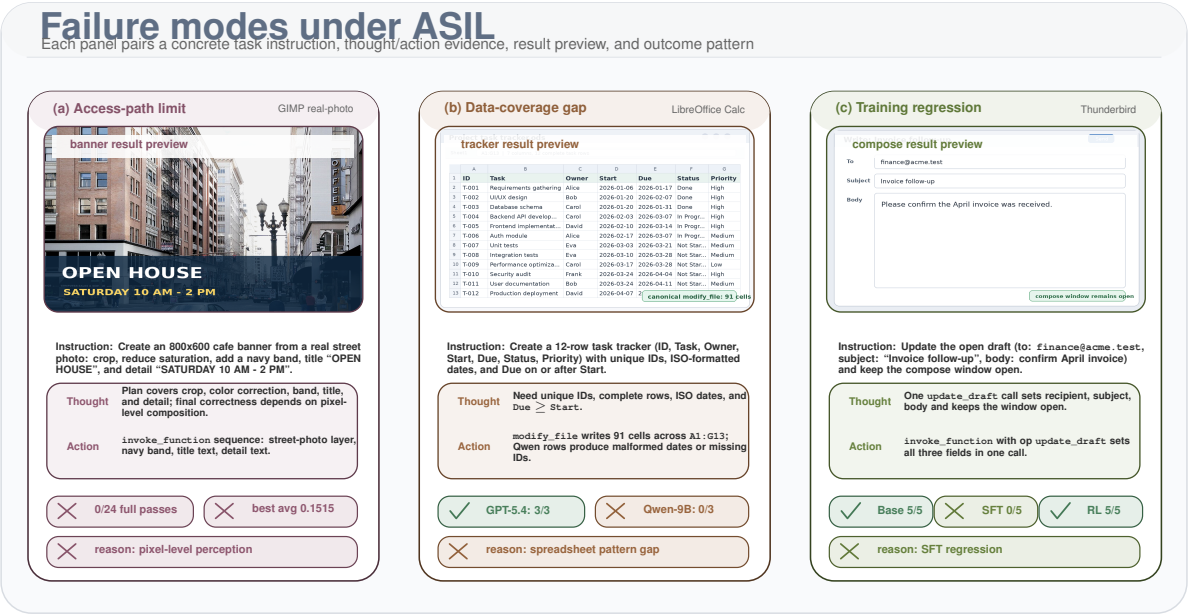

Figure 6: **Representative failure modes under ASIL.** Each panel pairs a real hard-suite task instruction with a canonical thought/action sketch and the aggregate pass counts observed in the seven-model hard-suite evaluation. Residual errors remain after screenshot grounding is removed: real-photo editing exposes access-path limits, spreadsheet tasks expose data-coverage gaps, and Thunderbird shows a supervised-training regression that RL later recovers. Thought/action snippets are abbreviated from the canonical task action specifications; aggregate pass counts come from the hard-suite evaluation report (Section 5.4).

Dimension	ASIL	CLI-Anything	OpenCLI	draw.io MCP	LibreOffice UNO/CLI
Access path	file+script+API	generated CLIs	browser/DOM	draw.io only	headless/UNO
Structured observation	typed JSON	command output	DOM snapshot	app-specific diagram	file/UNO objects
Semantic actions	typed vocabulary	CLI subcommands	browser commands	diagram tools	native API calls
Stable IDs	yes	no	partial	partial	partial
Final-state validator	yes	no	no	no	no
Cross-app evaluation	15 apps	no	no	no	no
Trajectory reuse for SFT/RL	yes	no	no	no	no
Onboarding	semi-auto gen+audit	harness generation	adapter primitives	ready interface	native interface

Table 5: Contract-level comparison with nearby agent-native or programmatic software interfaces. The table compares documented interface dimensions, not task performance; matched task results for LibreOffice UNO and draw.io MCP appear in Table 2.

the normalized trace artifacts they produce.

Adapter contract. Each ASIL adapter implements three required methods: `observe()` extracts the current structured state, `execute(action)` applies one semantic action and returns the next observation, and `validate_action(action)` checks whether an action is valid for the application. The shared adapter base also supports optional hooks for source cloning, task context, real-GUI launch specifications, GUI-to-canonical-state synchronization, and rendering. These hooks allow the same application backend to support ASIL execution, GUI comparison, evaluator replay, and visual inspection without changing the agent-facing protocol.

Observation schema. Each ASIL observation is normalized into a typed JSON object with six main fields: metadata, application state, interactive elements, environment context, navigation structure, and a textual data summary. Metadata records the application and observation source, such as file parsing, native script output, REST state, or DOM snapshots. Application state records the current view, active document, and document path. Interactive elements carry stable IDs, type, label, value, editability, data type, constraints, available actions, child links, and metadata. Environment and navigation fields expose background conditions and reachable views that are usually absent

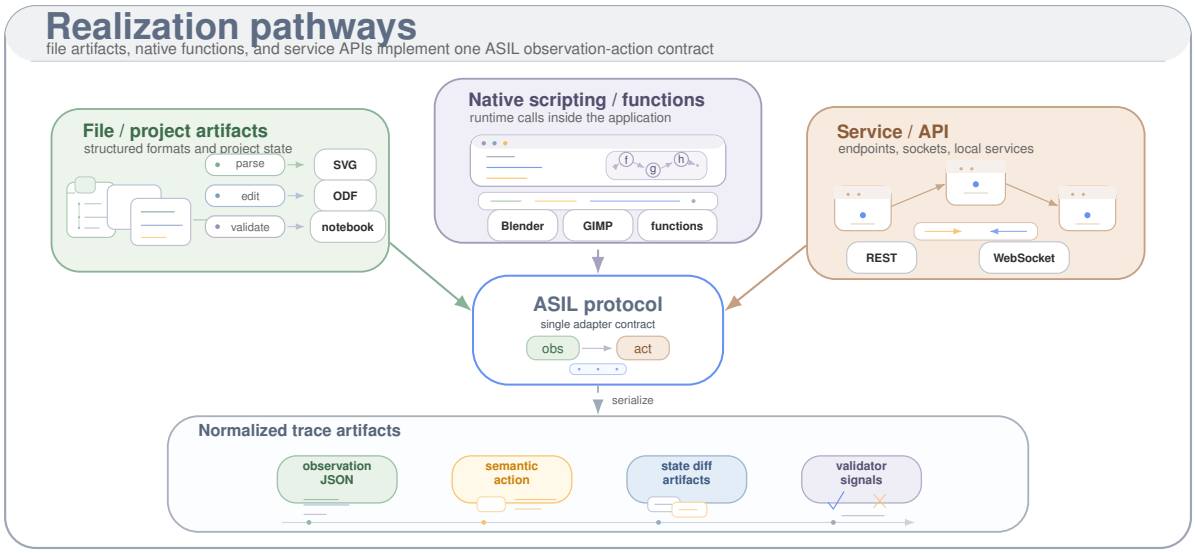

Figure 7: **Realization pathways.** ASIL selects the deepest feasible access path for each software environment while keeping one observation-action protocol; file artifacts, native functions, and service APIs all serialize to the same reusable trace artifacts.

Onboarding stage	Automated
Access-path discovery, profile authoring/review	no
Interface plan (typed observation/action)	yes
Observation schema and action schema	yes
Adapter wrapper and extension bundle	yes
Direct vs. bridge-assisted classification	yes
Static, host, and Docker validation gates	yes
Application-specific bridges	no
GUI synchronization and rendering	no
Task and evaluator design	no

Table 6: Manual versus automated stages of the ASILization pipeline. The automated block is the repeated interface-to-contract work; the manual stages are reviewed setup and evaluation work.

from a screenshot.

Representative adapters. The concrete parsers are application-specific but follow the same normalization rule. The Inkscape adapter parses SVG XML with namespace-aware XPath, exposes shapes, text, images, groups, and layers as elements, and preserves geometry, style, transform, parent, and canvas metadata. Its actions mutate SVG nodes or attributes and then write the file back. The LibreOffice Calc adapter reads `content.xml` inside an ODS archive, exposes cells using stable IDs such as `Sheet1!A1`, records value types and formulas, and executes batched cell edits by updating the underlying ODF XML while extending rows or columns when needed. The Blender adapter generates Python code that runs inside Blender to dump scene objects, transforms, materials, modifiers, animation data, render settings, and timeline settings; actions are realized as generated `bpy` scripts. Service-backed adapters follow the same pattern through APIs: for example, the Gitea adapter exposes repositories, issues, pull requests, milestones, and labels with stable resource IDs and executes REST calls against the corresponding endpoints.

Action schemas and traces. An ASIL action contains an `action_type`, a `target`, and a `params` object, with supported types including `set_value`, `invoke_function`, `modify_file`, `api_call`, `navigate`, and `batch`. Each software environment has a JSON action schema that specifies allowed action types, target format, parameter schema, examples, a `done` action, and software-specific tips. Agentic ASIL runs record a `traj.jsonl` file with per-step observations, thoughts, actions, execution status, latency fields, render metadata, and evaluator scores. GUI runs write the same result structure with screenshot observations and GUI actions. Final scores are written to `result.txt`, while per-step visual artifacts are stored as `step_N.png` and `step_N.render.json`.

C Training Details under the ASIL Modality

C.1 SFT Trajectory Sources

We construct two step-level SFT datasets. SFT-v0 replays the known-correct ASIL actions stored in each task definition: for every task, the pipeline re-executes the annotated correct actions in the ASIL environment and collects per-step (observation, action, post-action observation) tuples, while the teacher model only annotates a short thought before each predetermined action. The supervision target is therefore the (thought, action) pair, but the action itself is not model-generated. The agentic-guided-v2 set instead records verified GPT-5.4 ASIL rollouts in which the model produces its own (thought, action) at each step. For tasks that failed under earlier rollout rounds, the agentic-guided-v2 generator also runs guided recovery using either a compact action hint that names the next operation or a detailed action hint that provides the full action JSON; only evaluator-verified steps with valid action JSON and zero execution errors are written into the dataset. For the final 9B SFT run, v0 and agentic-guided-v2 are merged and exactly deduplicated, producing the merged split summarized at the bottom of Table 7.

Split	Rows	Tasks	Apps	Hints
v0 train	556	512	15	0
v0 valid	138	128	15	0
guided-v2 train	2,330	500	15	148
guided-v2 valid	594	128	15	60
merged train (9B)	2,886	512	15	148
merged valid (9B)	732	128	15	60

Table 7: SFT data construction. All retained rows have valid action JSON, zero execution or annotation errors, and non-empty thought fields; the merged 9B split is the exact deduplication of v0 and agentic-guided-v2.

C.2 SFT Hyperparameters and Selected Checkpoints

Both 2B and 9B runs use FSDP, bfloat16 precision, gradient checkpointing, AdamW with $\beta = (0.9, 0.95)$ and weight decay 0.01, warmup ratio 0.1, gradient clipping at 1.0, and left truncation. Inputs are stored as ASIL chat-format `prompt_messages` and supervised targets are the corresponding (thought + action) assistant response. The final 2B SFT continues from a short v0-pretrained checkpoint for three additional epochs on the agentic-guided-v2 split (111 total steps), and the final 9B SFT trains for six epochs on the merged v0+v2 deduplicated split (1,086 total steps). For both models the selected checkpoint is chosen by downstream ASIL-380 benchmark performance rather than training loss alone, which leads the 9B run to select `global_step_543` (epoch 3 of 6) over the later epochs. Table 8 reports the per-model settings.

Setting	2B SFT	9B SFT
Init model	v0-pretrained cont.	`Qwen3.5-9B`
Data split	guided-v2	merged v0+v2
Train / valid rows	2,330 / 594	2,886 / 732
Global batch	64	16
Micro batch / GPU	1	1
Max seq length	8,192	12,288
Learning rate	5e-6	5e-6
Epochs / total steps	3 / 111	6 / 1,086
Saved ckpts (step)	37, 74, 111	181, . . . , 1086 (6 ckpts)
Selected step	111	543
GPUs (A100)	4	8
380-task score	72.1	80.4

Table 8: Final SFT settings. The selected step is chosen by downstream ASIL-380 benchmark performance; for the 9B run this favors an intermediate epoch over the last epoch.

C.3 RL Curriculum and Rollout Service

For reinforcement learning, we use the learnable v4 curriculum with 320 training tasks and 80 validation tasks, exported as Verl-compatible rows. A training step is one on-policy GRPO-style update over `TRAIN_BATCH_SIZE` task prompts. For every prompt, the trainer requests `ROLLOUT_N` trajectories from a vLLM policy server, which routes ASIL tool calls through an external rollout service. Each rollout job restores the task state, runs an ASIL agent loop for up to 20 action turns inside a managed Singularity ASIL worker, computes a scalar reward from the same evaluator used at benchmark time, and returns the reward together with the full trajectory messages. The trainer, the vLLM policy server, the rollout service, and the Singularity workers run as decoupled processes, and all RL evaluations set `ASIL_DISABLE_RENDER=1` so that the training environment matches the no-render benchmark environment. Both 2B and 9B runs use a reference model for KL regularization, one PPO epoch per update, and an overlength shaping reward with coefficient 0.1 and a buffer of 256 tokens to discourage invalid or excessively long responses.

C.4 RL Hyperparameters and Two-Stage Schedule

The 9B run uses a two-stage schedule. The first stage initializes from the 9B SFT checkpoint `global_step_543`, runs to `global_step_80`, and saves every 40 steps near step 80 and 120. The resume stage loads the full Verl checkpoint at step 80, lowers the actor learning rate from 5e-7 to 3e-7, saves every 20 steps, and produces checkpoints at 100, 120, 140, 160, and 180. Downstream ASIL-380 benchmark performance selects `global_step_140` from the resume stage. The 2B run uses a single 40-step schedule with checkpoints saved every 8 steps; `global_step_8` maximizes the aggregate ASIL score, and later updates do not improve it further, indicating that the small-model policy reaches its useful operating point early. Table 9 reports the per-model RL settings; both models reuse identical KL, entropy, advantage-normalization, and overlength-reward configurations.

Setting	2B RL	9B RL
Init model	2B SFT step 27	9B SFT step 543
Curriculum train / valid	320 / 80	320 / 80
Hardware	4 A800	8 A800
Max ASIL steps / traj.	20	20
`NUM_ENVS`	12	8
`ROLLOUT_N` / `OVER_SAMPLING`	4 / 4	2 / 2
`TRAIN_BATCH`	32	16
`PPO_MINI_BATCH`	4	8
PPO epochs / update	1	1
Actor lr	1e-6	5e-7 → 3e-7
KL / entropy coef	0.001 / 0.0	0.001 / 0.0
Overlength (coef / buf)	0.1 / 256	0.1 / 256
Max prompt / resp.	12,288 / 1,024	12,288 / 1,024
vLLM max batch tokens	8,192	4,096
vLLM GPU mem util.	0.55	0.35
FSDP actor / ref offload	off / on	on / on
Total RL steps	40	80 + 100
Save freq.	every 8	40 / 20
Selected ckpt	step 8	step 140
380-task score	74.4	82.2

Table 9: Final RL settings. The 9B run uses a two-stage schedule with reduced learning rate during the resume stage; the 2B run uses a single short schedule whose best aggregate-score checkpoint occurs early.

D Training Task Generation Overlap Audit

Generation pipeline. The RL curriculum reuses the ASIL full15 software environments and evaluators, but generates new train and validation tasks by rewriting task instructions and literal slots. The generator first builds a template catalog with action skeletons, slot categories, evaluator keys, and non-textual risk fingerprints. It does not expose held-out raw final-test instructions to the model. A task-generation model then proposes structured task specs that change workplace scenarios, target artifact names, labels, paths, layer names, track names, cell names, and other slot values. Deterministic slot replacement turns each spec back into executable ASIL task JSON, after which static audits check parser validity, evaluator consistency, and quota coverage.

Overlap scoring. To reduce overlap with held-out tasks, each candidate is converted into a fingerprint containing software domain, normalized instruction tokens, character n-grams, action-operation signatures, evaluator-rule signatures, literal tokens, target artifacts, asset IDs, and source tags. The overlap score combines lexical, character n-gram, structural, and semantic similarities with weights 0.36, 0.18, 0.28, and 0.18, respectively. Hard rejects cover identical task IDs, identical sources, same target artifacts with high textual overlap, and forbidden held-out GIMP assets. For each candidate, a GPT-5.4 judge inspects the top heuristic-risk pairs and labels them as exact duplicate, near duplicate, same template, same domain but distinct, or unrelated. We reject candidates with hard rejects or maximum risk at least 0.72, warn on risk between 0.55 and 0.72, and accept below 0.55.

Selected curriculum. The final v3 generated pool selects 512 training tasks and 128 validation tasks across all 15 software domains, with strict quotas of 32 train tasks per non-GIMP application and 64 for GIMP, and 8 validation tasks per non-GIMP application and 16 for GIMP. The selected sets contain zero rejected or hard-rejected tasks. The selected v3 train set has 91 strict-accept tasks, a 17.8% strict-accept ratio, and maximum selected overlap risk 0.690; the selected v3 validation set has 19 strict-accept tasks, a 14.8% strict-accept ratio, and maximum selected overlap risk 0.650. We treat this as an engineering-valid low-leakage source pool rather than as a fully de-templateized task generator, because many selected tasks remain in the warning band. The final RL runs use a learnable v4 subset with 320 train tasks and 80 validation tasks.

Table 10(a) reports the candidate-generation and overlap-audit iterations used to build the ASIL RL curriculum, and Table 10(b) summarizes the selected task sets after quota selection and internal duplicate checks.

(a) Per-iteration candidate generation and overlap audit

Run	Split	Cand.	Refs	Accept	Warn	Reject	Acc. %	Non-rej. %	Method change
v1_baseline	combined	16	306	0	0	16	0.0	0.0	handwritten v1 train/valid baseline
train_r1	train	1024	306	65	155	804	6.3	21.5	API slot rewrite, initial prompt
train_r2_heuristic	train	1024	306	978	46	0	95.5	100.0	expanded target-slot extraction, heuristic-only audit
train_r2_calibrated	train	1024	306	60	515	449	5.9	56.2	expanded target slots plus calibrated GPT judge
train_r3_shortfall	train	512	306	26	206	280	5.1	45.3	targeted shortfall generation for low-yield software
train_r4_gimp	train	64	306	0	37	27	0.0	57.8	targeted GIMP repair
train_final_merged	train	1600	306	86	758	756	5.4	52.8	merged v2 final train pool
train_r5	train	1536	306	89	963	484	5.8	68.5	risk-aware R5 train pool
train_r6_merged	train	1600	306	91	982	527	5.7	67.1	R6 targeted train repair
valid_r2	valid	256	818	5	139	112	2.0	56.2	initial valid pool judged against test plus train
valid_r3_shortfall	valid	448	818	8	284	156	1.8	65.2	targeted valid shortfall generation
valid_r4_blender	valid	64	818	1	9	54	1.6	15.6	targeted Blender repair
valid_final_merged	valid	768	818	14	432	322	1.8	58.1	merged v2 final valid pool
valid_r5	valid	384	818	19	272	93	4.9	75.8	risk-aware R5 valid pool

(b) Selected task-set summary after quota selection

Set	Selected	Acc.	Acc. %	Rej.	Max risk
v2_train	512	86	16.8	0	0.700
v3_train_r6	512	91	17.8	0	0.690
v2_valid	128	14	10.9	0	0.700
v3_valid	128	19	14.8	0	0.650

Table 10: Training task generation overlap audit. **(a)** Per-iteration candidate generation and overlap audit: candidates with maximum overlap risk below 0.55 are accept, risk in $[0.55, 0.72)$ is warn, and hard rejects or risk ≥ 0.72 are reject; non-reject rate is accept plus warn. Rows are grouped into baseline, train pools, and validation pools. **(b)** Selected task-set summary after quota selection and internal duplicate checks. All listed sets are quota-complete with zero quota backfill; the v3 pool is further filtered into the learnable v4 subset used for final RL.

E Reproducibility

We release the ASIL inference and evaluation stack to support reproduction of the benchmark setup and extension to new applications. Public resources are separated into the GitHub code repository, released model checkpoints, benchmark tasks, benchmark runtime images, and training data. The release covers: (i) the adapter library for all 15 applications, the shared observation–action protocol, the evaluator implementations described in Appendix B, and the semi-automatic ASILization pipeline (adapter templates, observation–schema scaffolds, evaluator-rule scaffolds, and validation tooling); (ii) task definitions for the 380-task main benchmark, the 80-task hard suite, the easy60 band, and the prepared 320/80 RL curriculum derived from the v3 generation pool of Appendix D; (iii) the prepared SFT-v0, agentic-guided-v2, and merged 9B datasets summarized in Table 7, together with the selected Qwen3.5-2B and Qwen3.5-9B SFT and RL checkpoints used in Tables 1, 3, and 4; and (iv) the repaired GUI, native-UNO, and draw.io-MCP baseline scripts used for the camera-ready audits. A one-command Docker path builds the runtime from source on x86_64 Ubuntu 22.04/24.04, launches the desktop/service dependencies, runs a deterministic no-key smoke test, and emits a fail-closed readiness report over the 15 adapter gates.