WHO SHOULD LEARN FROM FAILURE? PLAYCODE-RSI FOR VERIFIED CO-EVOLUTION OF CODE AND GUI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Code agents can repair interactive software, but some defects only appear during execution. GUI agents provide runtime evidence, yet their failures may reflect their own perception, planning, or control mistakes rather than faulty code. This raises a central question: **who should learn from failure?** We study this question in interactive games, where editable code and repeatable interaction allow us to reproduce similar failures with different causes. Automatically verifiable outcomes then help determine whether the game, the player, or both need to change. We introduce **PlayCode-RSI**, a framework for verified co-evolution of code and GUI agents. Its core mechanism, CARE (Causal Attribution and Routed Evolution), combines visual trajectories, execution feedback, program states, and source code. It selects among five decisions: patch the game, update the player, update both, retry, or make no update. Candidate updates are replayed and must pass source-task, transfer, retention, safety, and cross-partner verification. Only verified code experience and GUI skills are inherited by later generations. We also introduce PlayAdaptBench, with Coding, GUI, and Joint RSI subsets, together with multi-turn vision–action–code training data. Evaluation separates partial progress from strict task completion and tracks both independent and joint capability growth. Experiments show sustained improvements in both agents and transfer to unseen tasks, with fewer incorrect updates and less partner-specific co-adaptation. These findings support failure attribution and verified inheritance as a basis for recursive self-improvement. The code, benchmark, and dataset will be publicly released.

1 INTRODUCTION

Large language models increasingly act as agents that develop software and operate graphical interfaces. Code agents can inspect repositories, locate defects, generate patches, and run tests Jimenez et al. (2024); Yang et al. (2024); Wang et al. (2024), but source code and existing tests may miss failures that only appear during interaction, such as incorrect collision behavior, input timing, or interface responses Lu et al. (2025); Liu et al. (2026); Peng et al. (2026). GUI agents complement this by interpreting screenshots and interacting through mouse and keyboard actions Zhou et al. (2024); Koh et al. (2024); Wang et al. (2026). They can expose runtime failures that are difficult to detect statically, but may also fail because of their own perception, planning, or control errors. These limitations make the two agents naturally complementary. GUI agents provide runtime evidence, while code agents use this evidence together with source code to diagnose and repair the software Lin et al. (2025); Huang et al. (2026a). Code changes then affect later interactions, which in turn provide new feedback. This creates a natural loop for co-evolution: code agents can accumulate reusable diagnosis and repair experience, while GUI agents can improve their interaction strategies. The goal is for both sides to improve across tasks, rather than only adapt to the current software or partner.

Recent work has begun to connect code agents with GUI agents. WebGen-Agent Lu et al. (2025), Coder-CUA Collaboration Lin et al. (2025), VF-Coder Liu et al. (2026), and PlayCoder Peng et al. (2026) use screenshots, real GUI interaction, or automated tests to provide runtime feedback for code generation and debugging. Play2Code Huang et al. (2026a) further introduces a continual play and code refinement loop in games, where a GUI agent tests the current version and a Code Agent updates the game based on interaction feedback. These works show that real execution can connect

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

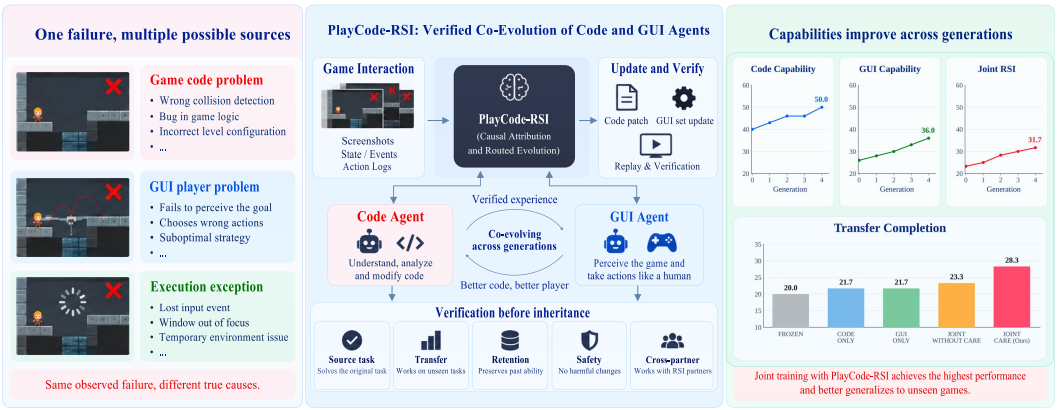

Figure 1: The same observed failure may originate from the game, the GUI agent, or a transient execution issue. PlayCode-RSI attributes the failure, updates the appropriate side, and inherits only verified experience, enabling transferable co-evolution of Code and GUI Agents across generations.

software behavior with code repair in a continuous feedback loop. However, a feedback loop does not guarantee reliable co-evolution. A failed interaction may come from faulty code, a poor GUI strategy, a transient execution error, or even a valid failure. Updating the wrong side can introduce harmful fixes or partner-specific adaptation. This makes the problem closely related to recursive self-improvement (RSI) Robeyns et al. (2025); Sun et al. (2025): useful experience should not only improve the current task, but also be correctly attributed, verified, inherited, and transferable to future tasks. This leads to our central question: **Who should learn from failure?**

To address these limitations, we propose PlayCode-RSI, a framework for verified co-evolution of Code Agents and GUI Agents as shown in Fig. 1. We use interactive games as the testbed because they combine editable program logic with repeatable GUI interaction, provide automatically verifiable outcomes, and allow similar failures to be created from different underlying causes. During interaction, the GUI Agent provides screenshots, action outcomes, states, and traces as runtime evidence, while the Code Agent combines this evidence with source code for diagnosis and repair. The core mechanism, CARE (Causal Attribution and Routed Evolution), decides whether to patch the game, update the player, update both, retry, or make no update. Each candidate update is replayed and must pass source-task, transfer, retention, safety, and cross-partner verification before being committed to long-term memory. PlayCode-RSI also separates task repair from capability improvement. A code fix is useful only if the learned diagnosis or repair experience helps on new errors or games, while a GUI skill should transfer beyond a specific layout or action sequence. We therefore evaluate the Code Agent, GUI Agent, and joint system after each generation using unseen tasks, fixed reference partners, and cross-generation pairings. This allows us to distinguish transferable co-evolution from task-specific or partner-specific adaptation.

To support this study, we introduce PlayAdaptBench, an execution-grounded benchmark with 260 tasks: 100 Coding, 100 GUI, and 60 Joint RSI tasks. All tasks run in browser-based games. Coding tasks evaluate code changes through hidden execution and regression tests, while GUI tasks require agents to complete goals through mouse and keyboard interaction. Joint RSI tasks evaluate the full loop of failure diagnosis, update selection, modification, replay, and verification. We report Process Score for intermediate execution progress and Completion Score for strict task success, enabling separate evaluation of code, GUI, and joint capability. Our main contributions are as follows.

1. We propose PlayCode-RSI, a framework for the continual co-evolution of Code Agents and GUI Agents through real interaction and verified experience inheritance. It evaluates both individual and joint capability growth across generations.

2. We introduce CARE (Causal Attribution and Routed Evolution) to identify failure sources and decide whether to update the code, the GUI Agent, both, or neither. Verified updates must pass source-task, transfer, retention, safety, and cross-partner checks.

3. We build PlayAdaptBench and multi-turn vision–action–code training data for Coding, GUI, and Joint RSI tasks under real execution. Experiments show that verified bidirectional feedback improves both code and GUI capabilities and transfers to unseen tasks.

2 Related Work

2.1 Code and GUI Agents for Interactive Software

Code agents increasingly handle repository-level software engineering tasks through code inspection, editing, and execution feedback. SWE-bench Jimenez et al. (2024), SWE-agent Yang et al. (2024), and OpenHands Wang et al. (2024) show that agents can diagnose issues, modify code, and validate changes in realistic development environments. However, these settings usually assume that a software defect is already known and focus on whether the generated patch is correct.

For interactive software, runtime behavior provides additional evidence that static tests may miss. WebGen-Agent Lu et al. (2025), VF-Coder Liu et al. (2026), and PlayCoder Peng et al. (2026) use screenshots, GUI interaction, or playtesting to expose behavioral failures and guide code repair. In parallel, GUI agents such as WebArena Zhou et al. (2024), VisualWebArena Koh et al. (2024), OSWorld Xie et al. (2024), OSWorld 2.0 Yuan et al. (2026), SaaS-Bench Shi et al. (2026) and WeaveBench Li et al. (2026) study how agents complete tasks through visual observation and mouse or keyboard actions. Game agents such as VPT Baker et al. (2022), Voyager Wang et al. (2023), Cradle Tan et al. (2025), VARP Chen et al. (2024) and CombatVLA Chen et al. (2025a) further develop reusable interaction skills in dynamic environments.

Recent work begins to connect the two sides. Coder–CUA Collaboration Lin et al. (2025) uses a Computer-Use Agent to evaluate generated interfaces, while Play2Code Huang et al. (2026a) builds a continual loop in which a GUI Agent plays the current game and a Code Agent updates it from interaction feedback. These methods demonstrate the value of code and GUI collaboration, but failures are still largely treated as signals for code modification. PlayCode-RSI instead treats failure as an attribution problem and explicitly decides whether the code, the GUI Agent, both, or neither should be updated.

2.2 Agent Self-Improvement and Co-Evolution

Many agent systems improve by storing and reusing past experience. Reflexion Shinn et al. (2023), ExpeL Zhao et al. (2024), and OS-Copilot Wu et al. (2024) use reflections, reusable experience, or learned skills to improve future behavior. SEAgent Sun et al. (2025) further studies self-evolving computer-use agents through exploration and trajectory-based learning. These approaches mainly focus on improving a single agent over time.

Other work studies stronger forms of self-improvement or multi-agent evolution. SICA Robeyns et al. (2025) and Darwin Gödel Machine Zhang et al. (2026) generate and evaluate improved agent variants. CURE Wang et al. (2025), ACE Huang et al. (2026b), Agent0 Xia et al. (2025), and Multi-Agent Evolve Chen et al. (2025b) explore learning through interaction between multiple roles. However, higher joint performance does not necessarily imply that each agent has gained transferable capability. PlayCode-RSI therefore combines failure attribution, routed updates, and verified inheritance, and separately evaluates the Code Agent, GUI Agent, and joint system across unseen tasks and different partners. This allows us to distinguish genuine co-evolution from short-term or partner-specific adaptation.

3 Methodology

This section first presents the PlayCode-RSI framework, covering CARE-based routing, code and GUI updates, and verified inheritance, and then introduces the PlayAdaptBench environment and multi-turn vision–action–code training data.

3.1 PlayCode-RSI

3.1.1 Overview

PlayCode-RSI organizes the GUI Agent, Code Agent, executable game, and long-term memory into a continual co-evolution loop. We first warm-start the base model through supervised fine-tuning on our multi-turn vision–action–code dataset to provide basic combined code and GUI capabilities. The model weights then remain fixed during online evolution, while improvement comes from

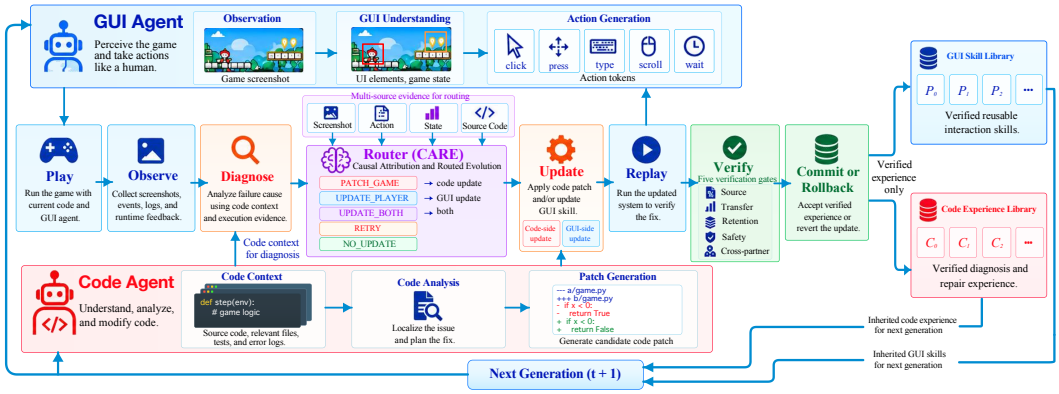

Figure 2: **Overview of PlayCode-RSI.** GUI interaction and code analysis provide evidence for CARE to route each failure to the appropriate update. Candidate code and GUI updates are replayed and verified before reusable experience is committed to long-term memory and inherited by the next generation.

accumulating reusable code experience and GUI skills in the Code Experience Library and GUI Skill Library. These verified memories are carried forward across generations, forming an RSI-style process based on experience accumulation and inheritance rather than online parameter updates.

As shown in Fig. 2, each online evolution round begins with **Play** and **Observe**. The GUI Agent interacts with the current game, while the system records screenshots, actions, public states, and execution feedback. During **Diagnose**, the Code Agent combines this runtime evidence with source code to identify the likely cause of failure. The resulting evidence is then sent to CARE for **Route**. CARE decides whether to patch the game, update the GUI Agent, update both, retry, or make no update. The selected change is applied in **Update** and evaluated through **Replay**. The candidate must then pass source, transfer, retention, safety, and cross-partner checks in **Verify**. Only verified experience is committed to the corresponding library and inherited by generation $t + 1$; otherwise, the system rolls back to the previous state.

3.1.2 CARE: Causal Attribution and Routed Evolution

CARE is the routing mechanism of PlayCode-RSI. Given the interaction history h_t, code context, and runtime evidence, it selects

$$r_t \in \{\texttt{PATCH_GAME}, \texttt{UPDATE_PLAYER}, \texttt{UPDATE_BOTH}, \texttt{RETRY}, \texttt{NO_UPDATE}\}.$$

As shown in Tab. 1, `PATCH_GAME` is selected when the failure is caused by incorrect game logic and triggers a code patch. `UPDATE_PLAYER` is used when the game is correct but the GUI Agent lacks the required perception, planning, or control skill. `UPDATE_BOTH` applies both a code patch and a GUI skill update when problems exist on both sides. `RETRY` handles transient execution failures such as dropped input or focus loss, while `NO_UPDATE` is used for valid failures or cases that require no modification.

CARE uses multiple sources of evidence, including screenshots, action outcomes, public states, and source code. When the cause is ambiguous, the system may collect additional evidence under a fixed probe budget, for example by repeating an action, inspecting related code, or using a reference player. The selected route therefore determines the concrete update applied in the current round. Reusable code experience and GUI skills are added to their respective libraries only after the resulting update passes verification.

Table 1: CARE routes and their corresponding updates.

Route	Update	Typical Cause
`PATCH_GAME`	Code patch	Incorrect program behavior
`UPDATE_PLAYER`	GUI skill	Insufficient player capability
`UPDATE_BOTH`	Code patch + GUI skill	Problems on both sides
`RETRY`	None	Transient execution error
`NO_UPDATE`	None	Valid failure or false alarm

3.1.3 CODE AND GUI UPDATES

The Code Agent and GUI Agent use different forms of long-term memory. On the code side, each update may produce a task-specific patch together with reusable diagnosis and repair experience. The reusable experience records the failure pattern, supporting evidence, affected mechanism, repair strategy, and rollback reason. A task-specific patch is not treated as general capability unless the underlying repair experience transfers to other tasks. On the GUI side, the system stores reusable interaction skills rather than fixed action sequences. Each skill specifies when it applies, what visual target to identify, which actions to perform, and how to recover from failure. For example, adjusting a drag endpoint according to the detected target can transfer across layouts, while clicking a fixed coordinate cannot.

3.1.4 VERIFICATION AND CROSS-PARTNER EVALUATION

A successful source task is not sufficient for committing an update. For a candidate update u_t, PlayCode-RSI requires

$$V(u_t) = V_{\text{src}} \wedge V_{\text{trans}} \wedge V_{\text{ret}} \wedge V_{\text{safe}} \wedge V_{\text{cross}}.$$

V_{src} checks whether the original task is solved. V_{trans} evaluates transfer to unseen tasks or layouts. V_{ret} checks whether previous capabilities are preserved. V_{safe} rejects harmful shortcuts or rule changes. V_{cross} tests whether the improvement remains useful with different partners. If any check fails, the candidate is rolled back and is not inherited. The failed update and its reason may be stored as negative experience to avoid repeating the same mistake.

Cross-partner evaluation is important because higher joint performance can result from partner-specific adaptation. Let G_0 and G_1 denote the game before and after a code update, and P_0 and P_1 denote the corresponding GUI Agent versions. We evaluate

$$(G_0, P_0), \quad (G_1, P_0), \quad (G_0, P_1), \quad (G_1, P_1).$$

These combinations separate the effects of code-side and GUI-side changes. We further evaluate cross-generation pairs $C_i \times P_j$, together with fixed reference partners and unseen tasks. Improvements that persist across different partners provide stronger evidence of transferable co-evolution than gains that appear only within the same generation.

3.2 PLAYADAPTBENCH: AN EXECUTABLE ENVIRONMENT

We build **PlayAdaptBench** as the executable environment for training and evaluating PlayCode-RSI. It contains 260 browser-based tasks covering code repair, GUI interaction, and joint Code–GUI adaptation. All tasks are evaluated through real execution rather than static matching. The GUI Agent interacts only through browser-executable actions, while the Code Agent accesses task-visible source code and public execution evidence. Hidden targets, gold patches, and private evaluation logic remain inaccessible to both agents.

PlayAdaptBench contains three subsets. **Coding** evaluates whether a Code Agent can repair faulty game logic while preserving existing behavior under hidden layouts and regression tests. **GUI** evaluates visual grounding, interaction, planning, and recovery through real mouse and keyboard actions, with success determined from the final environment state rather than a fixed action sequence. **Joint RSI** evaluates the full PlayCode-RSI loop, including failure diagnosis, update routing, code or player modification, replay, and verification. The current Joint RSI split balances the three persistent update routes, `PATCH_GAME`, `UPDATE_PLAYER`, and `UPDATE_BOTH`; `RETRY` and `NO_UPDATE` remain supported CARE decisions but are not separately instantiated as benchmark tasks in the current release candidate.

As summarized in Tab. 2, all three subsets follow a consistent 40% Easy, 40% Medium, and 20% Hard difficulty split. We report both Process Score, which measures checkpoint-level progress, and Completion Score, which requires all task-specific conditions to be satisfied. Overall performance is computed as the macro average across the three subsets so that Joint RSI receives equal weight

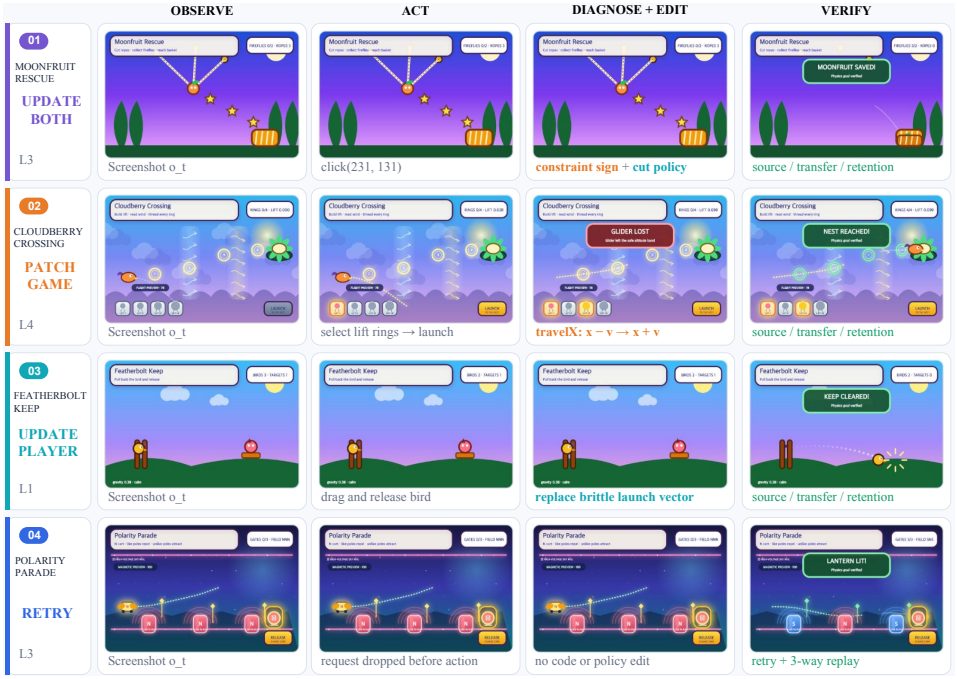

Figure 3: **Representative multi-turn training trajectories.** Each row shows one executable interaction sequence from observation and action to diagnosis, update, and verification. The examples cover different CARE routes, including updating both sides, patching the game, updating the player, and retrying.

despite having fewer tasks. Additional benchmark construction and evaluator details are provided in Appendix.

Table 2: Statistics of the PlayAdaptBench.

Subset	Easy	Medium	Hard	Total
Coding	40	40	20	100
GUI	40	40	20	100
Joint RSI	24	24	12	60
Total	**104**	**104**	**52**	**260**

3.3 Multi-Turn Vision–Action–Code Dataset and Training

We collect multi-turn vision–action–code trajectories from real game execution using teacher models, human correction, and execution-based verification. Each trajectory records visual observations, GUI actions, execution feedback, code context, and verification results. Joint RSI trajectories additionally include failure diagnosis, CARE routing, code-side or GUI-side updates, and the final commit or rollback decision. As shown in Fig. 3, each row illustrates a different failure type and the corresponding CARE decision. In the first example, both the game logic and the player policy are adjusted, leading to an `UPDATE_BOTH` decision. The second example contains a game-side motion error and is handled with `PATCH_GAME`. The third example is caused by an insufficient player strategy, so only the GUI Agent is updated through `UPDATE_PLAYER`. In the fourth example, the failure comes from a dropped action rather than either agent, so the system selects `RETRY` without modifying the code or policy. These cases show that the training data covers distinct failure sources and teaches the model to choose different update routes instead of treating every failure as a code defect.

We use supervised fine-tuning with multi-turn prefix supervision, where the model predicts each next decision from the execution history available at that point. The same interaction protocol is used during training and inference. We train GUI-only, Code-only, GUI + Code, and full GUI +

Table 3: Process Score on PlayAdaptBench.

Model	Coding P	GUI P	Joint RSI P	Overall P
GPT-5.6 Sol	**63.7**	**58.4**	45.1	**55.7**
Claude Opus 4.8	61.2	56.7	42.8	53.6
Kimi-K3	56.9	51.3	38.9	49.0
Qwen3.5-35B-A3B Base	38.6	31.9	23.4	31.3
GUI-only SFT	40.8	42.6	27.1	36.8
Coding-only SFT	50.7	34.1	28.6	37.8
GUI + Coding SFT	53.1	45.2	33.4	43.9
PlayCode-RSI SFT	58.3	50.9	**46.3**	51.8

Table 4: Completion Score on PlayAdaptBench.

Model	Coding C	GUI C	Joint RSI C	Overall C
GPT-5.6 Sol	**47.0**	**29.0**	20.0	**32.0**
Claude Opus 4.8	43.0	27.0	18.3	29.4
Kimi-K3	37.0	23.0	16.7	25.6
Qwen3.5-35B-A3B Base	17.0	9.0	3.3	9.8
GUI-only SFT	20.0	19.0	6.7	15.2
Coding-only SFT	27.0	11.0	6.7	14.9
GUI + Coding SFT	30.0	21.0	11.7	20.9
PlayCode-RSI SFT	40.0	26.0	**23.3**	29.8

Code + Joint RSI variants to measure the contribution of different training data. Detailed trajectory construction and supervision rules are provided in Appendix.

4 Experiments

4.1 Experimental Setup

Models. We use Qwen3.5-35B-A3B as the base model and perform supervised fine-tuning on our multi-turn vision–action–code data. We train four variants using GUI-only, Coding-only, GUI + Coding, and full GUI + Coding + Joint RSI data. We also evaluate GPT-5.6 Sol, Claude Opus 4.8, and Kimi-K3 under the same task inputs, action space, execution interface, and evaluation protocol.

Metrics. We report Process Score (P) and Completion Score (C) on the Coding, GUI, and Joint RSI subsets. Process Score measures verified intermediate progress, while Completion Score counts a task as successful only when all required conditions are satisfied. Completion is computed separately on the 100 Coding, 100 GUI, and 60 Joint RSI tasks. Overall performance is the unweighted macro average across the three subsets:

$$S_{\text{overall}} = \frac{S_{\text{Coding}} + S_{\text{GUI}} + S_{\text{Joint}}}{3}.$$

For online evolution, we additionally report four metrics. **Transfer Completion** measures success on unseen tasks, layouts, or parameters after an update. **Retention Rate** measures how well previously acquired capabilities are preserved. **Wrong-Target Update Rate** measures how often the system updates the wrong component given the actual failure source. **Harmful Update Rate** measures the fraction of candidate updates that introduce regressions or violate safety constraints.

4.2 Main Results

We first evaluate static model capability on PlayAdaptBench. Process Score measures partial progress through executable checkpoints, while Completion Score provides a stricter measure of end-to-end success.

Table 5: Comparison of online evolution strategies.

Setting	Coding C	GUI C	Joint C	Transfer C	Retention	Wrong-Target
FROZEN	40.0	26.0	23.3	20.0	100.0	–
CODE_ONLY	46.0	27.0	25.0	21.7	94.2	13.3
GUI_ONLY	40.0	32.0	25.0	21.7	93.3	11.7
JOINT_WITHOUT_CARE	47.0	32.0	28.3	23.3	83.3	25.0
JOINT_CARE	**50.0**	**36.0**	**31.7**	**28.3**	**95.0**	**8.3**

Table 6: Capability growth across online evolution generations.

Generation	Coding C	GUI C	Joint C
$t = 0$	40.0	26.0	23.3
$t = 1$	43.0	28.0	25.0
$t = 2$	46.0	30.0	28.3
$t = 3$	46.0	33.0	30.0
$t = 4$	**50.0**	**36.0**	**31.7**

As shown in Tab. 3, strong general-purpose models remain competitive on the individual Coding and GUI subsets. GPT-5.6 Sol achieves the highest Coding and GUI Process Scores, while Claude Opus 4.8 and Kimi-K3 follow closely. In contrast, the training variants show clear specialization. GUI-only SFT mainly improves GUI interaction, while Coding-only SFT mainly improves code repair. Combining GUI and Coding data improves both sides, but the gain on Joint RSI remains limited. After adding Joint RSI trajectories, PlayCode-RSI SFT reaches the highest Joint RSI Process Score of 46.3, exceeding GPT-5.6 Sol by 1.2 points. This indicates that explicit supervision on failure attribution, routed updates, replay, and verification provides additional capability beyond simply combining coding and GUI data.

Tab. 4 shows a similar trend under the stricter Completion Score. General-purpose models still perform best on the standalone Coding and GUI subsets, with GPT-5.6 Sol reaching 47.0 and 29.0, respectively. However, their advantage becomes smaller on Joint RSI, where PlayCode-RSI SFT achieves the best Completion Score of 23.3. Compared with GUI + Coding SFT, adding Joint RSI supervision nearly doubles Joint Completion from 11.7 to 23.3, while Coding and GUI Completion also improve from 30.0 to 40.0 and from 21.0 to 26.0. These results suggest that strong coding and GUI abilities alone are not sufficient for joint adaptation. The model must also identify the source of failure, select the correct update target, and verify whether the resulting change should be retained.

Overall, the two tables show that PlayCode-RSI SFT does not simply improve one isolated capability. It strengthens the interaction between code repair and GUI control, leading to substantially better performance on the Joint RSI subset while remaining competitive on the individual tasks.

4.3 Co-Evolution and Long-Term Improvement

We next study whether verified experience produces continued improvement when the model parameters remain fixed.

Joint Evolution vs. One-Sided Updates. All settings start from the same PlayCode-RSI SFT checkpoint. FROZEN does not inherit experience, CODE_ONLY and GUI_ONLY update only one library, JOINT_WITHOUT_CARE updates both sides without causal routing, and JOINT_CARE uses the full framework. As shown in Tab. 5, one-sided evolution mainly improves the corresponding capability. Updating both sides without CARE gives stronger Joint Completion, but also causes more wrong-target updates and a clear drop in retention. The full JOINT_CARE setting improves Coding, GUI, Joint, and transfer performance while preserving previous capabilities, showing that bidirectional updates are most effective when paired with explicit failure attribution.

Capability Growth Across Generations. We evaluate the inherited state after every generation on a fixed held-out set. As shown in Tab. 6, $t = 0$ denotes the warm-started PlayCode-RSI SFT model before online inheritance. All three completion metrics improve across generations. Coding shows a temporary plateau at $t = 3$, while GUI and Joint capability continue to improve. The

Table 7: Illustrative cross-generation Joint Completion.

Code / GUI	P_0	P_1	P_2	P_3	P_4
C_0	23.3	25.0	23.3	26.7	25.0
C_1	25.0	25.0	26.7	26.7	28.3
C_2	25.0	28.3	28.3	26.7	30.0
C_3	26.7	28.3	28.3	30.0	30.0
C_4	28.3	26.7	30.0	31.7	31.7

$t = 4$ results match `JOINT_CARE` in Tab. 5 because both correspond to the final state after four generations of online evolution.

Cross-Generation Compatibility. To distinguish transferable improvement from partner-specific co-adaptation, we pair Code and GUI states from different generations and evaluate each combination on the same Joint RSI tasks as shown in Tab. 7. The gains are not concentrated on the diagonal. Later Code states improve performance with several earlier GUI states, and later GUI states also improve earlier Code states. This suggests that the inherited experience transfers beyond the same-generation partner rather than only producing local co-adaptation.

4.4 CARE Analysis

We study whether causal routing and verification are necessary for reliable experience inheritance. The first three baselines replace CARE with a fixed update rule, while the remaining variants remove individual verification components or bypass verification entirely. The results are as shown in Tab. 8.

Fixed update rules consistently underperform Full CARE, showing that failures cannot be handled reliably by always modifying the same component. Always updating both sides achieves higher Process Score than the other fixed rules, but also produces the highest Harmful Update Rate, indicating that broader updates are not necessarily safer or more effective. The verification ablations further show that different checks protect different aspects of long-term improvement. Removing transfer verification mainly reduces performance on unseen conditions, while removing retention and safety checks causes stronger capability loss and more harmful updates. Committing all experiences without filtering performs worse across completion, transfer, and retention. Full CARE achieves the best overall balance, with the highest Completion, Transfer Completion, and Retention, together with the lowest Harmful Update Rate. These results show that solving the current task is not sufficient; an update should be inherited only when it also transfers, preserves prior capabilities, and avoids harmful changes.

Table 8: Ablation of CARE and its verification components.

Method	**Process**	**Completion**	**Transfer C**	**Retention**	**Harmful Update**
Always Patch Game	39.8	16.7	11.7	80.8	31.7
Always Update Player	38.6	15.0	13.3	83.3	28.3
Always Update Both	43.7	20.0	16.7	78.3	35.0
w/o Transfer Verification	55.2	30.0	21.7	93.3	11.7
w/o Retention & Safety	55.6	28.3	26.7	80.0	23.3
Commit All Experiences	52.7	25.0	23.3	75.0	30.0
Full CARE	**56.1**	**31.7**	**28.3**	**95.0**	**8.3**

5 Conclusion

We introduce PlayCode-RSI, a framework for continual and verifiable co-evolution of Code Agents and GUI Agents through real interaction. Its core mechanism, CARE, attributes each failure and routes the system to patch the game, update the player, update both, retry, or make no update. Candidate updates are inherited only after passing source-task, transfer, retention, safety, and cross-partner verification. We further build PlayAdaptBench and multi-turn vision–action–code training data to evaluate code repair, GUI interaction, and joint self-improvement. Experiments show that verified bidirectional feedback improves both agents across generations, while reducing wrong-target updates, harmful adaptation, and partner-specific co-adaptation. These results highlight failure attribution and verified inheritance as key to reliable recursive self-improvement.

AI USE STATEMENT

In this work, we used generative AI tools for synthetic data generation. Specifically, Kimi-K2.6 was used as a teacher model to generate part of the multi-turn vision–action–code training data. All generated trajectories were subject to execution-based verification and filtering before being included in the dataset. Additionally, GPT-5.6 was used to assist with language polishing, grammatical correction, and improving the readability and organization of the manuscript. All AI-assisted content was reviewed and edited by the authors.

Generative AI tools were not used to fabricate or alter experimental results. The authors verified all reported results and take responsibility for the final content of this work, including text, claims, data, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies code and GUI agents in controlled browser-based game environments. Human experts contributed to data collection and trajectory correction, and all contributors were informed about the purpose and use of the collected data. No sensitive personal data were collected, and no human behavior or personal attributes were analyzed. All software modifications are executed in isolated environments with restricted interfaces, private evaluators, rollback mechanisms, and safety checks. We will document data, code, and asset licenses before release and will not expose private evaluation logic or hidden labels.

REPRODUCIBILITY STATEMENT

We provide the full PlayCode-RSI methodology, PlayAdaptBench construction, evaluation metrics, and main experimental settings in the paper. Additional details on benchmark composition, training configuration, trajectory construction, online evaluation, and evaluator checks are included in the Appendix. Code, benchmark tasks, and training data will be released to support reproduction of the reported results.

REFERENCES

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. In *Advances in Neural Information Processing Systems*, 2022.

Peng Chen, Pi Bu, Jun Song, Yuan Gao, and Bo Zheng. Can vlms play action role-playing games? take black myth wukong as a study case. *arXiv preprint arXiv:2409.12889*, 2024.

Peng Chen, Pi Bu, Yingyao Wang, Xinyi Wang, Ziming Wang, Jie Guo, Yingxiu Zhao, Qi Zhu, Jun Song, Siran Yang, Jiamang Wang, and Bo Zheng. Combatvla: An efficient vision-language-action model for combat tasks in 3d action role-playing games. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 10919–10928, 2025a.

Yixing Chen, Yiding Wang, Siqi Zhu, Haofei Yu, Tao Feng, Muhan Zhan, Mostofa Patwary, and Jiaxuan You. Multi-agent evolve: Llm self-improve through co-evolution, 2025b.

Yixu Huang, Bo Li, Na Li, Zhe Wang, Kaijie Chen, Haonan Ge, Qingyi Si, Yuanzhe Shen, Ruihan Yang, Guangjing Wang, and Hongcheng Guo. Gui agents for continual game generation, 2026a.

Yixu Huang, Xinglei Yu, and Zhongyu Wei. Ace: Self-evolving llm coding framework via adversarial unit test generation and preference optimization, 2026b.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 881–905, 2024.

Wanli Li, Bowen Zhou, Yunyao Yu, Zhou Xu, Yifan Yang, Dongsheng Li, and Caihua Shan. Weavebench: A long-horizon, real-world benchmark for computer-use agents with hybrid interfaces. *arXiv preprint arXiv:2606.09426*, 2026.

Kevin Qinghong Lin, Siyuan Hu, Linjie Li, Zhengyuan Yang, Lijuan Wang, Philip Torr, and Mike Zheng Shou. Computer-use agents as judges for generative user interface, 2025.

Zhilin Liu, Ye Huang, Ting Xie, Ruizhi Zhang, Wen Li, and Lixin Duan. Coding with eyes: Visual feedback unlocks reliable gui code generating and debugging, 2026.

Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Junting Pan, Mingjie Zhan, and Hongsheng Li. Webgen-agent: Enhancing interactive website generation with multi-level feedback and step-level reinforcement learning, 2025.

Zhiyuan Peng, Wei Tao, Xin Yin, Chenhao Ying, Yuan Luo, and Yiwen Guo. Playcoder: Making llm-generated gui code playable. In *Proceedings of the 34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2026.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent, 2025.

Kean Shi, Zihang Li, Tianyi Ma, Zengji Tu, Jialong Wu, Xinbo Xu, Qingyao Yang, Ruoyu Wu, Weichu Xie, Ming Wu, et al. Saas-bench: Can computer-use agents leverage real-world saas to solve professional workflows? *arXiv preprint arXiv:2605.15777*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. Seagent: Self-evolving computer use agent with autonomous learning from experience, 2025.

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, Xiyun Li, Ceyao Zhang, Long Tian, Chaojie Wang, Xinrun Wang, Börje F. Karlsson, Bo An, Shuicheng Yan, and Zongqing Lu. Cradle: Empowering foundation agents towards general computer control. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2024.

Xinyi Wang, Wei Dai, Kyle Qiao, Ke Wang, Peng Chen, Gang Cao, Zhongpu Wang, Xiaode Zhang, Yanming Liu, Jihao Gu, et al. Gui0: Self-evolving foundational gui agents in super app ecosystems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 44348–44361, 2026.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems*, 2025.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024.

Peng Xia, Kaide Zeng, Jiaqi Liu, Can Qin, Fang Wu, Yiyang Zhou, Caiming Xiong, and Huaxiu Yao. Agent0: Unleashing self-evolving agents from zero data via tool-integrated reasoning. *arXiv preprint arXiv:2511.16043*, 2025.

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. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024.

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

Mengqi Yuan, Zilong Zhou, Xinzhuang Xiong, Weiming Wu, Jiayang Sun, Jiamin Song, Kaiqian Cui, Bowen Wang, Haoyuan Wu, Yitong Li, et al. Osworld2. 0: Benchmarking computer use agents on long-horizon real-world tasks. *arXiv preprint arXiv:2606.29537*, 2026.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, volume 2026, pp. 104223–104294, 2026.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024.

A APPENDIX

A.1 OVERVIEW

This appendix provides additional details on the benchmark, auxiliary game environments, training procedure, evaluation protocol, and implementation considerations. It is organized as follows:

- **PlayAdaptBench** (Section A.2): benchmark composition, task subsets, and agent interfaces.
- **Physics Games** (Section A.3): objectives, controls, physical rules, and difficulty progression of the auxiliary environments.
- **Training Data and Experience Inheritance** (Section A.4): trajectory construction, supervised fine-tuning, and online memory inheritance.
- **Evaluation Protocol and Evaluator Checks** (Section A.5): scoring, online evolution metrics, cross-generation evaluation, and evaluator validation.
- **Discussion and Limitations** (Section A.6): interpretation of capability growth, scope, limitations, and safety.

A.2 PLAYADAPTBENCH

PlayAdaptBench contains 260 executable tasks: 100 Coding tasks, 100 GUI tasks, and 60 Joint RSI tasks. Each subset follows the same difficulty distribution of 40% Easy, 40% Medium, and 20% Hard.

The shared execution interface supports environment reset, screenshot capture, GUI actions, public event logging, and state-hash recording. The GUI Agent interacts only through browser-executable actions, while the Code Agent accesses task-visible source code, public tests, and execution traces. Hidden layouts, gold patches, and private evaluation logic are inaccessible to both agents. Submitted patches are applied to the executable environment and evaluated through replay rather than static matching.

Coding. The Coding subset contains 100 tasks derived from 14 physical source families and 28 game scenarios. It covers 54 mutation mechanism classes instantiated through 100 mutation templates. The tasks include 24 frozen handwritten physics tasks, 36 handwritten multi-state tasks, and 40 synthetic microcontroller tasks. Each task contains a public case, five hidden target layouts, and five sibling regression layouts.

Table 9 gives the weighted components of Coding Process Score. Strict Completion requires successful recovery of the public task, all hidden target layouts, and all sibling regression layouts, while invalid or unauthorized controls must remain rejected.

Table 9: Weighted components of Coding Process Score.

Checkpoint	Maximum Points
Parse	5
Patch application	5
Patched runtime loading	5
Public recovery	25
Hidden-layout recovery	30
Sibling regression	20
Invalid-control safety	10
Total	**100**

This design prevents trivial solutions that only repair the visible case or bypass the intended game mechanism.

GUI. The GUI subset contains 100 tasks from 20 game families, with five structural or layout instances per family. Tasks cover visual grounding, physical reasoning, planning, recovery, and

generalization across layouts. The supported GUI actions are

$$\{\texttt{click}, \texttt{drag}, \texttt{press}, \texttt{type}, \texttt{scroll}, \texttt{wait}\}.$$

The agent cannot directly access hidden game states or invoke game-specific semantic commands. The evaluator checks the final environment state rather than matching a fixed reference action sequence, allowing multiple valid interaction strategies.

Table 10: Environment categories in the GUI subset.

Environment Type	Tasks	Families / Games
Tutorial-stage games	35	7
Tutorial state-dynamics	5	1
Goal-state dynamics	40	8
Real-physics games	20	4
Total	**100**	**20**

During interaction, the three most recent screenshots are retained in context, while older observations are summarized using public textual information. Screenshots generated after an action are available only at the next decision step, preventing future information leakage.

Joint RSI. The Joint RSI subset contains 60 tasks from 20 environment lineages, with three RSI rounds per lineage. It balances the three persistent update routes: `PATCH_GAME`, `UPDATE_PLAYER`, and `UPDATE_BOTH`, with 20 tasks each.

Table 11: CARE route distribution in the Joint RSI subset.

CARE Route	Tasks
`PATCH_GAME`	20
`UPDATE_PLAYER`	20
`UPDATE_BOTH`	20
Total	**60**

CARE also supports `RETRY` and `NO_UPDATE`, but these are protective decisions rather than standalone benchmark task categories.

Each Joint RSI task requires failure diagnosis, route selection, generation of a code patch, GUI update, or both, followed by replay and verification. Candidate updates are evaluated under source-task, transfer, retention, safety, and negative-control conditions before being accepted.

A.3 Physics Games

We construct a physics-focused auxiliary set based on six interactive games for the benchmark. It contains 90 tasks, including 30 GUI tasks, 30 Coding tasks, and 30 Joint RSI tasks. Each game provides five structural difficulty levels, five mutation settings, and five route conditions for the corresponding task types.

Featherbolt Keep. The player uses a limited flock of birds to defeat every ember imp in a wooden keep. Dragging a bird backward from the slingshot controls its launch direction and speed. Gravity, wind, and collisions determine the trajectory, while strong impacts can break wooden supports. Enemies may be defeated directly or by collapsing their supports. Higher levels introduce crosswinds, armor, different target heights, interacting structures, shorter trajectory previews, and multi-shot plans.

Moonfruit Rescue. The player cuts ropes to collect fireflies and guide a moonfruit into a basket. Intact ropes constrain the fruit, while cutting a rope preserves its current momentum. Both rope selection and timing therefore affect the trajectory. Higher levels add more ropes, moving fireflies, stronger or reversing winds, narrower baskets, and longer timing-dependent sequences.



Figure 4: **Overview of the six auxiliary physics games.** Each game contains five difficulty levels with progressively more challenging interaction and physical dynamics.

Beetle Bridgeworks. The player places anchor tokens on bridge joints before releasing a beetle cart. The cart loads a spring-connected bridge whose deformation propagates across neighboring spans. Anchors stabilize nearby joints, but poor placement can still cause excessive sag or stretching. Higher levels introduce heavier carts, longer spans, fewer anchors, narrower support regions, and tighter placement tolerances.

Pearlshot Grotto. The player launches a limited number of luminous pearls to wake sleeping coral. Drag direction and length determine the initial velocity. Pearls may hit targets directly or rebound from walls and curved bumpers, with each collision reducing energy. Higher levels introduce blocked sight lines, multiple corals, curved bumpers, reduced previews, and multi-bank shots.

Cloudberry Crossing. The player configures balloon pods and launches a glider through required sky rings toward a landing nest. Active balloons provide lift, while gravity and local gusts influence the trajectory. Momentum persists across the flight, requiring the player to account for later rings and landing conditions. Higher levels add more balloon choices, alternating gusts, narrower rings, shorter previews, and tighter landing corridors.

Polarity Parade. The player switches magnetic towers between north and south before releasing a charged cart through crystal gates. Matching poles repel and opposite poles attract, while inertia carries earlier effects into later stages. Higher levels add more towers, stronger forces, alternating gates, narrower lanes, shorter previews, and more complex polarity configurations.

A.4 Training Data and Experience Inheritance

Trajectory construction. Our multi-turn vision–action–code dataset is collected from teacher-model rollouts, human corrections, and execution-verified recovery trajectories. Each rollout records visual observations, GUI actions, execution feedback, public events, state hashes, code context, code versions, and verification outcomes. Joint RSI trajectories additionally include failure attribution, CARE routing, candidate code or GUI updates, replay results, and the final commit or rollback decision. The dataset contains successful trajectories, recovery trajectories after failures, rejected updates followed by rollback, and control cases where no persistent update is required. These examples allow the model to learn not only how to act and repair, but also when an observed failure should not produce a long-term update.

Prefix supervision. For a rollout with T decisions, we construct T prefix samples. The i-th sample contains only the interaction history available before decision i, and the training loss is applied to the current assistant response. Earlier assistant responses remain in context but are masked from the loss. Visual observations follow the actual execution order. Screenshots and feedback produced after an action are available only in the following turn, preventing future information from leaking into earlier decisions. We retain the three most recent screenshots and summarize older observations using public textual information.

Supervised fine-tuning. We use Qwen3.5-35B-A3B as the backbone and perform supervised fine-tuning to warm-start the basic code and GUI capabilities required for subsequent online evolution. We train GUI-only, Coding-only, GUI + Coding, and GUI + Coding + Joint RSI variants using the same optimization recipe. Training and inference follow the same interaction protocol. Code outputs are restricted to the allowed edit scope, while GUI outputs use the browser actions defined in Section A.2. Table 12 summarizes the training configuration.

Table 12: Supervised fine-tuning configuration.

Configuration	Value
Backbone	Qwen3.5-35B-A3B
Fine-tuning method	LoRA
LoRA rank	64
LoRA alpha	128
LoRA dropout	0.05
Vision encoder	Frozen
Maximum context length	32,768 tokens
Maximum screenshots / sample	3
Precision	BF16
Optimizer	AdamW
Learning rate	1.0×10^{-5}
Weight decay	0.1
Warmup ratio	0.03
Learning-rate schedule	Cosine decay
Epochs	2
Effective batch size	64
Gradient clipping	1.0
Gradient checkpointing	Enabled
Training hardware	8× NVIDIA H800 GPUs
Random seed	42

Online experience inheritance. After supervised fine-tuning, model weights remain fixed during online evolution. Improvement is realized through two forms of long-term memory. The Code Experience Library stores reusable diagnosis and repair experience, while the GUI Skill Library stores reusable state-conditioned interaction skills. A task-specific patch is not itself treated as general capability improvement. Its underlying diagnosis and repair experience must remain useful on other tasks or conditions. Similarly, a GUI skill should transfer beyond a specific layout or fixed action sequence. Candidate experience is inherited only after source-task, transfer, retention, safety, and

cross-partner verification. Rejected updates are rolled back, while their failure reasons may be stored as negative experience to avoid repeating ineffective updates.

A.5 Evaluation Protocol and Evaluator Checks

Scoring. For subset s, let N_s denote the number of tasks, $p_{s,j} \in [0, 100]$ the Process Score of task j, and $y_{s,j} \in \{0, 1\}$ its strict completion indicator. We compute

$$\text{Process}_s = \frac{1}{N_s} \sum_{j=1}^{N_s} p_{s,j},$$

and

$$\text{Completion}_s = \frac{100}{N_s} \sum_{j=1}^{N_s} y_{s,j}.$$

Here, $N_{\text{Coding}} = 100$, $N_{\text{GUI}} = 100$, and $N_{\text{Joint}} = 60$.

Each task retains its evaluator-specific scoring rules. In particular, Coding uses the weighted checkpoints in Table 9. Completion is recorded only when all required conditions are satisfied.

Process and Completion are aggregated across the three subsets using

$$S_{\text{overall}} = \frac{S_{\text{Coding}} + S_{\text{GUI}} + S_{\text{Joint}}}{3}, \qquad S \in \{\text{Process}, \text{Completion}\}.$$

The overall score is therefore a macro average rather than pooled accuracy over all 260 tasks.

Online evaluation. All online settings start from the same PlayCode-RSI SFT checkpoint. After each generation, the inherited Code and GUI states are frozen and evaluated on the same held-out tasks. Cross-generation pairs (C_i, P_j) further test whether improvements remain effective with partners from different generations.

We report four additional metrics for online evolution. **Transfer Completion** is the success rate on held-out transfer cases after an update. **Retention Rate** is the fraction of previously passed tasks that remain successful after the update. **Wrong-Target Update Rate** is the fraction of candidate update decisions applied to an incorrect component given the true failure source. **Harmful Update Rate** is the fraction of candidate updates that introduce regressions or violate safety constraints before rollback. The evaluation pools and counting rules are fixed across methods, while each metric uses its corresponding eligible cases or candidate updates as the denominator rather than the total benchmark size.

GUI evaluator checks. The GUI evaluator is checked using 100 oracle playthroughs, 100 negative controls, and 60 successful alternative-policy trajectories. The alternative policies cover all 40 Medium and 20 Hard tasks. These checks verify that valid interaction strategies are accepted without requiring a single reference action sequence.

Joint RSI evaluator checks. The Joint RSI evaluator is validated using 60 gold-worker runs, 60 no-op worker runs, and 60 wrong-update worker runs. Gold workers are expected to pass, while no-op and wrong-update workers are expected to fail.

We also perform 720 fixture-level checks, summarized in Table 13. These checks validate expected acceptance and rejection behavior for predefined cases. They should not be interpreted as model performance or as guarantees against every possible shortcut.

Table 13: Fixture coverage for the Joint RSI evaluator. Counts describe scripted checks rather than model performance.

Check Type	Count
Positive fixtures	60
Adversarial fixtures	300
Checkpoint-knockout checks	360
Total	**720**

A.6 Discussion and Limitations

PlayCode-RSI follows an RSI-style process in which verified experience from earlier rounds supports later tasks. Online evolution changes reusable memory and skills rather than model parameters, while supervised fine-tuning is used only for the initial warm start. We therefore evaluate offline SFT gains and online experience gains separately.

Improved performance on a repaired task alone does not necessarily indicate capability growth. The game may have become easier, the GUI Agent may memorize a layout, or the Code and GUI Agents may adapt only to each other. We therefore evaluate Code and GUI capabilities separately and use transfer tests, retention tests, and cross-generation pairings to examine whether improvements extend beyond the current task and partner.

Games provide a controlled setting because they combine editable implementations, repeatable interaction, and explicit success conditions. Similar ambiguity can arise in web or desktop software, where failures may result from program logic, interaction strategy, or transient execution errors. Extending PlayCode-RSI to these domains would require suitable execution interfaces and independent verification protocols.

The current benchmark focuses on browser-based environments with restricted actions and code edits. It does not capture the full complexity of large repositories, open-ended desktop interaction, or real-time 3D applications. CARE relies on controlled probes and execution comparisons rather than general causal discovery. The Joint RSI subset also contains only the three persistent update routes, so the benchmark does not provide a balanced evaluation of all five CARE decisions.

The current experiments cover a limited number of online generations. Longer evolution may introduce stale experience, conflicting memories, or accumulated errors. Automatically modifying executable software may also introduce unintended behavior or unsafe shortcuts. Restricted workspaces, private evaluation logic, rollback, retention checks, and safety verification reduce these risks but cannot eliminate them. Future work can extend the framework to broader software environments, longer evolution sequences, and verified parameter updates based on execution-grounded trajectories.