DUOAGENT: SELF-IMPROVING ON SOFTWARE ENGINEERING FROM VISUAL AND STRUCTURAL FEEDBACK

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents are increasingly used for real-world software engineering (SWE), yet they still struggle to resolve challenging practical software issues involving visual behavior, such as broken layouts or incorrect UI states, due to weak visual grounding, scarce training data, and inadequate supervision. To address these challenges, we propose a self-improving agentic framework based on visual and structure feedback to address multimodal SWE problems, where we ground agents in interactive browser environments, exploited during deployment, data generation, and training. In particular, we first propose DuoAgent, a *multi-agent system* with specialized agents: orchestrator, reproducer, solver, and verifier. The reproducer builds a browser sandbox and a fixed action sequence that triggers the reported defect. The solver analyzes the repository and proposes candidate fixes. For each candidate, the verifier rebuilds the software, replays the same actions, compares the resulting trace with the buggy one, and returns a visual and a structural diagnosis. This allows the solver to reason about the effect of its changes on the reported behavior. Then we propose a *synthetic task generation* algorithm that localizes visual elements and injects perturbations into clean sandboxes with various strategies for data augmentation. We scale up the training data from 100 seed instances to 513 synthetic instances, each verified by re-executing the seed's action sequence on the mutated and fixed programs and checking that the injected defect is visible, stable, and removed by the inverse mutation. In addition, we design a novel post-training algorithm based on on-policy self-distillation, DuoOPSD, which trains a student model on its own rollouts with teachers conditioned on diagnoses of visual and structural discrepancies as privileged information. On SWE-bench Multimodal, DuoAgent establishes a new state of the art at 52.0%, substantially surpassing the previous best result of 36.0%. Meanwhile, training with DuoOPSD on our synthetic tasks significantly improves student models and consistently outperforms existing post-training algorithms trained on the same tasks with the same number of optimizer steps. These results establish a new paradigm for self-improving multimodal SWE agents.

1 INTRODUCTION

Large language model (LLM) agents are widely used to resolve real-world software issues (Jimenez et al., 2024; Yang et al., 2024a): given an issue report, an agent navigates the repository, edits the code, runs tests, and submits a patch, and production systems such as Claude Code and Codex follow the same recipe. A large fraction of issues in user-facing software, however, are *visual*: reporters attach screenshots of a misrendered component, a tooltip at the wrong position, or a table that drops a cell, and correctness is defined by what appears on screen. SWE-bench Multimodal (Yang et al., 2024b) benchmarks this setting with issues from popular JavaScript libraries, each with reporter visuals and hidden tests. Resolve rates remain low even for frontier agents and far lower for open-weight models. The main reason is that the standard workflow of existing SWE agents runs unit tests to verify their patches. However, these multimodal issues are invisible to unit tests: the failure is in what is rendered, so the suite stays green. Also, many of them (45% of the test split) are *dynamic*: the reported state is reached only after a sequence of interactions, so the defect lives in a trajectory of interface states rather than in a static one. Resolving them requires driving the running application through the reporter's steps and looking at the result, which existing agents cannot do.

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.

Despite progress in recent work (Huang et al., 2025; Liu et al., 2026), they still fall short in three respects. (1) **No or little reproduction or verification.** Coding agents receive feedback only from existing tests, and visual defects rarely come with a failing one; agents localize the bug by reading code and submit fixes they have never seen rendered. (2) **No training data.** SWE-bench Multimodal has a few hundred instances and a development split of 100, and synthetic pipelines for coding agents (Yang et al., 2026; Pan et al., 2024) verify injected bugs with tests, exactly the signal visual defects lack. (3) **No effective training signal for small models.** Imitating a frontier agent transfers behaviors a small model cannot afford, as a 200k-context expert's whole-file reading style exhausts a 64k student's context in half the turns; outcome-reward RL assigns one binary reward to about a hundred tool calls and cannot tell which decision was right. On-policy distillation supervises every token of the student's own trajectories instead (Agarwal et al., 2024; Zhao et al., 2026a), but only helps if the teacher's privileged information is one the student can use: we find that a 4B student cannot turn raw execution evidence into a repair target, yet resolves about a quarter more instances when an expert diagnosis of that evidence is in its context.

To address these challenges, we propose a self-improving agentic framework that lets an agent see a visual bug the way its reporter did and improve itself from what it sees, in the spirit of recursive self-improvement (Good, 1966; Schmidhuber, 2003). From the issue and its screenshots, it reverse-engineers an interactive browser environment and the action sequence that drives it into the reported state, so that every candidate patch can be executed and inspected on screen. These executions let the agents that build and inspect the environment improve their *skills*, and the agent that repairs the code improve its *model* (Figure 1). It contains three components. (1) **DuoAgent, a self-improving multi-agent multimodal debugging harness.** An *orchestrator* coordinates the loop and tracks all candidates; a *reproducer* builds the sandbox and the action sequence once per instance and certifies that the resulting trace exhibits the defect; a *solver* proposes candidate patches; and a *verifier* replays the actions on each candidate, compares the new trace with the buggy one step by step, and returns a visual and a structural diagnosis for the solver's next step. The reproducer and the verifier update their skills from the outcomes of their own runs on the development split. (2) **Verified synthetic task generation.** Injected defects in an instance with a browser environment become observable: a strong model mutates the code behind the interactive elements under multiple strategies, replaying the same actions on the mutated and the fixed program yields aligned buggy and correct traces, and corresponding expert visual and structural diagnoses. A difficulty filter keeps the tasks that a strong agent solves, including a hard subset that the student fails. (3) **DuoOPSD, multi-teacher on-policy self-distillation.** The solver improves its own weights without a stronger teacher: we train it on its own rollouts with two teachers that are the solver itself, sharing its EMA weights and differing only in one appended *diagnosis*, of the visible or of the structural discrepancy, written from the aligned traces; the student matches both where it reads the verifier's diagnoses.

On the test split of SWE-bench Multimodal, we find that (1) DuoAgent improves every model over its native harness, raising Claude Fable 5.1 from 46.8% under Claude Code to 52.0%, a new state of the art; (2) DuoOPSD improves the untrained 4B student from 20.3% to 24.2%, outperforming the existing post-training baselines; (3) the correct execution trace does not help the student at inference time while diagnoses written from it do, which explains why diagnosis-conditioned teachers succeed where raw-evidence teachers fail; and (4) both teachers and the synthetic tasks are necessary.

Our technical **contributions** are summarized below:

- We introduce DuoAgent, the first **self-improving multimodal SWE harness** that reproduces a reported issue in an interactive browser environment and verifies every candidate patch in it, providing execution-grounded visual and structural feedback from which its reproducer and verifier also improve their skills.
- We develop the first **verified synthetic task generation algorithm** for visual issues, which injects defects into reproducible instances and validates their behavioral effects through browser execution. This produces diverse training tasks with aligned buggy and correct executions.
- We propose DuoOPSD, a **multi-teacher on-policy self-distillation algorithm** through which the solver improves itself: teachers that are the solver conditioned on complementary visual and structural expert diagnoses supervise its own rollouts. DuoOPSD achieves better performance than competing post-training baselines.

- We establish a **new state of the art on the test split of SWE-bench Multimodal**. DuoAgent improves every evaluated model over its native harness, raising Claude Fable 5.1 from 46.8% under Claude Code to 52.0%.

2 Related Work

Multimodal software engineering agents. LLM agents resolve repository-level issues by navigating the code, editing files, and running tests (Jimenez et al., 2024; Yang et al., 2024a; Wang et al., 2025; Xia et al., 2024). For the visual issues of SWE-bench Multimodal (Yang et al., 2024b), existing systems fall into two categories: (1) coding agents extended with visual inputs, such as SWE-agent Multimodal (Yang et al., 2024b), Agentless (Xia et al., 2024), and OpenHands-Versa (Soni et al., 2026), which read the reporter's screenshots and may browse the web; and (2) pipelines specialized for visual repair, such as GUIRepair (Huang et al., 2025), which renders the issue scenario with generated code to check candidate patches, and GALA (Liu et al., 2026), which aligns screenshots with code for bug localization. Front-end generation agents (Si et al., 2025; Lu et al., 2026; Sansford et al., 2026) and web agents (Zhou et al., 2024; Koh et al., 2024; Xie et al., 2024) also use rendered pages. However, these approaches inspect a static rendering or act in a browser for a task stated in natural language; none reproduces a reported defect as an executable action sequence and verifies each candidate against it step by step. In contrast, DuoAgent drives the running application through the reporter's interactions and compares the traces.

Agent Post-training. Small coding agents are commonly trained with (1) supervised fine-tuning on trajectories of stronger agents over mined or synthetic tasks (Pan et al., 2024; Yang et al., 2026; Jain et al., 2025), or (2) reinforcement learning with outcome rewards (Shao et al., 2024; Wei et al., 2026). On-policy distillation instead lets a teacher score every token of the student's own trajectories (Agarwal et al., 2024; Gu et al., 2024; Lu & Thinking Machines Lab, 2025; Song & Zheng, 2026), turning a sequence-level signal into a dense token-level one (Xu et al., 2026). On-policy self-distillation conditions the model itself on privileged information (Zhao et al., 2026a; Vapnik & Vashist, 2009; Chen et al., 2020); recent work studies which privileged information helps, finding abstracted guidance better than raw solutions (Zhao et al., 2026b), and extends the idea to visual question answering (Bi et al., 2026) and to multi-turn agents (Su et al., 2026). These methods condition the teacher on a reference solution or on the input image; in contrast, DuoOPSD conditions two teachers on visual and structural expert diagnoses of aligned executions and applies the loss on the verification turns. We discuss self-improving agents in Appendix A.

3 Method

In this section, we present our self-improving framework for multimodal problem solving. After the problem formulation and overview (Section 3.1), we introduce the harness DuoAgent (Section 3.2) and then the model updates of its solver: verified synthetic task generation (Section 3.3) and DuoOPSD (Section 3.4).

3.1 Problem Formulation and Overview

Problem formulation. An instance of multimodal problem solving consists of an issue description x, one or multiple screenshots that illustrate the reported issue I_{report}, and a buggy repository P_{bug} at a fixed commit. An agent must produce a patch Δ such that the candidate program $P_{\text{cand}} = P_{\text{bug}} + \Delta$ passes hidden tests that fail on P_{bug}; the benchmark harness returns the outcome $r(\Delta) \in \{0, 1\}$. We model the agent as a policy π_θ over multi-turn tool use: at turn t it conditions on the history h_t, including all previous tool results and images, and generates the assistant tokens y_t.

Overview. DuoAgent (Section 3.2) is, to our knowledge, the first multi-agent harness that verifies every candidate patch by replaying the reported interactions in a browser: an interactive sandbox B with a fixed action sequence A that drives it into the reported state turns a visual report into an executable specification. The replayed executions also drive self-improvement (Figure 1, center): the reproducer and the verifier update their *skills* (Section 3.2), and the solver π_θ updates its *model* θ. The same environments make injected bugs verifiable, which enables the first verified synthetic

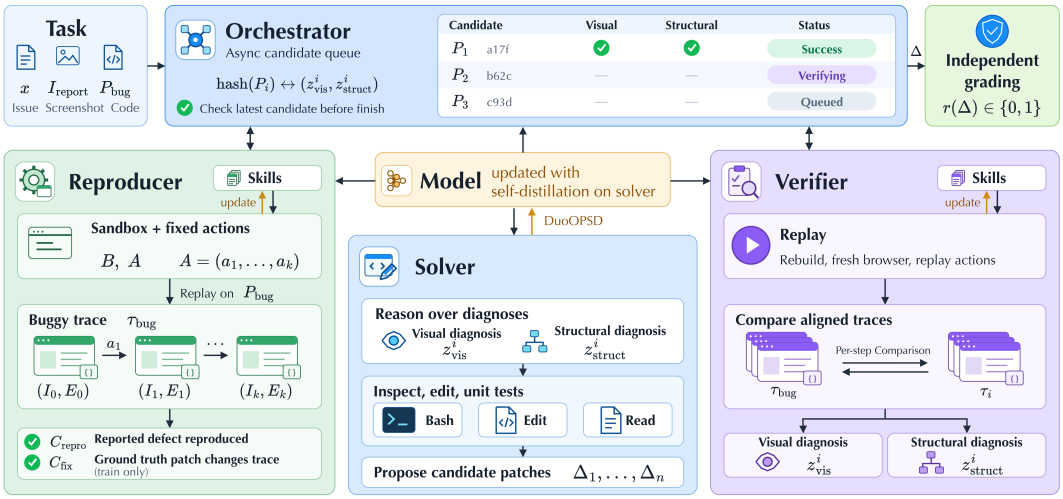


Figure 1: **Overview of the self-improving framework.** The reproducer builds the sandbox B and the actions A and records the buggy trace τ_{bug}; the solver proposes candidate patches, and the verifier replays each and returns a visual and a structural diagnosis, while the orchestrator tracks the candidates. The reproducer and the verifier update their skills, and the model is updated with DuoOPSD on the solver's rollouts.

task generation for visual issues (Section 3.3), and provide the aligned buggy and correct executions from which DuoOPSD derives its teachers (Section 3.4).

3.2 DuoAgent: Multi-Agent Debugging Harness

DuoAgent decomposes multimodal debugging into four agents (Figure 1): an *orchestrator* owns the task state and dispatches the other three; a *reproducer* constructs and verifies the reproduction, a *solver* proposes candidate patches, and a *verifier* re-executes the reproduction on each candidate and diagnoses it against the buggy trace. No agent reads another agent's context: the sandbox and actions, the candidates, and the diagnoses are the only objects exchanged, via the orchestrator.

Orchestrator. The orchestrator obtains a certified reproduction $(B, A, \tau_{\text{bug}})$ from the reproducer, runs the solver and the verifier asynchronously around a queue of candidates $P_i = P_{\text{bug}} + \Delta_i$ whose diagnoses it returns to the solver, requires the latest candidate to be verified before a run ends, and submits the patch to the untouched benchmark harness (Appendix B).

Reproducer. Given $(x, I_{\text{report}}, P_{\text{bug}})$, the reproducer R builds a browser sandbox B, a minimal application that mounts the affected component from the repository's local build, together with a fixed action sequence $A = (a_1, \ldots, a_k)$ of clicks, inputs, and navigations that drives B into the reported state. Executing A in B on a program P yields a trace

$$\tau(P) = \big((I_0, E_0),\, a_1,\, (I_1, E_1),\, \ldots,\, a_k,\, (I_k, E_k)\big), \tag{1}$$

where I_t denotes the screenshot and E_t the browser state after step t, including the DOM outline, the accessibility tree, the geometry and computed styles of visible elements, console and network errors, and the URL. The reproducer iterates until a check $C_{\text{repro}}(x, I_{\text{report}}, A, \tau_{\text{bug}})$ confirms that the buggy trace $\tau_{\text{bug}} = \tau(P_{\text{bug}})$ exhibits the reported defect (Appendix D).

Verifier. The verifier examines a candidate with a replay tool,

$$\textsc{Replay}(P_i; B, A) \longrightarrow \tau_i, \tag{2}$$

which rebuilds the program, executes A in a fresh browser session, and records the candidate trace τ_i. The verifier then compares the two traces step by step, (I_t, E_t) of τ_{bug} against (I_t^i, E_t^i) of τ_i for every t, and writes two diagnoses,

$$z^i_{\text{vis}} = D_{\text{vis}}(\tau_{\text{bug}}, \tau_i), \qquad z^i_{\text{struct}} = D_{\text{struct}}(\tau_{\text{bug}}, \tau_i), \tag{3}$$

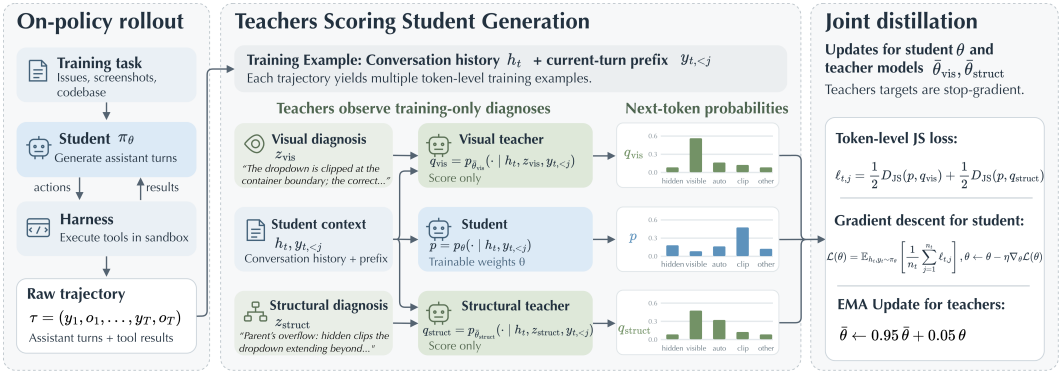


Figure 2: **Overview of DuoOPSD.** *Left:* The student rolls out repairs under DuoAgent with real tools; the harness does not let a run end with unverified changes, so every trajectory contains *verification turns*, the assistant turns in which the solver reads the verifier's diagnoses of a candidate and decides what to do next. *Middle:* Two teachers sharing the student's EMA weights score the same continuation, each with the visual or the structural expert diagnosis appended to the task prompt; the diagnoses are training-only and never shown to the student. *Right:* The student matches both teachers on the verification turns only; the rest is context.

a *visual diagnosis* from the aligned screenshots, stating whether the reported appearance changed and whether anything else regressed, and a *structural diagnosis* stating the same for the DOM, styles, console errors, and action outcomes. The verifier never reads or edits the source code.

Solver. The solver π_θ is the main coding agent. It receives the task and the reproducer's action sequence, and works in rounds: it reasons over the diagnoses of its previous candidates, inspects and edits the repository with ordinary file and shell tools, and proposes the next candidate Δ_i.

Skills. The reproducer and the verifier load a library of natural-language skills (Wang et al., 2023), which they update on the development split from the outcomes of their own runs: the reproduction checks for the reproducer, and the agreement of its verdicts with the hidden tests for the verifier. The skills are then frozen for all test runs and training rollouts (Appendix C).

3.3 Verified Synthetic Task Generation

Improving the solver's model needs data beyond the 100 development instances, and existing synthetic pipelines (Yang et al., 2026; Pan et al., 2024) verify injected bugs with unit tests. A development instance with a browser environment removes this obstacle, since any change to the code exercised by A shows up in the replayed trace. Beyond verification, a useful pool must be *varied*, spanning the kinds of visual defects a repository exhibits, and of the *right difficulty*, tasks the student fails but that are solvable. Starting from the working program P_{good} of a development seed, a strong model reads the code behind the sandbox's interactive elements and proposes localized mutations m under ten strategies, from layout and overflow to option handling and state updates (Appendix F), for variety. Replaying (B, A) on $P_{\text{syn}} = m(P_{\text{good}})$ and P_{good} yields aligned traces τ_{syn} and τ_{good}, from which we derive a behavioral check that accepts any repair restoring the stable fields and pixels of τ_{good}, so that alternative repairs pass and invisible or unstable mutations are rejected. For each valid mutation, a strong model writes a synthetic issue with reporter visuals and two expert diagnoses of the aligned traces, which serve as training-only privileged information (Section 3.4). For difficulty, following Kulikov et al. (2026), we keep the tasks a strong agent solves, including the hard subset the untrained student fails; test instances are never used (Appendix F).

3.4 DuoOPSD: Multi-Teacher On-Policy Self-Distillation

With verified tasks in hand, the question is the training signal. Imitating a strong agent exceeds the student's context budget, and the outcome reward of RL (Shao et al., 2024) is silent about which of a hundred tool calls went wrong (Table 3). On-policy distillation instead scores every token of the student's own trajectories (Agarwal et al., 2024; Xu et al., 2026), and on-policy self-distillation obtains the teacher from the student itself by giving it privileged information (Zhao et al., 2026a; Bi

et al., 2026). This matches our setting: the untrained student already resolves about a quarter more development instances when an expert diagnosis is in its context (Appendix H), so what it lacks is the ability to use a diagnosis, and a teacher that is the student plus the diagnosis demonstrates exactly this ability on the student's own trajectories. The privileged information must also be usable by the network: the correct trace itself does not help the student when handed to it directly, whereas a diagnosis written from it does. Since a defect has a visual and a structural view, each gets its own teacher (Figure 2).

On-policy rollouts and verification turns. We collect trajectories from the current student through the same harness, tools, and frozen skills it uses at test time. For assistant turn t, the prefix h_t contains the actual tool results and $y_t = (y_{t,1}, \ldots, y_{t,n_t})$ is the generated continuation. The *verification turns* $\mathcal{V}$ are the turns whose prefix ends with the verifier's diagnoses of a candidate, in which the solver decides whether its fix worked and what to change next; since a run may not end unverified (Section 3.2), every trajectory that edits the program contains one.

Diagnosis-conditioned teachers. At test time the verifier's diagnoses compare a candidate with the buggy execution: they tell the solver what its patch changed, not what the correct behavior is. On a training task we know more, since its correct execution τ_{gold} is available (Section 3.3), and an expert diagnosis of τ_{bug} against τ_{gold} names the discrepancy and the target state. Let $\bar{\theta}$ denote an exponential moving average of the student weights θ. A teacher is the same network with a *Diagnosis* section z appended to the task prompt, as is standard in on-policy self-distillation (Zhao et al., 2026a), and scores the student's own continuation with an otherwise identical context; the diagnoses are stored with each task like the reference solution of OPSD:

$$p_{\theta,t,j} = p_\theta(\cdot \mid h_t, y_{t,<j}), \qquad q_{k,t,j} = p_{\bar{\theta}}(\cdot \mid h_t, z_k, y_{t,<j}), \quad k \in \{\text{vis}, \text{struct}\}, \tag{4}$$

where z_k is an *expert diagnosis* in the same form as the verifier's but written from the aligned buggy and correct executions, without code: z_{vis} describes the visible discrepancy and the target appearance, and z_{struct} the structural or behavioral discrepancy. Teachers only score the student's continuation, so no teacher rollout is needed.

Training objective. With the teacher targets detached from the gradient, we minimize the token-averaged Jensen–Shannon divergence to both teachers over the verification turns,

$$\mathcal{L}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\frac{1}{\sum_{t \in \mathcal{V}} n_t} \sum_{t \in \mathcal{V}} \sum_{j=1}^{n_t} \left(\tfrac{1}{2} D_{\text{JS}}\big(p_{\theta,t,j}, q_{\text{vis},t,j}\big) + \tfrac{1}{2} D_{\text{JS}}\big(p_{\theta,t,j}, q_{\text{struct},t,j}\big)\right)\right], \tag{5}$$

estimated over the student's top-k tokens plus a tail bucket, with the teachers updated as $\bar{\theta} \leftarrow 0.95\,\bar{\theta} + 0.05\,\theta$ after each step (Appendix G). We restrict the loss to $\mathcal{V}$ because the expert diagnosis changes what a reader of the verifier's diagnoses should conclude, but not how to list a directory, so over a whole trajectory the disagreement that matters is diluted by hundreds of reading turns. Across rounds, the improved student also becomes the next teacher through $\bar{\theta}$, so the solver improves recursively. The on-policy baselines share this loop and differ only in the training signal, while SFT trains on the strong agent's trajectories (Appendix G).

4 Experiments

Through evaluations on SWE-bench Multimodal for DuoAgent and DuoOPSD, we find that (1) DuoAgent consistently outperforms existing methods under different models. Notably, DuoAgent achieves 52.0% under Claude Fable 5.1, establishing a new state of the art that is 16 points above the best prior system; (2) the gains are consistent for different repositories and concentrate in the repositories whose issues are visible in a rendered page; (3) among post-training methods trained on the same training pool with the same number of optimizer steps, DuoOPSD gives the largest gain for both students; and (4) our design in DuoOPSD, which gives the visual and the structural expert diagnoses to two separate teachers, outperforms other design variants such as teachers given the raw execution evidence instead of the diagnoses and single-teacher variants.

Table 1: **Resolution rate (%) on the SWE-bench Multimodal test split.** DuoAgent achieves higher resolution rate across different models compared to their native harness, with a slight increase in cost. We evaluate the 480 instances with images available in the v2 benchmark but use all 517 as the denominator for comparability to published results. The remaining 37 instances are marked as unresolved.

Harness	Model	#Resolved	%Resolved	Cost
SWE-agent Multimodal	GPT-4o	63	12.2	–
Agentless Lite	Claude-3.5 Sonnet	131	25.3	–
OpenHands-Versa	Claude Sonnet 4	178	34.4	–
GUIRepair	GPT-4.1	161	31.1	–
GUIRepair	o3	186	36.0	–
pi	Qwen3.5-4B	82	15.9	–
DuoAgent	Qwen3.5-4B	105	20.3 (+4.4)	–
Codex	o3	128	24.8	$0.84
DuoAgent	o3	189	36.6 (+11.8)	$0.97
Codex	GPT-6 Astra (medium)	202	39.1	$0.75
DuoAgent	GPT-6 Astra (medium)	211	40.8 (+1.7)	$0.96
Claude Code	Claude Fable 5.1 (medium)	242	46.8	$1.13
DuoAgent	Claude Fable 5.1 (medium)	**269**	**52.0** (+5.2)	$1.45

4.1 Evaluation Setup

Benchmark. We evaluate on SWE-bench Multimodal (Yang et al., 2024b), which consists of issues from open-source JavaScript libraries with reporter screenshots, graded by hidden tests. Its test split has 517 instances, of which the latest public task repository[1] provides evaluation environments for 480; the remaining 37 (all 24 of quarto-cli and 13 others) count as unresolved, and 517 is the denominator throughout for comparability with published results. The development split is used for construction checks, skill updates, synthetic task generation, and validation.

Metrics. The primary metric is the resolution rate, the percentage of test instances whose hidden tests pass after the submitted patch is applied. Following GALA (Liu et al., 2026), we also report file- and function-level *localization recall*, the fraction of the files, or of the hunks by enclosing function or line range, modified by the reference repair that the submitted patch also modifies, averaged over instances; test files are excluded, an instance without a patch has recall zero, and the reference repairs never enter an agent's context (Appendix D).

Models and harnesses. We evaluate three frontier models, Claude Fable 5.1, GPT-6 Astra, and o3, and the open-weight Qwen3.5-4B; for post-training we additionally use the open-weight Gemma-4-E4B. Each model is run under its native harness, Claude Code for Claude Fable 5.1, Codex for GPT-6 Astra and o3, and pi for Qwen3.5-4B, and under DuoAgent. Every configuration has the same repository access and instructions, and web access is disabled throughout. Skills are learned on the development split and frozen (Section 3.2). Any pre-training exposure of later frontier models to the benchmark's fixes affects both arms of each comparison alike. We further report the published numbers of prior systems (Yang et al., 2024b; Xia et al., 2024; Soni et al., 2026; Huang et al., 2025).

Baselines. For post-training the student, we consider (1) **SFT** on resolved trajectories of a strong agent collected through DuoAgent under the student's context budget; (2) **GRPO** (Shao et al., 2024) with outcome rewards; and (3) **OPSD** (Zhao et al., 2026a), on-policy self-distillation with the reference patch as privileged information. GRPO and OPSD share the student's rollouts, harness, and trainer with DuoOPSD and differ only in the loss; SFT instead learns from the strong agent's trajectories (Table 11). *Same budget* means the same optimizer steps (and rollouts, for on-policy methods), not compute: distillation adds teacher forward passes, and GRPO keeps a group size of 4 and the shared learning rate.

Training protocol. The training pool contains the development seeds and the synthetic tasks generated from them, while a fixed set of 19 seeds and all their descendants is held out for validation

[1] https://github.com/SWE-bench/swe-bench-multimodal-tasks

Table 2: **Localization recall (%) on the test split**, file-level and function-level. DuoAgent improves the localization recall across different base models.

Harness	Model	Loc. Recall (%)	
		File	Function
pi	Qwen3.5-4B	36.1	32.5
DuoAgent	Qwen3.5-4B	43.8	40.9
Codex	o3	42.5	39.7
DuoAgent	o3	50.3	47.2
Codex	GPT-6 Astra	61.1	57.4
DuoAgent	GPT-6 Astra	62.3	59.3
Claude Code	Fable 5.1	67.2	64.2
DuoAgent	Fable 5.1	**72.0**	**69.6**

Table 3: **Resolution rate (%) of different post-training methods.** All methods are based on DuoAgent. The gains are relative to the untrained student under DuoAgent. DuoOPSD achieves the best performance compared to different post-training algorithms.

Method	Qwen3.5-4B	Gemma-4-E4B
DuoAgent (Base)	20.3	13.2
+ SFT	21.9 (+1.5)	13.9 (+0.8)
+ GRPO	22.4 (+2.1)	16.4 (+3.3)
+ OPSD	22.2 (+1.9)	14.3 (+1.2)
+ DuoOPSD (ours)	**24.2** (+3.9)	**17.2** (+4.1)

(Appendix F). Rollouts are graded by the harness for seeds and by the behavioral check for synthetic tasks. Validation uses a held-out probe of 64 synthetic tasks, of which the untrained student resolves 43.8% (Appendix G).

4.2 DuoAgent Outperforms Existing Harnesses Across Models

Table 1 compares each model under its native harness and under DuoAgent on the test split, alongside prior systems; Table 2 reports localization recall, and Table 6 in Appendix E breaks the results down by repository. DuoAgent consistently achieves higher performance and sets a new state of the art: Claude Fable 5.1 rises from 46.8% to 52.0%, 16 points above the best prior system (GUIRepair with o3, 36.0%), and Qwen3.5-4B from 15.9% to 20.3%. o3 under DuoAgent reaches 36.6%, surpassing GUIRepair with the same model, although GUIRepair requires human-in-the-loop environment setup per instance while DuoAgent is fully automated.

Per repository (Appendix E), the gains concentrate where the reported behavior is visible in a rendered page, such as bpmn-js, carbon, and prism, which is exactly what the replay provides.

DuoAgent also improves localization by 1.2 to 8.4 points (Table 2), since the verifier's reports of which element and browser state a candidate changed point the solver to the relevant code.

4.3 DuoOPSD Outperforms Post-Training Baselines

Table 3 compares DuoOPSD with three post-training methods on Qwen3.5-4B and Gemma-4-E4B, all sharing the training pool and a budget of 64 optimizer steps (Table 11).

DuoOPSD gives the largest gain on Qwen3.5-4B (24.2%, against 22.4% for GRPO) and on Gemma-4-E4B (17.2%, against 16.4% for GRPO). The ceiling analysis explains why (Appendix H): with the expert diagnoses in its context, the untrained student resolves 26 of 98 development instances instead of 21, and 90% instead of 75% of its edits touch the files of the reference repair. The student thus lacks the ability to read a diagnosis, which is what DuoOPSD's teachers supply; its training rollouts resolve more tasks in every round (33.2% to 45.3% on Qwen3.5-4B).

The baselines fall short for different reasons. SFT is limited by context: the expert's whole-file reads exhaust a 64k-context student's budget. GRPO is limited by its reward: one binary outcome for a hundred tool calls cannot tell which decision was right, a credit-assignment problem that dense on-policy distillation avoids (Agarwal et al., 2024; Xu et al., 2026). OPSD has dense supervision but the wrong privileged information: a reference patch says what to write, not how to judge a candidate, in line with findings (Zhao et al., 2026b) that abstracted guidance helps more than a raw solution.

4.4 Ablation Studies

Beyond on-policy self-distillation itself, DuoOPSD rests on two design choices: *what the student is trained on*, and *what the teachers know*. We ablate both along three axes, changing one factor at a time relative to DuoOPSD. (i) *Training data*: the seeds alone, the seeds with all strong-solved

Table 4: **Ablation studies of DuoOPSD.** We report the resolution rate (%) on the same held-out probe of 64 synthetic tasks. Counts: seeds + synthetic tasks. Each row differs from DuoOPSD (shaded) in the training data, or the number of teachers and their privileged information z. Separate: one per teacher; merged: both in one teacher's context. Two separate diagnosis teachers on the full synthetic pool (DuoOPSD) perform best on Qwen3.5-4B and within one task of the best variant on Gemma-4-E4B.

Training data	Teachers	Privileged information z	Qwen3.5-4B	Gemma-4-E4B
Untrained student	–	–	43.8	26.6
Seeds + all synthetic (79 + 386)	2	separate visual / structural diagnoses	**51.6**	28.1
Training data				
Seeds (79)	2	separate visual / structural diagnoses	37.5	20.3
Seeds + hard synthetic (79 + 202)	2	separate visual / structural diagnoses	29.7	23.4
Expert diagnoses				
Seeds + all synthetic (79 + 386)	1	visual diagnosis	42.2	25.0
Seeds + all synthetic (79 + 386)	1	structural diagnosis	34.4	**29.7**
Seeds + all synthetic (79 + 386)	1	merged visual / structural diagnoses	34.4	21.9
Raw execution evidence				
Seeds + all synthetic (79 + 386)	1	screenshots	37.5	23.4
Seeds + all synthetic (79 + 386)	1	browser states	48.4	25.0
Seeds + all synthetic (79 + 386)	2	separate screenshots / browser states	37.5	25.0

synthetic tasks (DuoOPSD), or the seeds with only the hard subset (Section 3.3). (ii) *Organization of the diagnoses*: one teacher with the visual, the structural, or both diagnoses, versus two teachers with one each (DuoOPSD). (iii) *Diagnoses versus raw evidence*: each diagnosis is replaced by the aligned screenshots or browser states it is written from.

To isolate these factors, every variant is trained for one round (16 optimizer steps) from the same untrained student, and the teacher ablations share the same rollouts, so that they differ only in z; due to limited resources, we evaluate on the probe.

Diverse synthetic data is necessary. Training on the seeds together with all synthetic tasks performs best on both students (51.6% and 28.1%). With the seeds alone or only the hard subset, Qwen3.5-4B even falls below the untrained student (37.5% and 29.7%): the seeds cover too few kinds of defects, and on always-failed tasks the candidates are too far off for the teachers to correct. The synthetic pool thus helps through its variety and mix of difficulties, not through difficulty alone, which is why it retains every task that the strong agent solves.

Separate teachers outperform a merged teacher. Merging both diagnoses into one teacher is the weakest organization (34.4% and 21.9%), 17.2 and 6.2 points below DuoOPSD. In one context the two diagnoses compete for the teacher's attention, whereas with two teachers each shapes its own target distribution. A structural-only teacher is slightly better than DuoOPSD on Gemma-4-E4B (29.7% versus 28.1%) but poor on Qwen3.5-4B, likely because Qwen3.5-4B makes better use of visual evidence, while the visual teacher of Gemma-4-E4B is sometimes noisy.

Diagnoses are more effective than raw evidence. Replacing the diagnoses with the raw evidence they are written from lowers the rate from 51.6% to 37.5% and from 28.1% to 25.0%. Raw screenshots and browser states contain the discrepancy, but the teacher must locate it among many unchanged pixels and fields, which is hard for small models, whereas a diagnosis names the discrepancy and the target state as the verifier does, giving the student a signal it can act on.

5 Conclusion

We presented a self-improving framework for visual software issues, which unit tests cannot catch: DuoAgent reproduces the reported behavior in a browser and verifies every candidate against it, the same environments make injected bugs verifiable for synthetic training, and through DuoOPSD the solver improves its own model from expert diagnoses. DuoAgent improves every evaluated model, reaching a new state of the art of 52.0%, and DuoOPSD outperforms SFT, GRPO, and OPSD. These results suggest that once agents can see failures that are otherwise invisible, the same observations drive resolving issues and self-improvement, from frontier models to small open-weight students.

AI USE STATEMENT

In this work, we used generative AI tools for designing experiments, generating synthetic datasets, implementing methods, and interpreting results. We have not used generative AI tools for generating research ideas and literature review, and assistance in providing mathematical claims and writing proofs are not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have made every effort to ensure the reproducibility of our results. The paper and its appendix describe the design of DuoAgent, the skill update of its reproducer and verifier, the construction and validation of the sandboxes, the synthetic task generation, and the training procedure of DuoOPSD and all baselines, including the full hyperparameters. All experiments are conducted on the public SWE-bench Multimodal benchmark with publicly available models. We will release the code, the trained models, and the datasets, including the sandboxes and the synthetic tasks, upon acceptance.

REFERENCES

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, volume 2024, pp. 21246–21263, 2024.

Jinhe Bi, Peng Liao, Zengjie Jin, Volker Tresp, Fei Shen, Yunpu Ma, Tat-Seng Chua, et al. Opd-v: Visual on-policy self-distillation with modality balance. *arXiv preprint arXiv:2608.05131*, 2026.

Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. Learning by cheating. In *Conference on robot learning*, pp. 66–75. PMLR, 2020.

Irving John Good. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pp. 31–88. Elsevier, 1966.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *International Conference on Learning Representations*, volume 2024, pp. 32694–32717, 2024.

Kai Huang, Jian Zhang, Xiaofei Xie, and Chunyang Chen. Seeing is fixing: Cross-modal reasoning with multimodal llms for visual software issue repair. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1156–1168. IEEE, 2025.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 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 (Volume 1: Long Papers)*, pp. 881–905, 2024.

Ilia Kulikov, Chenxi Whitehouse, Tianhao Wu, Yixin Nie, Swarnadeep Saha, Eryk Helenowski, Weizhe Yuan, Olga Golovneva, Jack Lanchantin, Yoram Bachrach, et al. Autodata: An agentic data scientist to create high quality synthetic data. *arXiv preprint arXiv:2606.25996*, 2026.

Zhuoyao Liu, Zhengran Zeng, Shu-Dong Huang, Yang Liu, Shikun Zhang, and Wei Ye. Gala: Multimodal graph alignment for bug localization in automated program repair. *arXiv preprint arXiv:2604.08089*, 2026.

Kevin Lu and Thinking Machines Lab. On-policy distillation. Thinking Machines Lab: Connectionism, 2025. https://thinkingmachines.ai/blog/on-policy-distillation/.

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. In *International Conference on Learning Representations*, volume 2026, pp. 83478–83525, 2026.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Hannah Sansford, Derek HC Law, Wei Liu, Abhishek Tripathi, Niresh Agarwal, and Gerrit JJ Van den Burg. Vision-guided iterative refinement for frontend code generation. *arXiv preprint arXiv:2604.05839*, 2026.

Jürgen Schmidhuber. Godel machines: Self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs.LO/0309048*, 2003.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3956–3974, 2025.

Mingyang Song and Mao Zheng. A survey of on-policy distillation for large language models. *arXiv preprint arXiv:2604.00626*, 2026.

Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig. Coding agents with multimodal browsing are generalist problem solvers. In *Findings of the Association for Computational Linguistics: EACL 2026*, pp. 6052–6069, 2026.

Xingyu Su, Abhishek Kumar, Qing Ping, Youzhi Luo, Jonathan Buck, Zach Zhang, Subramanian Chidambaram, and Vinayak Arannil. From self-distillation to self-practice: Privileged information for multi-turn agents, 2026. URL https://arxiv.org/abs/2609.29051.

Vladimir Vapnik and Akshay Vashist. A new learning paradigm: Learning using privileged information. *Neural networks*, 22(5-6):544–557, 2009.

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, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Yuanda Xu, Hejian Sang, Zhengze Zhou, Ran He, Zhipeng Wang, and Alborz Geramifard. Beyond grpo and on-policy distillation: An empirical sparse-to-dense reward principle for language-model post-training. *arXiv preprint arXiv:2605.12483*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024a.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024b.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 2024.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

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.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. Self-distilled reasoner: On-policy self-distillation for large language models. *arXiv preprint arXiv:2601.18734*, 2026a.

Xuyang Zhao, Liting Zhang, Zichen Xu, Zhihu Wang, Xu Caiyue, Shiwan Zhao, and Qicheng Li. Is more privileged information better? from solution traces to problem-solving structure in self-distilled reasoning. *arXiv preprint arXiv:2608.01589*, 2026b.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

A Additional related work

Self-improving agents. Self-improvement dates back to recursive self-improvement (Good, 1966) and the Gödel machine (Schmidhuber, 2003). LLM agents improve (1) *in context*, by reflecting on failures (Shinn et al., 2023), extracting insights (Zhao et al., 2024), accumulating skill libraries (Wang et al., 2023; 2024), or editing their own scaffold (Robeyns et al., 2025; Zhang et al., 2026); or (2) *in weights*, by training on their own filtered or self-judged outputs (Zelikman et al., 2022; Yuan et al., 2024). These methods improve one agent through one channel; our framework improves the skills of the reproducer and the verifier and the weights of the solver, grounding every update in replayed executions.

B Details of DuoAgent

Orchestration. The orchestrator first dispatches the reproducer and accepts its output once the reproduction is certified; an instance whose report cannot be reproduced falls back to the solver's native workflow without a sandbox. It then launches the solver and the verifier asynchronously around a *candidate queue*: every candidate P_i that the solver proposes is queued and sent to the verifier together with $(B, A, \tau_{\text{bug}})$, and the verifier's diagnoses are stored under $\text{hash}(P_i)$ and handed back to the solver. The solver therefore never waits for a verification, several candidates can be verified at the same time, and every diagnosis is matched to the candidate it describes. When the solver ends while its program differs from the last verified one, the orchestrator returns a request to verify first, at most twice per instance; runs that changed nothing or ended in an error are left alone.

Verifier and solver. Every Replay call rebuilds the program when its sources have changed, starts a fresh browser session, and records a new trace; only builds are cached, so a stale observation can never pass for the state of the current program. Each diagnosis ends with a status for its channel, resolved or not, which the orchestrator records for the candidate. The verifier receives the candidate, the reproduced environment, the buggy trace, and its skills $\mathcal{S}_V$, never the repository source. The solver runs the relevant existing tests with ordinary file and shell tools, and is instructed to fix the defect where the data is produced rather than where it is displayed, to sweep for sibling components and mirrored files, and not to modify tests or snapshots.

Agent settings. Every configuration has the same repository access and instructions, and web access is disabled throughout. Each run has a wall-clock budget of 1,800 seconds per instance; a patch present at the deadline is submitted. The open-weight students are served with vLLM at a 64k-token context and an 8,192-token output limit per turn, and sample at temperature 1. Under DuoAgent, a run whose latest candidate is unverified receives at most two requests to verify before it may end.

C Skill self-improvement

Skill format. A skill is a short Markdown document with a title, a one-line condition stating when it applies, the procedure or rule itself, and the failure that motivated it. The reproducer skills $\mathcal{S}_R$ and the verifier skills $\mathcal{S}_V$ are kept in separate libraries. Each library is placed in the agent's context in full, after the task prompt, and is read-only: an agent cannot add to it during a run, which is why the reproducer prompt forbids saving notes for later sessions (Appendix I.1).

Update step. Skills are edited by an update step U, run by the same model as the agent whose skills it edits, so the skills of each model are its own. After a run, U receives the agent's trajectory, the feedback on its outcome (below), and the current library, and returns a list of edits, each of which adds, revises, merges, or deletes a skill. Edits are applied in order; a skill that names an instance, a file or function of a repository, or a code fragment of a repair is rejected, so that the library contains procedures rather than solutions.

Feedback. For the reproducer, the feedback consists of the verdict, the observation, and the reasoning of every C_{repro} attempt, the outcome of every C_{fix} attempt with its changed fields, and the number of attempts until both passed. From runs that fail a check or need many attempts, U writes

what went wrong, for instance a prebuilt bundle that the relink does not refresh, or an action sequence that stops before the state where the repair would show; from runs that pass at once, it writes the procedure that worked. For the verifier, we run the solver under DuoAgent on the development instances with an approved sandbox and collect every verified candidate. Each diagnosis ends with a verdict for its channel, and we write $s^i \in \{0, 1\}$ for whether the verifier judges candidate P_i resolved on both. For each candidate whose verdict s^i disagrees with the hidden tests, U receives the candidate's two diagnoses, the hidden-test outcome $r(\Delta_i)$, and the correct trace τ_{gold}. A disagreement is either a candidate that the verifier accepted but the tests reject, or one that it rejected but the tests accept, and the resulting skills name what the verifier overlooked or over-weighted, such as a regression outside the reported region, a field that varies across identical replays, or a difference that appears only after the last action.

Isolation from the test split. Skills are updated only on the development split and are frozen before any test run. The reference repairs, the correct traces, and the hidden-test outcomes of the development split are read only by U and never reach an agent during a run. The development and test splits of SWE-bench Multimodal share no repository, so skills cannot carry repository-specific knowledge from one split to the other.

D SANDBOX CONSTRUCTION

Development instances additionally pass $C_{\text{fix}}(\tau_{\text{bug}}, \tau_{\text{gold}})$, an objective check that the reference repair changes the trace, whereas test instances use C_{repro} only, so that no information about the reference repair influences a test reproduction. The reference repairs of both splits are taken from the public task repository; they are read only by these checks, by the skill update on the development split (Appendix C), and by the offline localization metrics, and are never mounted into an agent's container or included in its context.

The reproducer. The reproducer is a coding agent that receives the issue text, the reporter's assets (the images of the issue and, when the issue links an online reproduction such as a CodeSandbox, its archived code) and the execution image of the instance, in which the repository is installed at the base commit. It writes a minimal Vite application that imports the library from the local repository and renders the component in the reported state, the scripts that rebuild the library and relink the application after every candidate repair, and the action sequence A, at most eight actions from a fixed vocabulary (click, double-click, hover, fill, select, key press, scroll, wait, and history navigation) together with a one-sentence expectation of what the buggy page shows. It iterates with commands that build the sandbox image, dump the interactive elements of the served page, record τ_{bug}, and run each check, until the checks pass (prompt in Appendix I.1). It may not modify the repository or its tests, embed the reporter's screenshots, or render any text that describes the defect, so the defect must come from the library as built; the reference repair is never visible to it.

The checks. For C_{repro}, the reproducer checks whether τ_{bug} exhibits the reported defect. The check receives (i) the issue text, (ii) up to three of the reporter's own screenshots, labeled as such, (iii) the action sequence A with the reproducer's expectation, (iv) the full-page screenshots of τ_{bug} at up to eight steps, always including the first and the last, each labeled with its pixel size against the 1280×720 viewport, and (v) the runtime evidence of the trace as text: the action and outcome of every step, URL changes, and console errors. It never uses the reference repair. The check returns a verdict, a confidence, a description of what the trace shows, and its reasoning (Appendix I.2); it passes only for a positive verdict with high or medium confidence, and after a failed check the reproducer revises the application or the actions based on that reasoning. The sandbox need not look like the reporter's application; only the defective behavior must be present. C_{fix}, in contrast, involves no model: it replays A on the program with the reference repair applied and passes when τ_{gold} differs from τ_{bug} at some step, in the screenshot or in the browser state (DOM outline, accessibility tree, element geometry and styles, console errors, and action outcomes).

Table 5 reports the outcome of the reproducer on the development and test splits, with Claude Fable 5.1 as the model, stage by stage: the instances with an execution image, those for which a sandbox application was built from the repository, those for which an action sequence reproduces the reported behavior under C_{repro}, those for which the reference repair changes the recorded trace

under C_{fix}, and the approved sandboxes. C_{fix} gates approval on the development split only; on the test split it is run for information and the reference repair never influences the sandbox, so the three test instances without an approved sandbox are the ones for which no action sequence passed C_{repro}. The lower block gives the length of the recorded action sequences of the approved sandboxes.

Table 5: **Sandbox construction on the development and test splits** (number of instances at each stage), with Claude Fable 5.1 as the reproducer's model. C_{repro}: the recorded actions reproduce the reported behavior in the sandbox. C_{fix}: the reference repair changes the recorded trace; it gates approval on the development split only and is informational on the test split. The C_{fix} rows break down the instances that pass C_{repro}. Actions are the recorded browser steps from the loaded page to the reported behavior.

Stage	**Dev**	**Test**
Instances with an execution image	100	480
Sandbox application built from the repository	100	480
Action sequence reproduces the report (C_{repro})	98	477
Reference repair changes the trace (C_{fix})	98	407
repaired trace identical to the buggy trace	0	31
reference repair does not apply or build	0	37
check not completed	0	2
Approved sandboxes	98	477
Recorded actions per approved sandbox		
none (defect visible on load)	63	263
one	20	96
two or more	15	118

E PER-REPOSITORY RESULTS

Table 6 breaks the test-split results of Table 1 down by repository, with each model under its native harness and under DuoAgent side by side.

Table 6: **Resolved instances by repository on the test split.** Each cell gives the count and the percentage of the repository's test instances (517 in total); the best cell per row is shaded green. The 37 instances without a public evaluation image in the v2 benchmark, including all of quarto-cli, are counted as unresolved.

Repo	**Num**	**Qwen3.5-4B**		**o3**		**GPT-6 Astra**		**Claude Fable 5.1**	
		pi	DuoAgent	Codex	DuoAgent	Codex	DuoAgent	Claude Code	DuoAgent
next	39	5 (12.8%)	10 (25.6%)	8 (20.5%)	12 (30.8%)	12 (30.8%)	13 (33.3%)	14 (35.9%)	15 (38.5%)
bpmn-js	54	5 (9.3%)	9 (16.7%)	19 (35.2%)	31 (57.4%)	31 (57.4%)	31 (57.4%)	34 (63.0%)	38 (70.4%)
carbon	134	8 (6.0%)	15 (11.2%)	20 (14.9%)	28 (20.9%)	33 (24.6%)	35 (26.1%)	45 (33.6%)	51 (38.1%)
eslint	11	2 (18.2%)	3 (27.3%)	6 (54.5%)	6 (54.5%)	5 (45.5%)	5 (45.5%)	5 (45.5%)	7 (63.6%)
lighthouse	54	6 (11.1%)	8 (14.8%)	3 (5.6%)	11 (20.4%)	11 (20.4%)	13 (24.1%)	23 (42.6%)	28 (51.9%)
grommet	21	2 (9.5%)	2 (9.5%)	7 (33.3%)	8 (38.1%)	7 (33.3%)	7 (33.3%)	7 (33.3%)	7 (33.3%)
highlight.js	39	4 (10.3%)	4 (10.3%)	5 (12.8%)	11 (28.2%)	13 (33.3%)	13 (33.3%)	20 (51.3%)	20 (51.3%)
openlayers	79	47 (59.5%)	50 (63.3%)	51 (64.6%)	60 (75.9%)	66 (83.5%)	67 (84.8%)	66 (83.5%)	67 (84.8%)
prettier	13	1 (7.7%)	2 (15.4%)	3 (23.1%)	6 (46.2%)	7 (53.8%)	7 (53.8%)	8 (61.5%)	8 (61.5%)
prism	38	2 (5.3%)	2 (5.3%)	6 (15.8%)	16 (42.1%)	16 (42.1%)	19 (50.0%)	19 (50.0%)	25 (65.8%)
quarto-cli	24	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
scratch-gui	11	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (9.1%)	1 (9.1%)	1 (9.1%)	3 (27.3%)
Total Resolved		82	**105**	128	**189**	202	**211**	242	**269**
%Resolved		15.9%	**20.3%**	24.8%	**36.6%**	39.1%	**40.8%**	46.8%	**52.0%**

Static and dynamic issues. Table 7 splits the test instances with an approved sandbox by whether the reported defect is visible on load (*static*, no recorded actions) or appears only after at least one interaction (*dynamic*), and reports the resolved instances of each model under its native harness and under DuoAgent within each group.

F SYNTHETIC TASK GENERATION

Bug injection. Applying the reference repair to a seed yields the working program P_{good}. We identify the interactive elements of the sandbox, namely those touched by A and those whose state

Table 7: **Resolved test instances on static and dynamic issues.** Static: the defect is visible on load (263 instances); dynamic: it appears only after at least one interaction (214 instances). DuoAgent improves every model on both groups, so its gains are not confined to issues that a single screenshot already shows.

		Static (263)		Dynamic (214)	
Model	**Native harness**	**Native**	**DuoAgent**	**Native**	**DuoAgent**
Qwen3.5-4B	pi	42	53 (+11)	39	51 (+12)
o3	Codex	72	104 (+32)	55	84 (+29)
GPT-6 Astra	Codex	115	119 (+4)	85	90 (+5)
Claude Fable 5.1	Claude Code	133	149 (+16)	107	118 (+11)

the reference repair changes, from the element dump and the aligned seed traces. A strong model reads the source code behind these elements and proposes localized mutations m following ten strategies: layout, overflow, positioning, visibility, text content, drawing math, option handling, event handling, interaction conditions, and state updates (Table 10); replaying (B, A) on $P_{\text{syn}} = m(P_{\text{good}})$ and on P_{good} yields aligned traces that share the same actions.

Validity and behavioral check. A trace difference establishes observability but not a bug. We require a successful build and derive a behavioral check from the correct execution: the *stable* fields of τ_{good}, those that do not vary across repeated correct replays, together with the pixel region in which τ_{syn} and τ_{good} differ. A candidate repair passes when its replay reproduces the stable fields, removes most of the pixel difference, and changes no other stable field. Alternative repairs therefore pass without matching the inverse mutation m^{-1}, while mutations whose effect is invisible or unstable are rejected.

Task grounding and reference material. For every valid mutation, a strong model writes a synthetic issue description and selects reporter visuals from the observed defect, so that the task is presented to the student exactly like a benchmark instance, and writes the two expert diagnoses from the aligned traces. The repository is handed to agents as a single commit without history, so the mutation cannot be recovered from version control. The correct trace, the inverse mutation, and the diagnoses remain with the task as training-only material that no student sees.

Difficulty filtering and splits. Following Kulikov et al. (2026), we run a strong agent and the untrained student on every valid task with the same tools and budgets and grade both with the behavioral check. Tasks that the strong agent solves and the student fails constitute the hard subset, and generation continues until a preset number of hard tasks is reached. The training pool retains every task the strong agent solves, so that training batches mix solved and unsolved tasks. Seeds are split into training and validation before generation, every descendant stays with its seed, and benchmark test instances are never used for generation or training.

Statistics. Table 8 gives the statistics of the pool, Table 9 breaks it down by repository, and Table 10 by mutation strategy. Synthetic tasks come from 93 of the 98 development seeds with an approved sandbox, across five repositories; the hard subset is spread over all strategies, with option handling, state updates, and layout contributing the most.

G Training Details

Relation between the compared methods. GRPO, OPSD, and DuoOPSD share the rollout loop of Section 3.4 and differ only in the training signal; SFT uses the same trainer and training pool but learns from resolved trajectories of the strong agent instead of the student's rollouts (Table 11). SFT maximizes the likelihood of reference turns; GRPO applies a clipped policy gradient with group-normalized outcome rewards; OPSD instantiates Equation (5) with one teacher whose z is the reference patch; and DuoOPSD instantiates it with the two diagnosis-conditioned teachers. OPD-V (Bi et al., 2026) targets visual question answering: its teachers see the input image zoomed to the relevant region or masked, which presupposes an informative image, whereas many of our

Table 8: **Statistics of the synthetic task pool** generated from the development seeds. A task is valid when it builds, differs visibly and stably from the correct execution, and passes the behavioral check under the inverse mutation. Difficulty is defined relative to the strong agent and the untrained student under matched tools and budgets.

Quantity	Count
Development seeds with an approved sandbox	98
Seeds with at least one valid synthetic task	93
Valid synthetic tasks	513
resolved by the strong agent	468
resolved by the untrained student	234
resolved by the strong agent but not the student (hard subset)	250
Training pool: 79 seeds and their 386 strong-solved synthetic tasks	465
Validation seeds, held out with all their synthetic tasks	19

Table 9: **Synthetic tasks by repository.** Hard: resolved by the strong agent but not by the untrained student. In pool: synthetic tasks of training seeds that the strong agent resolves. Every repository contributes hard tasks, so the pool is not dominated by one codebase.

Repository	Seeds	Tasks	Strong	Student	Hard	In pool
Automattic/wp-calypso	33	180	161	108	61	126
chartjs/Chart.js	21	115	103	42	64	82
processing/p5.js	15	87	83	33	53	72
markedjs/marked	13	75	75	37	38	66
diegomura/react-pdf	11	56	46	14	34	40
Total	93	513	468	234	250	386

tasks come without a valid one (e.g., a completely blank screenshot), so it is not applicable to our setting.

Implementation of DuoOPSD. The divergence in Equation (5) is estimated over the student's top-k tokens plus a tail bucket, with an importance-ratio clip against the rollout policy since a batch is collected before it is consumed. The teachers use equal weights. Training proceeds in rounds: we roll out the current student on a batch of tasks, train on the batch, and repeat. When the agent compacts its context during a run, the trajectory is split at the compaction and each segment is trained as its own sequence, so verification turns after a compaction are not lost.

Hyperparameters. Table 12 lists the training configuration shared by all methods. All methods use the same learning rate and number of optimizer steps, and the on-policy methods also the same rollouts, so that they differ only in the training signal.

H Ceiling analysis of privileged information

Table 13 reports what the untrained student (Qwen3.5-4B under DuoAgent) resolves on the development split when it is given, at inference time, a piece of information that is otherwise available only to a teacher: nothing, the correct execution trace τ_{gold} mounted read-only next to the buggy trace, or the two expert diagnoses written from the aligned traces. For each configuration the table lists the number of resolved instances and the fraction of the student's edits that touch a file modified by the reference repair. The strong agent used to write the diagnoses is included as a reference.

I Prompt templates

This section lists the prompt templates of DuoAgent and of the data pipeline. Placeholders are in braces; text in square brackets is abridged. The same templates are used for every model.

Table 10: **Synthetic tasks by mutation strategy**, with the number that falls into the hard subset. The strategies cover both visible defects (layout, visibility, positioning) and behavioral ones (option handling, state updates, interaction conditions).

Strategy	Tasks	Hard
Option handling	99	44
Layout	62	33
Interaction condition	60	28
Text content	60	19
State update	57	34
Visibility	57	22
Drawing math	47	28
Positioning	41	25
Overflow	23	15
Event handling	7	2
Total	513	250

Table 11: **Training methods compared.** All methods use the same harness and trainer, and differ in the privileged information z and the per-token loss; GRPO, OPSD, and DuoOPSD use the same student rollouts, while SFT uses resolved trajectories of the strong agent; $q(\cdot \mid h_t, z, y_{<j})$ denotes the EMA teacher with extra context z.

Method	Privileged information z	Per-token loss on y_t
SFT	reference turns	$-\log p_\theta(y_j)$
GRPO	outcome reward r	clipped policy gradient with group-normalized advantage
OPSD	reference patch	$D_{\mathrm{JS}}(p_\theta, q_{\mathrm{patch}})$
DuoOPSD	visual + structural diagnoses	$\frac{1}{2}D_{\mathrm{JS}}(p_\theta, q_{\mathrm{vis}}) + \frac{1}{2}D_{\mathrm{JS}}(p_\theta, q_{\mathrm{struct}})$

I.1 Reproducer

The task prompt of the reproducer, which builds the sandbox B and the action sequence A of an instance (development-split variant with both checks; on the test split the C_{fix} step and its failure analysis are replaced by the instruction that C_{repro} is the only check). Environment-specific paragraphs are abridged in brackets.

```
# Build the visual sandbox (browser runner + reproduction actions) for `{instance_id}`

You are building the Replay browser harness for one SWE-bench Multimodal GUI bug. Your
deliverables go in `{sandbox_dir}` (absolute path); the dumps, traces, check verdicts you
read and the status you write live in `{out_dir}`. Everything else on this machine is off
limits: do not write anywhere else, and do not save memory notes or recipes for later
sessions -- put everything worth keeping in `status.json`.

## The bug
Problem statement (also in `{out_dir}/agent/instance_public.json`):

{problem_statement}

Reporter's assets: `{problem_assets}` -- the issue's images ({images}) and, when the issue
linked a codesandbox / jsfiddle /codepen / stackblitz, the archived code of that
reproduction. Look at all of it first: when a reproduction is archived, the demo is that
reproduction, ported to this app and pointed at the local library; the images show what the
page must look like.

## What exists
[... the execution image of the instance (repository at the base commit with its
dependencies installed), the Dockerfile of the sandbox image, the default
`build_library.sh` / `build_sandbox.sh`, an empty `app/`, same-repository sandboxes that
already pass both gates as examples, and the reproducer skills, read-only ...]

## Deliverables (1-4 inside `{sandbox_dir}` -- these are committed to git; 5 inside
`{out_dir}`)
1. `app/index.html`, `app/main.jsx` (or `main.js`), `app/package.json`, `app/vite.config.js`
(+ any helper files): a minimal Vite app that imports the library FROM THE LOCAL PACKAGE
and renders the exact component/props/data/state in which the reported bug appears (match
the reporter's screenshots).
2. `build_library.sh`: builds the library under test (see above).
```

Table 12: **Training configuration**, shared by all methods unless noted. Every method makes 64 optimizer steps of 16 trajectories.

Setting	Value
Rounds; tasks per round; samples per task	4; 64; 4
Optimizer steps per round; trajectories per step	16; 16
Optimizer; learning rate; gradient-norm clip	8-bit AdamW; 2×10^{-6}; 1.0
EMA teacher update	$\bar{\theta} \leftarrow 0.95\,\bar{\theta} + 0.05\,\theta$ after each step
Divergence (distillation methods)	Jensen–Shannon, top-64 tokens plus a tail bucket
Importance-ratio clip (distillation methods)	2.0
GRPO clip ratio; group size	0.2; 4 samples per task
SFT	390 resolved expert trajectories, 64 steps
Packed sequence length	49,152 tokens (distillation), 65,536 (SFT, GRPO)
Rollouts	vLLM, temperature 1, 64k context, 1,800 s per run
Hardware (Qwen3.5-4B)	4×48 GB GPUs; about 5.4 h of rollouts and 45 min of training per round

Table 13: **Ceiling analysis on the development split.** Each row provides the untrained student (Qwen3.5-4B under DuoAgent) with a different piece of training-only information at inference time; 98 instances. The raw correct trace does not help the student, whereas expert diagnoses written from it raise both the resolution rate and the share of edits in the files of the reference repair.

Information provided to the student	Resolved	Edits in gold files
None	21	75%
Correct execution trace τ_{gold}, mounted read-only	18	71%
Expert diagnoses $z_{\text{vis}}, z_{\text{struct}}$	26	90%
Strong agent (GPT-6 Astra, 200k context)	47	–

```
3. `build_sandbox.sh`: installs the app's deps and links the local package (see above).
4. `actions.json`: the fixed interaction sequence that reaches the buggy state (EMPTY list
when the bug is visible on load). Schema:
   ```json
   {"version": 1, "actions": [ACTION, ...], "expectation": "<one sentence: what the BUGGY
   page shows after these actions>"}
   ACTION = {"type":"click"|"dblclick"|"hover","target":T,"note":"why"} |
   {"type":"fill","target":T,"value":"..."}
          | {"type":"select","target":T,"value":"<option label>"} |
          {"type":"press","key":"Enter","target":T|null}
          | {"type":"scroll","target":T|null,"dx":0,"dy":300} | {"type":"wait","ms":500}
          | {"type":"back"} | {"type":"forward"}      (real browser history navigation; key
          shortcuts do not navigate)
   T = {"by":"role","role":"button","name":"Submit","nth":0} |
   {"by":"text","text":"...","nth":0}
     | {"by":"testid","testid":"..."} | {"by":"css","css":"#id ..."}      (prefer role,
     then text)
   ```
   At most 8 actions; hidden inputs may be targeted by role. Use `wait` conservatively: the
   runner already settles the page and waits for a stable screenshot after every action, so
   add a `wait` (`ms <= 3000`) only when a transition between actions cannot be observed
   without it. Each action, including `wait`, adds a step to the trace.
5. `{out_dir}/status.json` (last step, see below).

Your loop (run from anywhere; these commands do the container work for you)
1. `python3 -m sandbox wip-build {instance_id}` -- builds `{wip_image}` from your files
(log tail printed; full log `{out_dir}/agent/wip_build.log`).
2. `python3 -m sandbox dump {instance_id}` -- serves the app in the wip image and writes
`{out_dir}/element_dump.json` (interactive elements with roles/names/bboxes, accessibility
tree, console errors on load). A blank page or console errors here mean the demo itself is
broken.
3. Write `actions.json`, then `python3 -m sandbox trace {instance_id}` -- replays it on the
buggy build and writes `{out_dir}/trace_bug/` (`index.json` + per step `step_N.png`
full-page screenshot and `step_N.json` evidence). Look at the screenshots.
4. `python3 -m sandbox repro-check {instance_id}` -- the first gate, **C_repro**: checks
whether `trace_bug/` shows the reported defect. Opens `{out_dir}/attempts/attempt_N/` with a
copy of your `actions.json`, `trace_bug/`, `verdict_repro.json` (`passed`, `observed`,
`reasoning`) and the raw check reply; the latest record is `{out_dir}/gates.json` -> `repro`
(`ok`, `verdict`, `attempt_dir`). Each attempt folder is one version of (demo, actions):
re-checking the same version overwrites it, changing the demo or the actions opens the next
one, so you can compare versions. If it fails, read the reasoning and fix the demo and/or
the actions.
```

```
5. `python3 -m sandbox fix-check {instance_id}` -- the second gate, **C_fix**. In a
throwaway container it applies the ground-truth fix (hidden from you), rebuilds the
library, relinks the demo, replays YOUR `actions.json` and records `{out_dir}/trace_gold/`
(the fixed page, same steps as `trace_bug/`). The gate passes when the gold trace differs
from the bug trace at some step -- in the screenshot or in the browser state (DOM outline,
accessibility tree, element boxes/styles, console errors, action outcomes) -- i.e. the fix
is observable through your demo and actions. Results: `verdict_fix.json` in the attempt and
`{out_dir}/gates.json` -> `fix` (`ok`, `verdict.changed_fields`, `verdict.explanation`).
Repeat until BOTH `gates.json.repro.ok` and `gates.json.fix.ok` are true for the SAME
`actions.json`. The sandbox is only usable when both gates pass.

## When a gate fails -- it is either the sandbox (demo) or the actions
Both gates run the same demo with the same actions, so read them together:
- **C_repro fails, C_fix fails (gold trace identical to bug trace)**: the demo most likely
does not exercise the affected code at all (wrong component/props/data, library not
actually imported from `/testbed`, a prebuilt bundle that the relink does not refresh, a
blank page / console error). Fix the demo first; check `element_dump.json`.
- **C_repro passes, C_fix fails (identical traces)**: the actions/expectation stop before
the state where the fix would show (the defect is visible, but the corrected behaviour
never reaches the screenshot or the browser state -- add the step that reveals it), or the
demo sidesteps the fixed code path (hard-coded values, a different entry point, a caught
error). Compare `trace_bug/` and `trace_gold/` step by step.
- **C_repro fails, C_fix passes**: the actions reach a state where the buggy build already
looks correct or ambiguous (timing: the bug may be a transition, so screenshots are taken
after the page settles; or the `expectation` describes something the screenshot cannot
show). Change the actions, or make the component's own buggy output visible (e.g. render
the frames it painted) -- never by adding text about it.
- The check only sees the screenshots and the runtime evidence (console errors, element
boxes/styles, accessibility tree). If the defect is not visible in those, make the
component show it -- never fake it with markup and never narrate it with text.

## Rules
- The page is a reproduction, not an explanation. Render the component in the reported
state, as the reporter's page does, and nothing else: no captions, headings, instructions,
status lines, verdicts or echoed state ("expected ...", "valid: false", "comment was
moved", "URL sort: ...", "Current value: ..."). The check compares screenshots and browser
state, and console errors / DOM / accessibility state are recorded as evidence
automatically; text that describes the bug, and above all text that would differ between
the buggy and the fixed build, leaks the verdict and is rejected. Rendering more of the
component's OWN output (another instance, the frames it painted) is fine; prose about it is
not.
- Do not modify the repository or its tests (`/testbed` is read-only for you); the bug must
come from the library as built, never from hard-coded markup imitating it. You do not have
the fix for this bug and must not try to find, guess or apply one. Never read the
instance's `patch`/`test_patch`, the task repo's files, or the repository's git history
past the base commit.
- Do not embed the issue screenshots in the page. No network data; deterministic content
only (no random data, no timers unless the bug needs them).
- Keep the demo minimal: one page, the affected component, the state from the report.
- If the issue cannot be reproduced in headless Linux Chromium (OS-specific, needs a
device, etc.), stop and say so in `status.json` with `reproducible: false`.

## Final step -- write `{out_dir}/status.json` exactly like this and stop
```json
{"ok": true|false, "reproducible": true|false, "n_actions": <int>, "attempts": <int>,
 "repro_ok": true|false, "fix_ok": true|false|null,
 "reason": "<only when ok is false>", "notes": "<what the demo shows, how the actions reach
 the bug, what the fixed page shows>"}
```
```

I.2 REPRODUCTION CHECK C_{repro}

The system prompt with which the reproducer checks its trace, followed by the layout of the user message it receives. The check passes when `reproduced` is true with high or medium confidence.

```
You validate that a recorded browser trace actually exhibits a reported GUI bug.

You are given: an issue report (text, possibly with the reporter's screenshots), a fixed
action sequence that was executed on the UNPATCHED buggy program, and the resulting trace
(screenshots per step plus runtime evidence). The trace comes from a minimal demo app that
mounts the affected component -- it does not need to look like the reporter's app; only the
defective BEHAVIOR must be present.

Trace screenshots are FULL-PAGE captures (each is labelled with its pixel size and the
browser viewport). A capture much taller than the viewport means the page content
overflows/scrolls -- for layout bugs about fixed heights, sticky headers, or internal
scrolling, the capture height itself is often the decisive evidence.
```

```
Answer whether the trace clearly shows the defect described in the report.
- reproduced=true only if the defect described in the report is visible in the trace (in a
screenshot, or unambiguously in the runtime evidence, e.g. a crash/console error the report
describes).
- If the trace looks like the EXPECTED (fixed) behavior, or the relevant UI state is never
reached (wrong element clicked, nothing rendered), answer reproduced=false and explain what
is missing.

Reply with ONLY a JSON object:
{"reproduced": true|false, "confidence": "high"|"medium"|"low", "observed": "<one or two
sentences: what the trace actually shows>", "reasoning": "<why this does or does not
exhibit the reported bug>"}

---------- user message ----------
## Issue report
{problem_statement}

[Issue-report image 1 -- the reporter's own screenshot/recording of the bug]  <image>
... (up to three report images)

## Fixed action sequence executed on the buggy program
{actions, one per line}

Expected bug appearance after these actions: {expectation}

[buggy-program trace, step 0 -- FULL-PAGE capture, {W}x{H} px; browser viewport 1280x720
px. A capture taller than the viewport means the page itself scrolls/overflows.]  <image>
... (up to eight steps, always the first and the last)

## Runtime evidence (buggy-program trace)
step 0: initial page load
step 1: {action as JSON} -> ok | FAILED: {error}
  url: {url, when it changes}
  {console errors}
```

I.3 BUG INJECTION

The system prompt that proposes mutations of a seed program for synthetic task generation; {strategies} is the list of the ten strategies of Table 10.

```
You inject realistic bugs into a JavaScript/TypeScript library for a benchmark of GUI bug
repair.

You get the CORRECT version of the source files behind a small demo page (the demo mounts
the library component and replays a fixed interaction), the demo's entry file, the original
issue that this code once fixed (for context only), and a summary of the browser state the
demo shows when everything is correct.

Propose {k} DIFFERENT localized bugs, each a small mutation of the correct source (change
1-15 lines in one file, at most two files), such that the demo's rendered page or its
interaction visibly changes: layout or geometry, overflow or clipping, positioning, an
interaction condition (wrong guard, inverted check), a state update that does not happen or
happens twice, drawing math (sign, off-by-one, wrong clamp, wrong scale), an ignored or
misread option, event handling, wrong text, or wrong visibility. Each bug must be plausible
(something a developer could write), must NOT be a syntax error or a crash on load (the
page must still render), and must affect what the demo shows or does. Prefer bugs that need
understanding of the code to fix, not typos in constants.

Reply with ONLY a JSON object: {"mutations": [{"strategy": one of {strategies}, "title":
"<short>", "symptom": "<what the demo will show>", "patch": "<unified diff against the
given files, paths relative to the repository root, `--- a/path` / `+++ b/path` headers,
valid for `git apply`>"}, ...]}
```

I.4 ISSUE WRITING

The system prompt that writes the issue report of a validated synthetic task from the buggy and correct traces.

```
You write GitHub issue reports for a JavaScript library, in the voice of a user who hit a
bug in the small demo below.
```

```
You are given (as the generator, not the reporter): the mutation that introduced the bug,
the demo's entry file, the interaction the demo performs, the browser state difference
between the buggy and the correct behaviour, screenshots of the BUGGY page at the affected
steps and, for your understanding, of the correct page. The reporter has only seen the
buggy page and knows what they expected.

Write a realistic report: a title, then a body with the setup (which component/options the
demo uses, in prose or a short snippet of USER code, never the library's internals), the
steps, what happens, what was expected, and a mention of the attached screenshot(s). Do not
quote or paraphrase the mutation diff, do not name the internal file or function, do not
mention that the bug was injected. Keep it to 80-200 words plus the snippet.

Reply with ONLY a JSON object: {"title": "...", "body": "..."}. Refer to the screenshots as
{images}.
```