SWE-Protégé: Small Language Models for Software Engineering Through Learned Expert Collaboration

Patrick Tser Jern Kon[2*] **Archana Pradeep**[1] **Ang Chen**[2] **Alexander P. Ellis**[1]
Warren Hunt[1] **Zijian Wang**[1] **John Yang**[3] **Samuel Thompson**[1]

[1]Meta [2]University of Michigan [3]Stanford University
patkon@umich.edu, samuelt@meta.com

Abstract

Small language models (SLMs) offer compelling advantages in cost, latency, and adaptability, but have lagged behind larger models on long-horizon software engineering tasks such as SWE-bench, where they suffer from pervasive action looping and low resolution rates. We introduce SWE-Protégé, a post-training framework that reframes software repair as an expert–protégé collaboration problem. In SWE-Protégé, an SLM remains the primary decision-maker while learning to selectively seek guidance from a strong expert model, recognize stalled states, and follow through on expert feedback. Our approach combines supervised fine-tuning on expert-augmented trajectories with agentic reinforcement learning that explicitly discourages degenerative looping and unproductive expert collaboration. We lightly post-train Qwen2.5-Coder-7B-Instruct to achieve 42.4% Pass@1 on SWE-bench Verified with SWE-agent and Claude Opus 4.1 as the expert, a +25.4% improvement over the prior SLM state of the art, while using expert assistance sparsely ($\approx$4 calls per task and 11% of total tokens). With Claude Sonnet 4.5 as the expert, it reaches 41.0% at a median expert API cost of \$0.15 per task, 8.2$\times$ lower than running the expert alone.

1 Introduction

Software engineering (SE) tasks, most notably SWE-bench, have emerged as an important proving ground for language model (LM) agents, with recent advances driven primarily by large open-source and proprietary LMs (e.g., Claude Sonnet). Yet this dominance stands in tension with a parallel trend (Abouelenin et al., 2025; Bakouch et al., 2025) toward small language models (SLMs; LMs with $\leq$10B parameters) (Belcak et al., 2025), which offer compelling advantages in cost, latency, and adaptability, but have so far remained largely absent from recent SWE-bench progress.

Indeed, this is unsurprising given the long-horizon/context nature of SE, which requires sustained progress over many turns (e.g., performing edits, recovering from errors) across complex repositories, rather than isolated single-turn code synthesis (e.g., HumanEval-style problems) where SLMs already perform reasonably well (Hui et al., 2024). Even recent data-scaling approaches that generate large volumes of high-quality agent trajectories for training (Wei et al., 2025; Pan et al., 2024), including the state-of-the-art (SOTA) SWE-smith (Yang et al., 2025b), remain insufficient for SLMs. Prior work shows that SLM agents post-trained with these methods still suffer from stalled progress, which commonly manifests as degenerative action loops (i.e., SLM repeatedly calling the same base commands), and correspondingly poor resolution rates ($\approx$10% Pass@1) on SWE-bench Verified (Pan et al., 2024; Yang et al., 2025b). Moreover, our analysis shows that progressively scaling SWE-smith training can even induce performance regressions.

*Work done at Meta.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

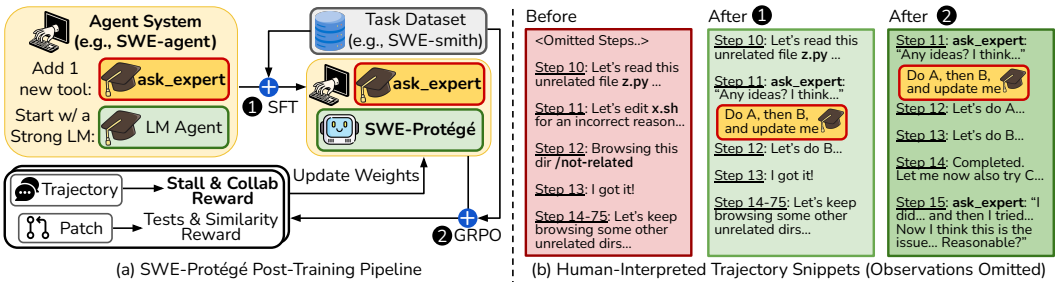


Figure 1: (a) Our two-phase pipeline yields SWE-Protégé-7B: Phase (1) uses SFT on expert-augmented trajectories; Phase (2) applies GRPO with trajectory-level rewards. (b) Paraphrased trajectories. *Before:* the SOTA SLM baseline (SWE-agent-LM-7B) fails to make reliable forward progress and degenerates into unproductive exploration. *After Phase (1):* our SLM can invoke the expert, but follow-through on guidance is inconsistent and it often relapses into stalling. *After Phase (2):* it learns to escalate when stalled, follow through on guidance, and report back, exhibiting multi-turn pair-programming behavior.

Motivated by these limitations, we explore a complementary paradigm in SWE-Protégé that reframes the problem as one of expert–protégé collaboration, analogous to pair programming, leveraging the respective strengths of both components. We study one effective instantiation of this paradigm in which the protégé (SLM) retains responsibility for the primary workflow and decision-making (handling most routine reasoning and interaction) with speed and low cost, while *learning* to selectively seek guidance from a strong expert (e.g., Claude Sonnet) that provides expensive but high-value intuition/understanding only when needed.

Specifically, SWE-Protégé post-trains an SLM to act as an autonomous pair programmer that can (i) recognize stalled states where it's not making progress, (ii) invoke an expert appropriately, and (iii) execute and follow through on expert guidance over multiple turns. We implement this in two phases (Fig. 1). First, we distill into the SLM the mechanics and semantics of expert interaction via supervised fine-tuning on expert-augmented trajectories generated by a strong LM that is also repurposed to serve as the expert. Second, we run on-policy agentic reinforcement learning with shaped rewards that target when to escalate and how to collaborate over multiple turns: the expert is repurposed as an in-trajectory judge (providing process-level supervision), which first explicitly discourages degenerative stalling, and then penalizes low-quality collaboration. Throughout, the SLM remains the primary decision-maker, treating the expert as an interactive collaborator rather than an oracle.

On SWE-Bench Verified, our SWE-Protégé prototype lightly post-trains Qwen2.5-Coder-7B-Instruct to achieve 42.4% accuracy with Claude Opus 4.1 as the expert (+25.4% over the prior state of the art, SWE-agent-LM-7B by SWE-smith). The model invokes expert assistance sparsely (only about 4 calls per task) with expert tokens comprising only $\sim$11% of total tokens, and with Sonnet 4.5 as the expert it reaches 41.0% at a median expert API cost of \$0.15 per task, 8.2$\times$ lower than direct expert execution (\$1.24), while preserving the SLM's role as the primary decision-maker. All results use the standard SWE-agent setup (75-step limit, no test-time sampling). Notably, these gains are achieved with minimal data and compute: we use the same SFT task dataset size ($\approx$5K trajectories) as SWE-smith, and perform a short single-node RL phase with rollouts generated from only 100 tasks for 160 steps. We envision a SWE-Protégé-like paradigm shining in settings where, for instance, access to frontier-model tokens is a binding constraint: e.g., in application services (Cursor, 2026; OpenClaw, 2026) constrained by cost, quotas, or rate-limits.

2 SWE-Protégé Post-training Pipeline

Motivated by the aforementioned limitations, we introduce a post-training pipeline that enables an SLM to engage in *autonomous, selective, and multi-turn collaboration* with an expert model during long-horizon tasks. SWE-Protégé trains the SLM itself to (i) recognize stalled states, (ii) initiate escalation when appropriate, and (iii) follow through on expert guidance in a structured, iterative

Model	System	Train Size	Accuracy (%)	Reference
Closed Weight Models				
Claude 3.5 Sonnet	OpenHands	-	53.0	Anthropic (2024)
Claude 3.7 Sonnet	SWE-agent	-	58.2	Anthropic (2025)
Claude 4.5 Sonnet	SWE-agent	-	**72**	-
Llama3-SWE-RL-70B	Agentless	11M	41.0	Wei et al. (2025)
Open Weight Models (Post-Trained)				
SWE-fixer-72B	SWE-Fixer	110k	32.8	Xie et al. (2025)
SWE-gym-32B	OpenHands	491	20.6	Pan et al. (2024)
R2E-Gym-32B	OpenHands	3.3k	34.4	Jain et al. (2025)
SWE-agent-LM-32B	SWE-agent	5k	**40.2**	Yang et al. (2025b)
Open Weight Small Language Models ($\leq$10B parameters (Belcak et al., 2025))				
SWE-gym-7B	OpenHands	491	10.6	Pan et al. (2024)
SWE-agent-LM-7B	SWE-agent	2.4k	17	Yang et al. (2025b)
Lingma-SWE-GPT-7B	SWE-SynInfer	-	18.2	Ma et al. (2024)
SWE-Protégé-7B (Sonnet 3.7)	SWE-agent	4.9k	30.8 ± 0.8	
SWE-Protégé-7B (Sonnet 4.5)	SWE-agent	4.9k	41.0 ± 0.5	
SWE-Protégé-7B (Opus 4.1)	SWE-agent	4.9k	$\mathbf{42.4 \pm 0.7}$	

Table 1: Resolve rates (Pass@1) on SWE-bench Verified for existing solutions compared to SWE-Protégé. We exclude open-weight models trained end-to-end from scratch (Copet et al., 2025), and systems that rely on verifiers or test-time sampling. The 17% result corresponds to our strongest internally reproduced SWE-agent-LM-7B baseline. Errors denote standard error over three runs. Parentheses denote the frontier expert used for collaboration at inference.

manner—treating the expert as an interactive collaborator rather than a one-shot oracle, analogous to pair programming initiated by the junior partner.

Problem Setting. We consider an agentic coding environment in which an SLM is embedded within an agent scaffold (e.g., SWE-agent (Yang et al., 2024)) and prompted over multiple turns to solve a task (e.g., from SWE-bench). At each turn, the SLM selects an action by invoking one of the agent's tools—such as issuing shell commands (e.g., `grep`) or editing files—which are executed in a task-specific environment (e.g., a cloned GitHub repository). Action selection is usually conditioned on a constructed representation of the interaction history (e.g., prior actions and tool outputs). Upon termination, task success is assessed via external verification (e.g., unit tests in SWE-bench-style setups).

2.1 Agent System Setup

Agent Tool and Expert Interfaces. SWE-Protégé requires only the addition of a single expert-invocation tool to an existing agent system. Let $s \in \mathcal{S}$ denote the full agent state, including the interaction history, tool outputs, and system prompts. The SLM defines a policy $\pi_\theta(a \mid s)$ over actions $a \in \mathcal{A}$, where actions correspond to tool invocations. We extend the action space to $\mathcal{A}' = \mathcal{A} \cup \{\texttt{ask_expert}\}$ where `ask_expert` is a structured tool call in which the SLM formulates a query (e.g., a question). The expert processes this query and returns textual guidance, which is appended to the agent state s and subsequently used for continued decision-making. In practice, all actions—including expert calls—are generated as text under a unified model distribution $p_\theta(y \mid s)$, and expert invocation is detected by the agent controller via a reserved tool-call format.

The expert model is invoked with the agent's query together with a system prompt (App. D). To keep escalation lightweight, the expert may be given only a partial view of s; specifically, it receives a compact summary $\tilde{s}$ consisting of the most recent K interaction turns and relevant tool outputs. In

contrast, the agent always acts from the complete state s. We show later (§4) that this asymmetric information flow keeps expert interactions focused and token-efficient.

Collaboration Mechanisms. Expert invocation occurs through two mechanisms. First, the SLM may autonomously emit `ask_expert` as part of its policy. This learned invocation behavior is the primary focus of SWE-Protégé; while training or adapting (e.g., prompt optimization (Khattab et al., 2024)) expert models is equally important and will likely yield even stronger performance, we assume that prompting strong frontier models yields sufficiently strong experts for our setting, and defer expert optimization to future work (§5). Second, we support optional rule-based intervention (e.g., keyword triggers) that can forcibly invoke the expert. We currently use this for ablation purposes (§4.1).

2.2 Phase I: Supervised Induction of Expert Usage

The first training phase equips the SLM with an operational understanding of expert interaction. Empirically, we find that SLMs fail to generalize to new tools (even simple ones) without explicit imitation learning. Crucially, this phase addresses *how to interact* with the expert at the level of mechanics and local semantics; e.g., invoking the tool, formulating contextually appropriate queries. It does not determine *when* escalation is appropriate, nor does it enforce correct downstream behavior after advice is given.

Synthetic Trajectory Generation. Training trajectories are generated using a strong code-capable model π_E conditioned on s with augmented action space $\mathcal{A}'$, in the same agent environment used for SLM inference. The model is prompted to invoke the expert tool only when appropriate (App. C). In practice, this yields a natural data mixture in which expert invocation is sparse yet present in most trajectories, alongside a smaller subset with no expert calls, without any manual data mixing. We also found it beneficial to instantiate the expert as the same model π_E, and to additionally grant the expert access to the ground-truth solution (while explicitly instructing it not to reveal the answer verbatim). This produces a substantially larger number of correct trajectories. To test whether this privileged information biases the learned policy, we also trained a P1 model on an equally sized trajectory pool generated without the gold patch, and observed negligible differences in test-time performance ($\pm 0.3\%$ across all expert backends; App. B).[2]

Supervised Fine-Tuning. We perform rejection-sampling supervised fine-tuning (SFT) on the expert-augmented trajectories. Given state–token pairs $\{(s_i, y_i)\}_{i=1}^{N}$, we minimize the standard next-token cross-entropy loss:

$$\mathcal{L}_{\mathrm{SFT}}(\theta) = -\mathbb{E}_{(s_i,y_i)}\left[\log p_\theta(y_i \mid s_i)\right].$$

We introduce no auxiliary losses (e.g., regularizers that incentivize expert calls). Instead, the SLM learns to sparsely invoke the expert in-context, emerging implicitly through imitation of trajectories where expert calls are infrequent.

2.3 Phase II: RL for Expert–SLM Pair Programming

Phase II (full details in App. E) aligns the post-SFT SLM to behave as an autonomous pair programmer: it should (i) *escalate when progress stalls* instead of looping, and (ii) *follow through and report back* on expert guidance across multiple turns. We achieve this with agentic RL and a composite reward that, in addition to standard objectives such as correctness, explicitly models (a) degenerative loops and (b) low-quality collaboration. Starting from the SFT checkpoint, we perform on-policy agentic RL with a GRPO-style objective (details in §3.1). Unlike Phase I, rollouts are the SLM's own, so the judge scores below are computed on the SLM's own stalled states and subsequent behavior, calibrating escalation to the SLM rather than imitating the expert's.

Collaboration Quality. Let τ denote the full agent trajectory (sequence of agent actions and tool responses). During τ, the agent may call the expert N times; for call i, let q_i be the agent query, g_i the expert guidance, and $\tilde{s}_i$ the compact context packet shown to the expert (e.g., containing recent agent messages). We associate each g_i with the subsequent *response segment* $\Delta_i(\tau)$ (the agent's actions and tool outputs after receiving g_i through the next expert call or termination).

[2] At any other point in time, including during evaluation, the expert no longer has access to the gold patch and unit tests and operates solely on $\tilde{s}$. Details of prompting, filtering, and data statistics are provided in App. C.

We repurpose the expert itself as an *in-trajectory judge* (akin to a value model (Yuan et al., 2024)) by defining two bounded functionals:

$$u_i := J_{\text{warrant}}(q_i, \tilde{s}_i) \in [0,1], \qquad f_i := J_{\text{follow}}(g_i, \Delta_i(\tau)) \in [0,1]. \tag{1}$$

where u_i scores whether escalation was warranted (discouraging lazy invocations), and f_i scores whether the agent followed guidance and reported back (enforcing multi-turn collaboration). These scores are computed via a hidden judge call and logged per event. We validated both judges against human labels on 140 warrant and 160 follow events sampled from Phase II rollouts: judge–human agreement is 84% (κ=0.66) and 89% (κ=0.77), respectively, with inter-annotator κ=0.81 (App. E.3).

Reward structure and gating. We use a trajectory-level composite reward inspired by (Huang et al., 2025).

$$R_{\text{total}}(\tau, x) = R_{\text{loop}}(\tau) + w_{\text{follow}}\, R_{\text{follow}}(\tau) \;+\; g_{\text{loop}}(\tau)\, g_{\text{follow}}(\tau)\, R_{\text{other}}(\tau, x), \tag{2}$$

$$R_{\text{other}}(\tau, x) = R_{\text{correct}}(\tau, x) + w_{\text{sim}}\, R_{\text{sim}}(\tau, x) \;+\; w_{\text{expert}}\, R_{\text{expert}}(\tau). \tag{3}$$

Here x denotes the task instance metadata (e.g., gold patch), $R_{\text{loop}} \leq 0$ penalizes degenerative looping (command repetition); R_{expert} shapes deferral quality using $\{u_i\}$ (and anti-spam penalties); and R_{follow} shapes follow-through using $\{f_i\}$. Concretely, we define *degenerative stalling* as a contiguous repetition of normalized tool-action patterns (App. E.2), and *low-quality collaboration* as unwarranted escalation (low u_i), back-to-back expert calls, or failure to follow through and report back on guidance (low f_i). The gates $g_{\text{loop}}, g_{\text{follow}} \in \{0, 0.5, 1\}$ downweight R_{other} under severe looping or failed follow-through, preventing correctness/similarity (derived from (Copet et al., 2025)) from masking pathological interaction behavior.

Reward shaping curriculum. To amplify the reward signal for the desired behaviors (i) and (ii), we adopt a two-stage reward shaping (Ng et al., 1999) schedule: (1) *loop aggressive* shaping, which makes R_{loop} and g_{loop} strongly suboptimal, anchoring stalled progress to expert invocation rather than continued unproductive exploration; and (2) *loop+follow aggressive* shaping, which additionally makes w_{follow} and g_{follow} strongly suboptimal, encouraging the SLM to follow expert guidance to completion and report back, thereby inducing true multi-turn pair programming. We specify the exact schedule (weights, caps, and gate thresholds) in App. E.5. As we show in §4, these behaviors emerge after only a small number of RL updates on modest hardware, yielding further efficiency and behavioral gains, with modest accuracy gains, over §2.2.

3 Experimental Setup

We explore the utility of our approach (§2) under a setting that is closely comparable to SWE-smith, the SOTA at the time of evaluation. Across all experiments, we used a single node equipped with 8 NVIDIA A100/H100 80G GPUs (e.g., AWS `p4de.24xlarge`) for both training and inference.

Agent System. We adopt the off-the-shelf SWE-agent framework exactly as used in SWE-smith, augmented with a single additional tool, `ask_expert`, subject to a 75-step and $2 budget. At each step, SWE-agent prompts the language model to produce a ReAct-style (Yao et al., 2022) *(thought, action)* pair, where actions correspond either to file edits or shell commands within the task environment. Only 6 expert calls are permitted throughout all task rollouts.

Models. Our base model is Qwen-2.5-Coder-Instruct (Hui et al., 2024), primarily the 7B variant, as used in SWE-smith. We additionally report selected ablations with the 32B variant. Inference and rollout generation are performed using vLLM. Expert models are accessed via AWS Bedrock and include Claude Sonnet 3.7, Sonnet 4.5, and Opus 4.1. Experts receive only the most recent 5 messages as context.

Evaluation. We evaluate on SWE-bench Verified (Chowdhury et al., 2024), a human-vetted 500-instance SWE-bench (Jimenez et al., 2023) subset (drawn from 12 real-world GitHub repositories) to reduce ambiguity in problem statements and improve evaluation reliability. We report %Resolved (Pass@1), i.e., the fraction of tasks solved by a single rollout per instance. We do not use multi-attempt sampling, majority voting, or other test-time scaling. To assess potential data contamination of our approach, we additionally evaluate on a held-out subset of the SWE-smith task dataset comprising much newer tasks that is disjoint from all training data (details in §4.1).

3.1 Training Setup

SFT Details. We perform rejection-sampling full SFT on the Qwen-2.5-Coder-Instruct-7B base model with Torchtune (meta-pytorch, 2026) using the same SWE-smith task dataset (SWE-bench, 2026a), for a fair comparison. Specifically, we train on trajectories generated by Claude Sonnet 3.7 using our procedure (§2.2; which yielded 38% more usable trajectories than when the ground-truth patch was not provided) by sampling until we obtained ≈4.8K resolved tasks. We use a maximum sequence length of 32,762 and batch size 32. We also experimented with LoRA and QLoRA under the same data and evaluation protocol; both underperformed full SFT on SWE-bench Verified Pass@1. Thus, we report full SFT results throughout.

RL Details. Starting from the SFT checkpoint, we apply an on-policy RL stage using GRPO (Shao et al., 2024), mirroring recent software-repair RL work (Wei et al., 2025; Copet et al., 2025) to keep our results interpretable against that line of work (we do not claim GRPO is uniquely suited to our setting). While stronger variants exist (Yu et al., 2025; Khatri et al., 2025), we only use a minimal modification to standard GRPO: asymmetric clipping (following (Yu et al., 2025)) with separate upper/lower clip thresholds, setting $\epsilon_{\text{high}} = 0.28$ and $\epsilon_{\text{low}} = 0.20$ to reduce premature entropy collapse. Concretely, for a prompt x, we sample G completions $\{a_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$ and optimize

$$J(\theta) = \frac{1}{G}\sum_{i=1}^{G} \min\Big(r_i(\theta)A_i,\ \text{clip}\big(r_i(\theta),\ 1-\epsilon_{\text{low}},\ 1+\epsilon_{\text{high}}\big)A_i\Big) - \beta\,\text{KL}\big(\pi_\theta(\cdot \mid x)\,\|\,\pi_{\text{ref}}(\cdot \mid x)\big).$$

where $r_i(\theta) = \pi_\theta(a_i \mid x)/\pi_{\theta_{\text{old}}}(a_i \mid x)$. The advantage is computed via group normalization,

$$A_i = \frac{r_i^{\text{env}} - \text{mean}(\{r_j^{\text{env}}\}_{j=1}^{G})}{\text{std}(\{r_j^{\text{env}}\}_{j=1}^{G})},$$

with r_i^{env} denoting the scalar rollout reward. We train on the default 100-task SWE-Gym subset (SumanthRH, 2026) bundled with SkyRL, with no dataset mixing; we did not ablate its selection or size (§5). We used 6 rollouts per prompt, batch size 16, and 160 total steps. This RL phase is intentionally short and data-light, and is primarily a behavior- and cost-shaping stage, targeting expert usage and trajectory hygiene rather than broad capability or accuracy gains.

Our reward shaping schedule is (details in App. E): (i) *loop aggressive* shaping for steps 1–80, where we set $(k_1, k_2, \lambda_{\text{loop}}, c_{\text{loop}}) = (15, 8, 0.5, -10)$ and only activate the g_{loop} gate. (ii) *loop+follow aggressive* shaping for steps 81–160, where we retain the same loop penalty, set $(w_{\text{expert}}, w_{\text{follow}}) = (0.3, 2.0)$ and $(\tau_{\text{follow}}, p_{\text{follow-low}}) = (0.5, -2.0)$, activate the g_{follow} gate, and impose a hard -10 penalty when no expert call is made.

Our RL system builds on Ray-based SkyRL (NovaSky-AI, 2026). We integrate SWE-agent by implementing a SkyRL *generator* that, for each sampled task, launches a Ray worker to run a full SWE-agent episode in a SWE-ReX Docker runtime. To improve stability and throughput, we (i) cap concurrent SWE-agent/Docker startups and introduce I/O backoff to mitigate stalls and transient failures, (ii) add trajectory-level checkpointing of completed rollouts to support mid-run reward-shaping updates, and (iii) pipeline inference with multiple in-flight batches to maintain high vLLM utilization despite stragglers.

4 Results

Table. 1 summarizes our main results on SWE-bench Verified. To ensure a fair comparison to SWE-smith (the SOTA open-weight scaling baseline) we restrict expert backends to the same evaluation lineage, using Sonnet 3.7 as in SWE-smith and additionally considering newer successors (Sonnet 4.5 and Opus 4.1), even though stronger options exist today. SWE-Protégé-7B achieves up to 42.4% accuracy with Opus 4.1 as the expert, exceeding the best prior open-weight scaling baseline SWE-agent-LM-32B (40.2%) by +2.2%, while substantially improving over the corresponding SLM baseline SWE-agent-LM-7B (+25.4%) and the strongest prior 7B-class SWE system Lingma-SWE-GPT-7B (+24.2%). This comparison is asymmetric, however: SWE-Protégé is a hybrid system that consults a frontier expert, and without expert access SWE-Protégé-7B performs on par with SWE-agent-LM-7B. We therefore read this result not as a 7B model matching a 32B one, but as evidence that learned selective collaboration is an attractive use of a fixed frontier-token budget.

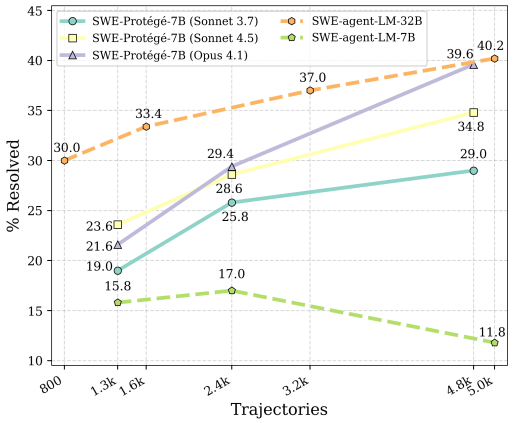


Figure 2: SWE-Protégé-7B exhibits consistent gains via SFT with increased SWE-smith training data.

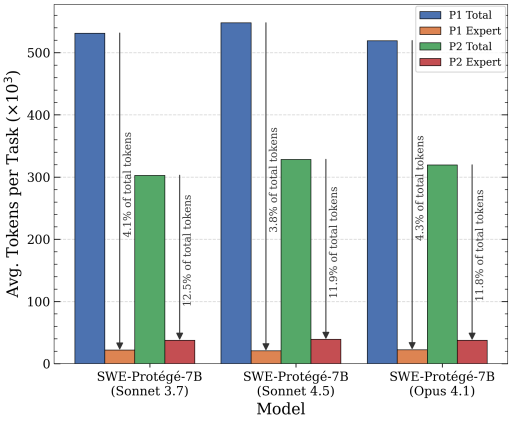


Figure 3: Expert tokens remain consistently low, while total token usage is substantially reduced after Phase II.

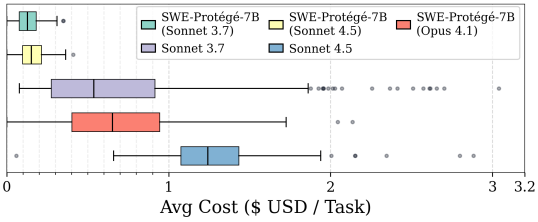


Figure 4: SWE-Protégé substantially reduces per-task expert API cost relative to direct expert execution (the expert solves the entire task). Details in §4.1.

Model	Pass@1 (%)	Δ
SWE-Protégé-7B (Sonnet 3.7)	30.8	+1.8
SWE-Protégé-7B (Sonnet 4.5)	41.0	**+6.2**
SWE-Protégé-7B (Opus 4.1)	**42.4**	+2.8

Table 2: Phase II RL is primarily a behavior- and cost-shaping stage; its accuracy gains on SWE-bench Verified are modest and vary by expert. Δ denotes absolute improvement over the corresponding SFT checkpoint.

Finally, Phase II RL (Table 2) yields modest accuracy gains across expert backends (+3.6% on average); its primary effect is on behavior and cost: it reduces total token usage by ≈40%, collapses degenerative loops, and sharply reduces runtime-limit aborts, without increasing expert calls (§4.1).

Performance Scales with More Training Data. Fig. 2 plots accuracy as a function of the number of training trajectories. For each point, we subsample trajectories uniformly at random from our ≈4.8k source task pool and train from the same base initialization. We also independently re-trained/evaluated the SWE-agent-LM-7B scaling curve using the *original* trajectory pool (SWE-bench, 2026b) and matching sampling protocol for each budget, and observed accuracy consistent with SWE-smith reports (e.g., ≈15–17% at the 2k–2.4k regime). We focus on the SWE-smith trajectory source because it was shown to provide stronger improvements than alternative sources (e.g., SWE-Gym (Pan et al., 2024)) at comparable dataset sizes. For SWE-Protégé-7B, increasing the SFT trajectory budget yields monotonic gains across all three experts: e.g., with Sonnet 3.7, accuracy improves from 19.0% (1.3k) → 29.0% (4.8k), a net gain of +10.0%; and with Sonnet 4.5, 23.6% → 34.8% (+11.2%). In contrast, SWE-agent-LM-7B exhibits a plateau from 2.2% at 100 trajectories to 17.0% at 2.4k, followed by regression at larger budgets (11.8% at 5.0k); only the higher-budget regime is shown in Fig. 2. These trends align with SWE-smith's finding that high-quality trajectories can drive strong performance gains, but extend it by showing that such gains can be realized in SLMs when trajectories are augmented with sparse expert interactions; we note that direct prompting is insufficient to elicit this behavior (we did targeted ablations on SWE-smith's trained 7B and 32B models).

4.1 Ablations on SWE-Protégé

Expert Collaboration is Token-light. Fig. 3 decomposes per-task generation into SWE-Protégé tokens vs. expert tokens. We focus our discussion on Phase II (P2), as the qualitative patterns are

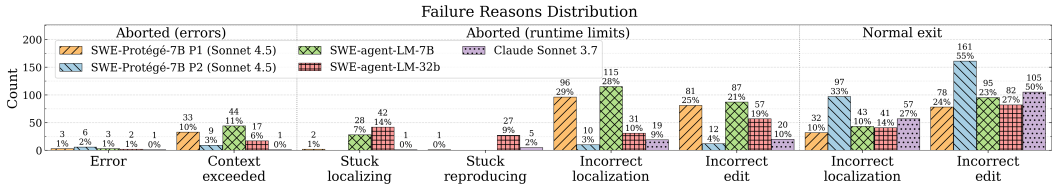


Figure 5: Phase II RL (P2) sharply reduces cost/step limit aborts relative to post-SFT (P1) and SWE-smith baselines.

consistent across phases. Expert tokens account for only a small fraction of each trajectory (e.g., 11.9% with Sonnet 4.5), implying that SWE-Protégé performs the bulk of reasoning, exploration, and patching, while using the expert as a sparse collaborator rather than a primary driver. Expert replies are short (median / p95 / max: 500 / 937 / 1,657 tokens) despite being conditioned on large input contexts (8,885 / 20,716 / 43,031 tokens) consisting of the SLM query and five prior messages (§3). Meanwhile, total tokens per task remain ≈constant across experts ($\sim$ (3–3.2) $\times$ 10^5), suggesting that swapping experts primarily changes *quality* rather than the amount of agent-side work. Phase II reduces total token usage by ≈40%, primarily by eliminating degenerative looping. SWE-Protégé also yields large reductions in expert API cost. From Fig. 4, direct expert execution has median per-task API costs of $0.54 (Sonnet 3.7) and $1.24 (Sonnet 4.5), with outliers reaching $3.04 and $2.88. Under SWE-Protégé, the median expert API cost drops to $0.13/$0.15 for Sonnet 3.7/4.5 (4.2× and 8.2× lower than direct execution), and to $0.65 with Opus 4.1; despite Opus 4.1 being 5× and 4.54× more expensive per-token, it remains cost-comparable to direct Sonnet 3.7 and cheaper than direct Sonnet 4.5. Within a single configuration, Sonnet 4.5 thus yields 41.0% Pass@1 at a median expert API cost of $0.15 per task. These figures count expert API cost only: we do not attach a dollar figure to protégé inference (≈88% of tokens), which runs on self-hosted GPUs whose cost depends heavily on hardware ownership, utilization, and batching. SWE-Protégé targets settings where frontier tokens, metered per call and capped by quotas and rate limits, are the binding constraint. Finally, we separately test Sonnet 3.7 and Sonnet 4.5 on SWE-agent with step limits of 8 and 16, respectively, which is a more generous budget that exceeds both our 6-expert-call cap and average expert cost (by $\sim$1.47× and 1.03×, respectively). We find performance drops sharply to 18.2% and 26%.

SWE-Protégé-7B Solves Tasks Efficiently. Beyond reducing expert token usage, SWE-Protégé also becomes progressively more efficient at the *trajectory level*. As training proceeds, the mean number of steps per task drops from ≈60 at the post-SFT checkpoint to ≈35 after Phase II, while the number of expert calls remains stable at around four (Fig. 8). Fig. 10 further shows that SWE-Protégé achieves step counts comparable to Sonnet 3.7 and SWE-agent-LM-32B. This decoupling indicates that RL does not simply suppress collaboration, but instead trains SWE-Protégé to collaborate *more productively*, converging in fewer overall steps. Importantly, shorter trajectories do not imply premature termination; SWE-Protégé maintains the ability to remain focused in long-horizon settings: on average, 10.8% of instances are resolved after $\geq$40 steps. Combined with our findings earlier, these effects explain the substantial cost savings observed in Fig. 4. SWE-Protégé saves cost along two orthogonal dimensions: (i) minimizing expensive expert tokens, and (ii) shortening trajectories overall.

Stalling is Replaced with Effective Collaboration. Prior work has identified repetitive actions as a major failure mode for SLMs (Pan et al., 2024; Yang et al., 2025b), e.g., models may repeatedly issue localization/inspection commands (e.g., search, open, grep) We quantify this behavior using repeated tool-use sequences: we define a *loop* as a contiguous repetition of a tool-action pattern with length at least L, and report the fraction of trajectories that contain any loop longer than L. Fig. 6 shows that SWE-Protégé-7B post-SFT (P1) suffers from severe looping: 31.0% of trajectories contain a repeated-action run longer than 10 steps (19.0% $>$ 20; 8.0% $>$ 40), comparable to SWE-agent-LM-7B/32B (33.6%/24.4% $>$ 10). Phase II RL (P2) sharply reduces this failure mode: only 0.8% of trajectories have runs longer than 10 and none exceed 20, outperforming Sonnet 3.7 and 4.5 (both 1.8% $>$ 10). As seen in Fig. 8 and Fig. 9, under our *loop aggressive* shaping stage, loop violations collapse (mean loop penalty rises from ≈ -1.3 toward ≈ 0; loop-negative rate drops from $\approx$ 0.50–0.65 to near-zero by $\approx$ 100 global steps), yet SWE-bench accuracy improves only marginally. Performance gains emerge only after switching to *follow-aggressive* shaping, where the policy learns to request and

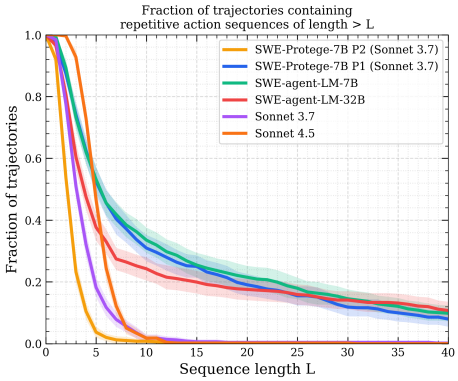


Figure 6: Repetitive unproductive actions are largely eliminated in SWE-Protégé-7B after Phase 2 RL, though they persist even after Phase 1 SFT.

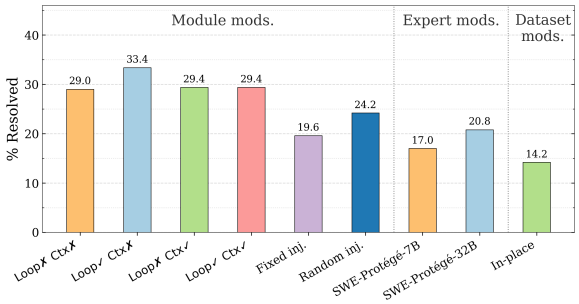


Figure 7: SWE-Protégé variants on SWE-bench Verified. Module modifications explore some alternative collaboration strategies. In "Expert mods.", we evaluate two lightly post-trained in-house experts. In "Dataset mods.", we approximate our SFT data generation process using in-place modifications to existing trajectories.

execute expert guidance faithfully: the mean follow-through score increases from roughly ≈ 0.2 early in training to ≈ 0.8 by the end, and the fraction of trajectories with negative follow-through declines from $\approx$ 0.30–0.40 to ≈ 0.05. This allows us to convert stalled states into forward progress. We corroborate this via failure-mode shifts (Fig. 5). Relative to post-SFT (P1), post-RL (P2) markedly reduces aborts due to runtime limits, a common failure even for SWE-agent-LM-32B, indicating that SWE-Protégé no longer gets stuck. Instead, it decisively follows expert guidance through to completed end-to-end attempts, even when that guidance is imperfect.

Ablations on SWE-Protégé Variants. We test a number of variants (Fig. 7) using SWE-Protégé-7B P1. In *Module mods.*, we study alternative expert–protégé collaboration strategies. Specifically, we evaluate (i) a looping intervention heuristic that forcibly invokes the expert once a loop-length threshold (initial: 15, subsequent: 8) is exceeded (*Loop*), and (ii) limiting the interaction history passed to the expert to only the 5 most recent messages (*Ctx*). We find that enabling the loop heuristic improves performance only when the expert observes the full interaction history: Loop✓ Ctx× achieves 33.4%, compared to 29.0% for Loop× Ctx× (+4.4%). However, with Ctx✓, the loop heuristic provides no measurable benefit (29.4% with or without Loop under Ctx✓). Moreover, variants that *remove* the policy's ability to autonomously request help and instead rely solely on passive expert invocation underperform sharply: fixed-interval invocation drops to 19.6%, and random invocation to 24.2%, despite receiving comparable or even more frequent expert advice. We view other collaboration strategies as an important direction for future work.

In *Expert mods.*, we evaluate in-house experts initialized from our own SWE-Protégé-7B and SWE-Protégé-32B checkpoints and fully SFT'ed to serve as experts using a separate instruction-tuning dataset, which is derived from the expert-augmented trajectories (§3.1): we treat the pre-expert interaction context as input and the expert message as the target, yielding 5,623 examples. Both in-house experts underperform frontier backends, though increasing expert capacity still helps: replacing SWE-Protégé-7B expert with SWE-Protégé-32B expert improves task resolution from 17.0% to 20.8% under the same protégé policy. We therefore treat expert post-training as an important but non-trivial and orthogonal direction, and leave a systematic exploration to future work. Finally, in *Dataset mods.*, we test whether §2.2's synthetic data generation (fresh expert-augmented rollouts) can be approximated by *in-place* modifications of existing SWE-smith trajectories. The in-place control—injecting additional expert calls into existing traces and minimally editing subsequent messages to acknowledge them—performs worst (14.2%), even below the SWE-agent-LM-7B baseline (17.0%). This gap suggests that the gains from our data pipeline stem not from simply adding expert tokens, but from inducing coherent interactions that the protégé conditions on.

Contamination Study. Because SWE-Protégé relies on external expert models, gains on SWE-bench Verified could in principle be inflated by data leakage or memorization. To assess robustness, we evaluate on a held-out SWE-smith-style subset of 400 tasks that was explicitly excluded from the trajectory-generation mixture used to train our 7B model. Importantly, this subset was released

after the expert models became available (e.g., Sonnet 3.7 was released on February 24, 2025, while SWE-smith was released on April 29, 2025), reducing the likelihood of overlap. We follow the same interaction protocol as in the main evaluation: a single rollout with a fixed step/cost budget while furnishing the expert with the same compact context $\tilde{s}_i$ (§2.3); as before, the expert observes only truncated recent context and has no access to the gold patch or unit-tests. Under this shifted evaluation, the 7B model achieves 32.0% accuracy without expert calls and 40.3% with Sonnet 3.7, while the 32B variant achieves 41.5% without expert calls and 43.0% with Sonnet 3.7, indicating that gains from SWE-Protégé persist beyond the training distribution. The smaller gain at 32B (+1.5%) suggests diminishing returns from a fixed black-box expert as the protégé grows stronger (§5).

General Capability Retention. Finally, we check whether SWE-Protégé degrades performance on general math and reasoning tasks, using single-turn benchmarks where expert use is unnecessary and not exposed to the model. On MATH (Hendrycks et al., 2021), GSM8K (Cobbe et al., 2021), and MMLU (Hendrycks et al., 2020), SWE-Protégé-7B achieves 66.3, 86.1, and 68.2, respectively, remaining within 1% absolute of the baseline Qwen2.5-Coder-7B-Instruct model (66.8, 86.7, and 68.7) (Hui et al., 2024). This is expected because our post-training is intentionally data-light and primarily targets specific behaviours (e.g., multi-turn stalled-state recognition), rather than broad capability acquisition.

5 Discussion

Limitations and Future Work. SWE-Protégé is a hybrid system whose accuracy depends on access to a frontier expert; our claims concern how an SLM can best use a limited frontier-token budget, not standalone SLM capability. While we focus on SWE-bench within the SWE-agent framework, SWE-Protégé's techniques are not tied to this setting and could in principle be applied to other domains (e.g., data analysis). We do not exhaustively explore design choices: e.g., Phase I/II hyperparameters, alternate collaboration strategies (e.g., expert interrupts or richer bidirectional control), or broader student model families; since our goal is to establish that SWE-Protégé can materially improve SLM performance rather than fully optimize the frontier. Phase II uses a single default 100-task subset whose selection and size we did not ablate, and our stall penalty relies on command normalization for bash-style actions (App. E.2), untested on scaffolds with other action spaces (e.g., GUIs). Finally, we treat the expert as a fixed black-box backend and leave more principled expert post-training and co-adaptation as an important future direction, which may also sustain gains at larger protégé scales, where they currently shrink (§4.1).

Related Work. We discuss prior work on software agents, model routing, SLMs, and knowledge distillation and LLM–SLM collaboration in App. A.

Conclusion. We show that SLMs, while previously lagging on long-horizon software repair, can achieve strong SWE-bench performance when trained to collaborate effectively with an expert. SWE-Protégé lightly post-trains Qwen2.5-Coder-7B-Instruct to reach **42.4%** Pass@1 on SWE-bench Verified with Opus 4.1 as the expert (**+25.4%** over the prior SLM SOTA), while using expert assistance sparsely (≈11% of total tokens). Our results suggest that learned expert–protégé collaboration is a practical path toward effective and cost-efficient agentic systems in which an SLM is the primary decision-maker and a frontier expert is consulted sparingly.

Acknowledgments and Disclosure of Funding

Funding. This work was conducted at and funded by Meta. No third-party funding supported this work. **Competing interests.** The authors declare no competing interests.

References

Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, Congcong Chen, et al. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. *arXiv preprint arXiv:2503.01743*, 2025.

Anthropic. Introducing claude 3.5 sonnet, 2024. `https://www.anthropic.com/news/claude-3-5-sonnet`.

Anthropic. Introducing claude 3.7 sonnet, 2025. `https://www.anthropic.com/news/claude-3-7-sonnet`.

Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, Xuan-Son Nguyen, Colin Raffel, Leandro von Werra, and Thomas Wolf. SmolLM3: smol, multilingual, long-context reasoner. `https://huggingface.co/blog/smollm3`, 2025.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai. *arXiv preprint arXiv:2506.02153*, 2025.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing swe-bench verified, 2024. *URL https://openai.com/index/introducing-swe-bench-verified*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Cursor. The best way to code with ai, 2026. `https://cursor.com/home`.

Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. Combining small language models and large language models for zero-shot NL2SQL. *Proceedings of the VLDB Endowment*, 17(11):2750–2763, 2024. doi: 10.14778/3681954.3681960.

Luyang Fang, Xiaowei Yu, Jiazhang Cai, Yongkai Chen, Shushan Wu, Zhengliang Liu, Zhenyuan Yang, Haoran Lu, Xilin Gong, Yufang Liu, Terry Ma, Wei Ruan, Ali Abbasi, Jing Zhang, Tao Wang, Ehsan Latif, Wei Liu, Wei Zhang, Soheil Kolouri, Xiaoming Zhai, Dajiang Zhu, Wenxuan Zhong, Tianming Liu, and Ping Ma. Knowledge distillation and dataset distillation of large language models: Emerging trends, challenges, and future directions. *Artificial Intelligence Review*, 59(1), 2025. doi: 10.1007/s10462-025-11423-3.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Zenan Huang, Yihong Zhuang, Guoshan Lu, Zeyu Qin, Haokai Xu, Tianyu Zhao, Ru Peng, Jiaqi Hu, Zhanming Shen, Xiaomeng Hu, et al. Reinforcement learning with rubric anchors. *arXiv preprint arXiv:2508.12790*, 2025.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

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, 2025. `https://arxiv.org/abs/2504.07164`.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. *arXiv preprint arXiv:2306.02561*, 2023.

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? *arXiv preprint arXiv:2310.06770*, 2023.

Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, et al. Universal model routing for efficient llm inference. *arXiv preprint arXiv:2502.08773*, 2025.

Devvrit Khatri, Lovish Madaan, Rishabh Tiwari, Rachit Bansal, Sai Surya Duvvuri, Manzil Zaheer, Inderjit S Dhillon, David Brandfonbrener, and Rishabh Agarwal. The art of scaling reinforcement learning compute for llms. *arXiv preprint arXiv:2510.13786*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

Lipeng Ma, Weidong Yang, Yixuan Li, Ben Fei, Mingjie Zhou, Shuhao Li, Sihang Jiang, Bo Xu, and Yanghua Xiao. Adaptivelog: An adaptive log analysis framework with the collaboration of large and small language model. *ACM Transactions on Software Engineering and Methodology*, 35(5): 1–42, 2026. doi: 10.1145/3749840.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

meta-pytorch. TorchTune: PyTorch native post-training library. `https://github.com/meta-pytorch/torchtune`, 2026.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287. Citeseer, 1999.

NovaSky-AI. SkyRL: A Modular Full-stack RL Library for LLMs. `https://github.com/NovaSky-AI/SkyRL`, 2026.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

OpenClaw. Personal ai assistant, 2026. `https://github.com/openclaw/openclaw`.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training Software Engineering Agents and Verifiers with SWE-Gym, December 2024. `http://arxiv.org/abs/2412.21139`. arXiv:2412.21139 [cs].

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

SumanthRH. Swe-gym-subset. `https://huggingface.co/datasets/SumanthRH/SWE-Gym-Subset`, 2026. Accessed: 2026-01-18.

SWE-bench. SWE-smith dataset. `https://huggingface.co/datasets/SWE-bench/SWE-smith`, 2026a.

SWE-bench. SWE-smith Trajectories. `https://huggingface.co/datasets/SWE-bench/SWE-smith-trajectories`, 2026b.

Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *ACM Transactions on Intelligent Systems and Technology*, 16(6):1–87, 2025.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing llm reasoning via reinforcement learning on open software evolution, 2025. `https://arxiv.org/abs/2502.18449`.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025.

Chuanpeng Yang, Yao Zhu, Wang Lu, Yidong Wang, Qian Chen, Chenlong Gao, Bingjie Yan, and Yiqiang Chen. Survey on knowledge distillation for large language models: Methods, evaluation, and application. *ACM Transactions on Intelligent Systems and Technology*, 16(6):1–27, 2025a. doi: 10.1145/3699518.

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. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. In *Forty-first International Conference on Machine Learning*, 2024.

Yiqun Zhang, Hao Li, Chenxu Wang, Linyao Chen, Qiaosheng Zhang, Peng Ye, Shi Feng, Daling Wang, Zhen Wang, Xinrun Wang, et al. The avengers: A simple recipe for uniting smaller language models to challenge proprietary giants. *arXiv preprint arXiv:2505.19797*, 2025.

A Related Work

Training Software Agents. SWE-Smith (Yang et al., 2025b) addresses data scarcity by synthesizing large numbers of verifiable software engineering task instances from 128 GitHub repositories. Lingma-SWE-GPT (Ma et al., 2024) adopts a development-process-centric training scheme for 7B/72B variants; SWE-Gym (Pan et al., 2024) introduces an open training environment that improves agents at 7B/32B scales; SWE-Fixer (Xie et al., 2025) trains specialized retriever and editor models for efficient issue resolution. The above approaches rely on distilled data from frontier models (e.g., GPT or Claude), and post-train Qwen2.5 models via supervised finetuning. SWE-RL (Wei et al., 2025) applies RL to Llama 3 (Grattafiori et al., 2024), requiring substantial data/compute (e.g., 273k seed tasks, 512 H100 GPUs) and a custom agent scaffold. CWM (Copet et al., 2025) (32B) performs extensive end-to-end training, combining large-scale data/compute, custom agent scaffold, and test-time scaling to achieve strong SWE-bench performance. In contrast, SWE-Protégé focuses on lightweight post-training and demonstrates that an SLM can reach competitive performance relative to SOTA open-weight models.

Model Routing. Here, a router selects the most suitable LM to handle a query. Existing work has so far focused on *per-task* routing and *single-turn* tasks: e.g., *non-predictive* routing, which executes models sequentially and escalates based on output evaluation until a quality threshold is met (Chen et al., 2023); and *predictive* routing, which selects a model *a priori* for a task using LM-based heuristics (Ong et al., 2024), learned neural routers (Jiang et al., 2023), or cluster-based methods (Jitkrittum et al., 2025; Zhang et al., 2025). In contrast, we study long-horizon, multi-turn agentic coding tasks, where defining a reliable per-step routing signal is ill-posed (e.g., a syntax error does not necessarily justify escalation). Moreover, we allow the participating LMs to *self-determine when and how to collaborate*; in our instantiation, the SLM performs the bulk of routine reasoning and actions and experts are invoked selectively as needed.

Small Language Models. SLMs are increasingly viewed as practical agents when the domain scope is narrow, offering advantages in inference efficiency (e.g., latency, memory footprint) and fine-tuning agility (Belcak et al., 2025). Many use-cases (Wang et al., 2025) have emerged: e.g., single-turn question answering (Abouelenin et al., 2025), mathematical reasoning (Guan et al., 2025), and single-turn coding tasks (Bakouch et al., 2025). In contrast, we show that an SLM, selectively collaborating with a frontier expert, can be effective on a long-horizon, agentic coding task.

Knowledge Distillation and LLM–SLM Collaboration. Knowledge distillation transfers capabilities from a large teacher into a smaller student (Yang et al., 2025a; Fang et al., 2025); the teacher is absent at inference, whereas in SWE-Protégé the expert remains available and is consulted selectively (we compare against plain distillation at matched data scale in App. B). Closer to our setting, recent systems pair SLMs with LLMs: AdaptiveLog (Ma et al., 2026) routes log-analysis queries on which the SLM is uncertain to an LLM via an external uncertainty estimate, and ZeroNL2SQL (Fan et al., 2024) fine-tunes an SLM to generate SQL sketches that an LLM then completes, with the division of labor fixed a priori by the pipeline. Both make the collaboration decision outside the small model, at per-task or per-stage granularity, on single-turn tasks. SWE-Protégé instead places the escalation decision inside the policy, at any step of a multi-turn trajectory, and additionally trains the SLM to formulate useful queries and follow through on guidance, which routing alone cannot supervise; our fixed-interval and random expert-invocation ablations (§4.1) serve as routing-like, non-adaptive controls.

B Extended Evaluation

We include additional diagnostics from Phase II training in Fig. 8. Mean *Expert* reward remains highly stable throughout training, staying within a narrow band of approximately 0.82–0.88 across all steps, indicating that expert responses consistently satisfy the reward criterion and are not degraded by later-stage shaping. In contrast, *Follow* reward exhibits a clear upward trend. During the first stage (steps 1–80), it fluctuates substantially, ranging roughly from 0.2 to 0.5. After step 80, following the transition to follow-aggressive shaping, it increases steadily and stabilizes in the 0.75–0.85 range by step 160. *Loop* reward shows the most dramatic change: it begins near -1.5, rises rapidly during the loop-first shaping phase, and approaches 0.0 by step 80. Thereafter it remains close to zero with minimal variance, indicating that degenerative looping behavior is largely eliminated and does not re-emerge in later training. The mean number of expert calls stays relatively stable at approximately

Model	Strategy	Pass@1 (%)	Avg. Expert Calls
SWE-agent-LM-7B	Random Injection	20.1	5.9
Lingma-SWE-GPT-7B	Random Injection	21.4	5.8
SWE-gym-7B	Random Injection	15.8	5.7
SWE-Protégé-7B	P1	29.0	3.6
SWE-Protégé-7B	P2	30.8	4.1

Table 3: Expert-access control using the same Sonnet 3.7 expert backend. Random injection provides baselines with expert access but does not train selective collaboration.

3.5–4.5 calls per episode, with a mild upward drift in later stages, suggesting that performance gains are not driven by increased expert reliance. Finally, mean step counts decrease substantially over training, dropping from roughly 55–65 steps early on to about 35–40 steps by the end of training, consistent with more direct trajectories and reduced redundant actions.

Fig. 9 further characterizes training dynamics by reporting the fraction of trajectories whose *Follow* or *Loop* reward is non-positive. Early in training, a substantial fraction of trajectories incur negative rewards for both components. Within the first 20–30 steps, approximately 40%–65% of trajectories receive non-positive *Loop* reward, while roughly 25%–40% exhibit non-positive *Follow* reward. As training progresses, both fractions decline steadily. The fraction of trajectories with non-positive *Loop* reward drops sharply during the loop-first shaping phase and falls below 10% by around step 50, approaching zero shortly thereafter and remaining near zero for the remainder of training. The fraction of trajectories with non-positive *Follow* reward decreases more gradually, falling below 10% by approximately step 100 and stabilizing in the 3%–8% range by the end of training. Train and evaluation curves closely track each other throughout, indicating that these behavioral improvements generalize beyond the training trajectories.

Fig. 6 analyzes degenerative looping behavior during Phase I, by measuring the fraction of trajectories that contain repetitive tool-use sequences of length greater than L. Our evaluation here also includes an active intervention mechanism that is activated at $L = 15$ for the initial looping sequence detected, and $L = 8$ for subsequent detections. Across all agents, the fraction decreases monotonically as L increases, but with substantial differences in tail behavior. Baseline agents exhibit long repetitive sequences with high frequency: the SWE-agent-LM-32B baseline retains approximately 15%–20% of trajectories with repetition length exceeding $L = 10$, and remains above 10% even at $L = 40$. In contrast, all SWE-Protégé-7B variants substantially suppress long repetitive sequences. For SWE-Protégé-7B, the fraction of trajectories with repetition length exceeding $L = 10$ drops below 10%, and falls to near zero by $L \approx 15$, indicating that extended tool-use loops are largely captured and eliminated properly by the active intervention mechanism; however, as we showed in Fig. 7, this is still ineffective in improving SWE-bench performance. Among SWE-Protégé variants, differences across expert backends (not shown in the figure for clarity) are minor and primarily affect short repetition lengths, while the long-tail behavior remains consistently suppressed. To construct these curves, for each trajectory we compute the maximum length of any consecutive identical tool-use action run, and then for each threshold $L \in [0, 40]$ we plot the fraction of trajectories whose maximum run length exceeds L (i.e., $\mathbb{E}[\mathbb{1}[\max_\text{run} > L]]$ over trajectories). Shaded regions indicate 95% bootstrap uncertainty bands obtained by resampling trajectories within each group (400 resamples; fixed random seed for determinism) and evaluating the same indicator-based curve on each resample.

Expert-access control. We also evaluate whether performance gains can be explained by simply granting baselines access to the same Sonnet 3.7 expert backend. For this control, we instrument the corresponding agent frameworks with the same `ask_expert_llm` tool and apply random expert injection rather than training the model to decide when and how to ask for help. Table 3 reports pass@1 and the average number of expert calls. Random injection improves the baselines modestly, but remains below SWE-Protégé P1/P2 despite using more expert calls on average, indicating that SWE-Protégé benefits from learning to selectively request and act on expert guidance rather than from expert access alone.

Reference-solution ablation. During Phase I data generation the expert has access to the gold patch (§2.2), whereas at inference neither the protégé nor the expert does. To test whether this asymmetry biases the learned policy, we trained a P1 model on an SFT trajectory pool generated with the expert

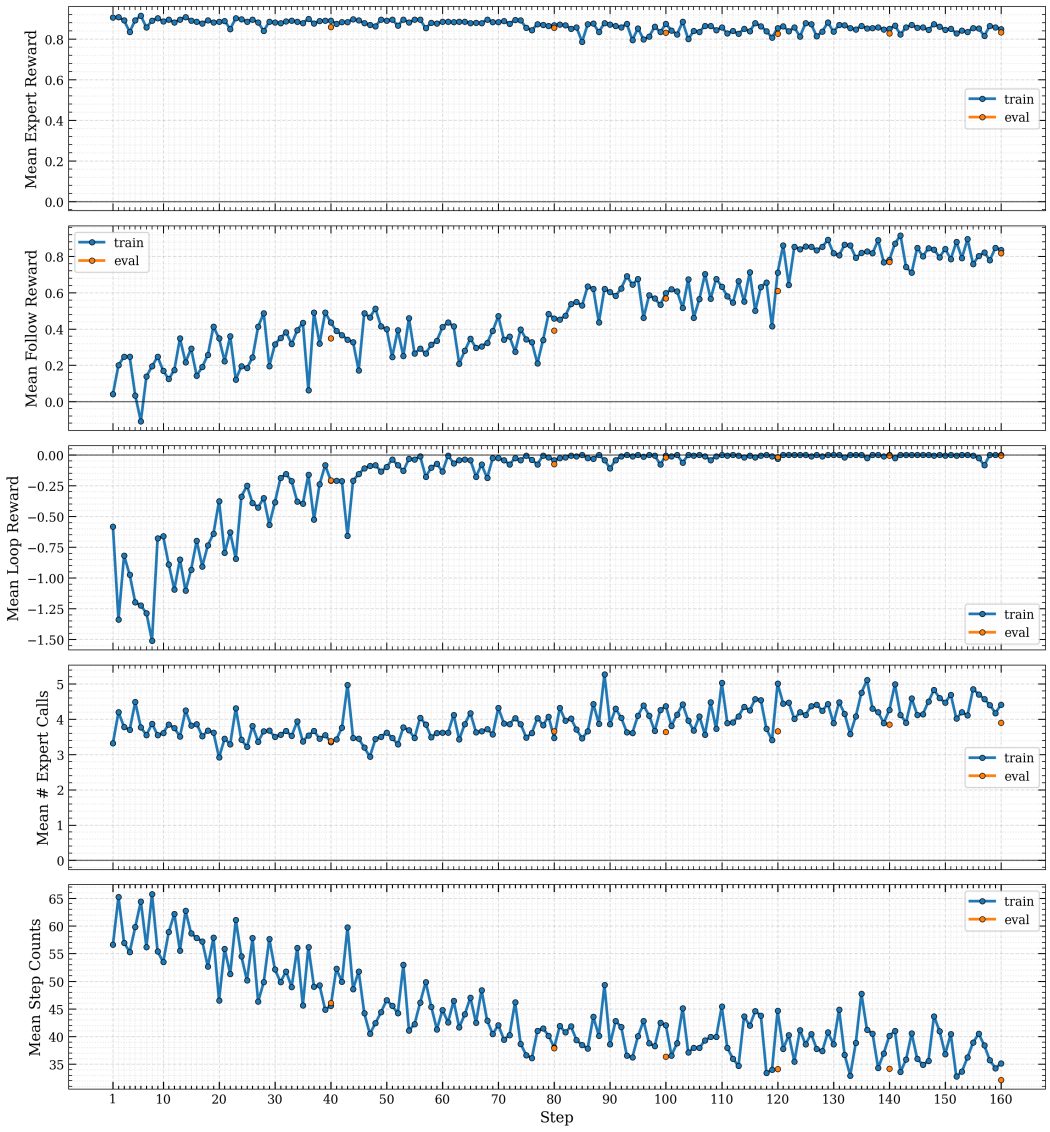


Figure 8: Phase II training logs starting from the SFT checkpoint, which include mean rewards for selected components.

given no gold patch, sampled to the same trajectory counts as in Fig. 2. We observed negligible differences ($\pm 0.3\%$) in test-time performance across all expert backends. The gold patch therefore mainly raises the yield of successful trajectories for rejection sampling (38% more usable trajectories) rather than leaking answers into the SFT targets.

Teacher-quality control. We additionally isolate whether the gains of SWE-Protégé can be explained by stronger teacher trajectories alone. Under plain distillation with the same $\approx$ 5K data scale, pass@1 increases as teacher quality improves: Sonnet 3.7 yields 17.0%, Sonnet 4.5 yields 23.6%, and Opus 4.1 yields 24.2%. Although stronger teachers improve distillation, these results remain substantially below SWE-Protégé trained from Sonnet 3.7 expert-augmented trajectories at the same data scale, suggesting that the learned expert-collaboration policy contributes beyond teacher strength alone.

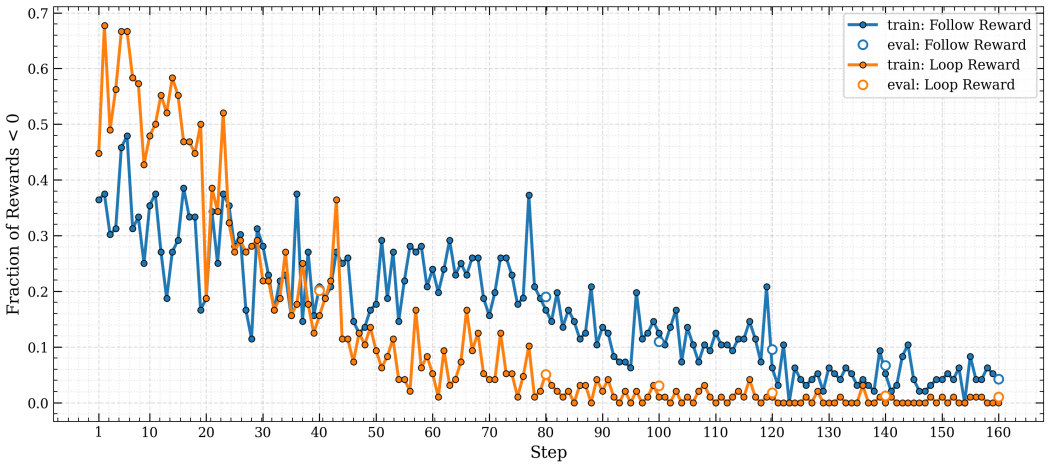


Figure 9: Fraction of trajectories with *Loop* and *Follow* reward ≤ 0 during Phase II training.

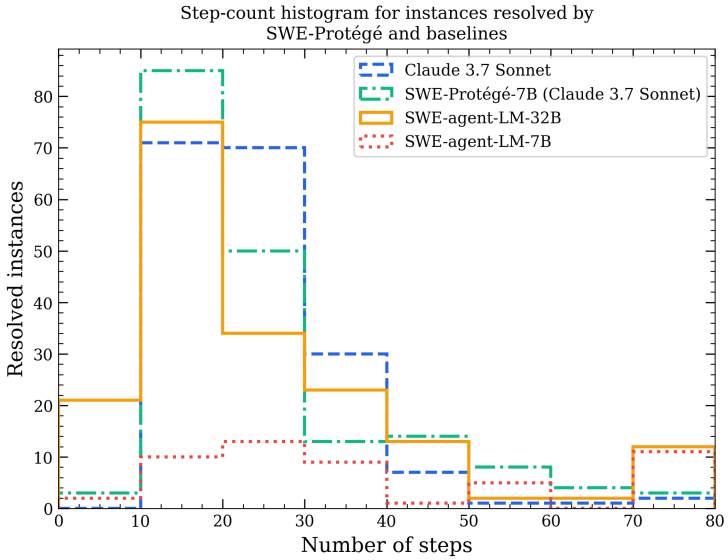


Figure 10: Agent behavior diagnostics adapted from Yang et al. (2025b). Histogram of the number of steps taken to complete the task per resolved trajectory.

C Expert-augmented Synthetic Trajectory Generation

We use Claude Sonnet 3.7 to generate full expert-augmented trajectories following the procedure described in §2.2, using the same underlying task dataset as SWE-smith (SWE-bench, 2026a; Yang et al., 2025b). Tasks are randomly sampled, and trajectories are generated with rejection sampling until we obtain 4.8K accepted trajectories, approximately matching the training set size used in SWE-smith. Because the exact task composition may differ due to random sampling, we perform a controlled check by regenerating trajectories *without* expert augmentation and applying SFT to the same baseline model (Qwen-2.5-Coder-Instruct, 7B and 32B variants) using an identical 4.8K-sample budget. The resulting SWE-bench performance closely matches that reported for SWE-smith, indicating that differences observed in later experiments are not attributable to dataset size or sampling effects. The prompt used to generate expert-augmented trajectories is shown below in Fig. 11. Importantly, the expert used in synthetic data generation is Claude Sonnet 3.7 itself; the system prompt fed to it is described in Fig. 13. The exact expert mechanism is described in §D. Note that, apart from synthetic data generation, the gold patch and unit tests are never provided to the expert.

You are a helpful assistant that can interact with a computer to solve tasks.

Call `ask_expert_llm` to help guide your thinking, before/after localizing the issue, applying a patch, when considering edge cases, and whenever you have trouble reproducing the error (though not always necessary).

Instance template.
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python code repository in the directory {{working_dir}}. Consider the following PR description:

<pr_description>
{{problem_statement}}
</pr_description>

<context>
{{context}}
</context>

Can you help me implement the necessary changes to the repository so that the requirements specified in the <pr_description> are met?
I've already taken care of all changes to any of the test files described in the <pr_description>. This means you **DON'T** have to modify the testing logic or any of the tests in any way!
Your task is to make the minimal changes to non-tests files in the {{working_dir}} directory to ensure the <pr_description> is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to find and read code relevant to <pr_description>. Call the `ask_expert_llm` tool before or after doing this as needed.
2. Create a script to reproduce the error and execute it with `python <filename.py>` using the bash tool, to confirm the error. Call the `ask_expert_llm` tool before or after doing this as needed.
3. Edit the source code of the repo to resolve the issue. Call the `ask_expert_llm` tool before or after doing this as needed.
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edge cases and make sure your fix handles them as well. Call the `ask_expert_llm` tool before or after doing this as needed.

Your thinking should be thorough and so it's fine if it's very long. Use `ask_expert_llm` in moderation—avoid consecutive calls, and always integrate its response into your reasoning (e.g., "Based on the expert's advice").

Figure 11: System prompt used for expert-augmented trajectory generation. This is a slight modification of the standard SWE-agent prompt, adding explicit guidance for when to invoke `ask_expert_llm` and how to incorporate expert feedback, while keeping the underlying task structure unchanged.

You are a helpful assistant that can interact with a computer to solve tasks.

<IMPORTANT>
• If the user provides a path, you should **NOT** assume it's relative to the current working directory. Instead, explore the file system to find the file before working on it.
• Call `ask_expert_llm` to help guide your thinking, e.g., before/after localizing the issue, before/after applying a patch, when thinking through edge cases, and whenever you encounter problems reproducing the error.
When you call `ask_expert_llm`, treat it as a multi-turn collaborator: ask for specific, actionable steps; then execute those steps; then report back what you did and what you observed before asking again.
</IMPORTANT>

You have access to the following functions:

```
-- BEGIN FUNCTION #1: bash --
Description: Execute a bash command in the terminal.
Parameters:
  (1) command (string, required): The bash command to execute. Can be empty
to view additional logs when previous exit code is -1. Can be ctrl+c to
interrupt the currently running process.
-- END FUNCTION #1 --

<....omitted for clarity...>

-- BEGIN FUNCTION #4: ask_expert_llm --
Description: Call ask_expert_llm to help guide your thinking, e.g., when
thinking through edge cases.
Parameters:
  (1) question (string, required): The expert question, e.g., Do you have
any advice on how to proceed?
  (2) budget_tokens (integer, optional): Token budget hint if applicable
-- END FUNCTION #4 --

<....omitted for clarity...>
```

Can you help me implement the necessary changes to the repository so that the requirements specified in the `<pr_description>` are met?
I've already taken care of all changes to any of the test files described in the `<pr_description>`. This means you **DON'T** have to modify the testing logic or any of the tests in any way!
Your task is to make the minimal changes to non-tests files in the `{{working_dir}}` directory to ensure the `<pr_description>` is satisfied.

Follow these steps to resolve the issue:
1. As a first step, it might be a good idea to find and read code relevant to `<pr_description>`. Call the `ask_expert_llm` tool before or after doing this as needed.
2. Create a script to reproduce the error and execute it with `python <filename.py>` using the bash tool, to confirm the error. Call the `ask_expert_llm` tool before or after doing this as needed.
3. Edit the source code of the repo to resolve the issue. Call the `ask_expert_llm` tool before or after doing this as needed.
4. Rerun your reproduce script and confirm that the error is fixed!
5. Think about edge cases and make sure your fix handles them as well. Call the `ask_expert_llm` tool before or after doing this as needed.

Your thinking should be thorough and so it's fine if it's very long. Use `ask_expert_llm` in moderation—avoid consecutive calls, and always integrate its response into your reasoning (e.g., "Based on the expert's advice").

Figure 12: Prompt used by SWE-Protégé agents. This prompt is a slight modification of the standard SWE-agent prompt: it (i) makes `ask_expert_llm` usage explicit and structured as a multi-turn collaborator, (ii) adds a path-resolution instruction to avoid incorrect assumptions about file locations, and (iii) exposes the tool/function-call interface used during rollouts, while preserving the underlying task and repository setup.

D Expert Tool Details

The expert prompt used by Claude Sonnet 3.7 when generating expert-augmented trajectories is shown in Fig. 13. The standard expert prompt (used everywhere else) is shown in Fig. 14. There are two prompts used by our expert judge: the first is used to judge whether the call to the expert was appropriate (Fig. 15), and the second is an expert judge call to obtain the *Follow* score (Fig. 16). Finally, the prompt used by SWE-Protégé SLM agents is shown in Fig. 12.

Expert tool implementation (SWE-agent instantiation). Our "expert" is implemented using standard agent tooling: an external advice function (`ask_expert_llm`) exposed to the agent alongside regular tools (e.g., `bash` and `submit`). Concretely, SWE-agent loads this tool via a tool-bundle entry (the `tools/expert_llm` bundle) in the agent YAML configuration.

Context passed to the expert. When invoked, `ask_expert_llm` reads the problem statement and a recent window of the agent's conversation from SWE-agent's exported history files inside the container. The tool constructs a single structured context block containing (i) the task description and (ii) a JSON dump of recent agent messages (after history processing), and appends the agent's explicit question. This context is provided as background (not as the expert's own prior turns), and the expert response is returned as an observation wrapped in `<expert_llm_guidance>` tags.

Expert routing, limits, and optional scoring. The expert model can be configured independently of the main agent (e.g., via `EXPERT_MODEL` and `EXPERT_OPENAI_API_BASE`), enabling a separate endpoint/model for expert calls. To control cost, the tool enforces a per-task call quota (default 6).

Preventing expert advice from being elided by history processing. In SWE-agent, for SWE-agent-LM 7B/32B models, the model input context is derived from the raw interaction history via a configurable chain of `history_processors`. A commonly used default processor (`last_n_observations`, often with $n{=}5$) elides older tool observations by replacing them with a short "(lines omitted)" placeholder to control context length. In our SWE-smith instantiation, we slightly modify this behavior so that observations containing the `<expert_llm_guidance>` marker are never elided by `last_n_observations`. This ensures the agent continues to see the expert's advice beyond the last-n window, preventing it from disappearing after a handful of turns.

RL-oriented expert supervision: warranted-score, follow-through, and a post-hoc terminal judge. To support reinforcement learning with expert-augmented trajectories, we extend the standard `ask_expert_llm` tool call with two auxiliary judging mechanisms and log their outputs into the trajectory metadata for reward shaping. First, when `ASK_EXPERT_LLM_ENABLE_SCORING=1`, the expert is prompted to return strict JSON containing both textual guidance and a continuous `warranted_score`$\in [0,1]$ indicating whether escalation was justified; the tool returns only the guidance to the agent but records the score for training-time penalties/rewards. Second, when `ASK_EXPERT_LLM_ENABLE_FOLLOW_JUDGE=1`, `ask_expert_llm` issues an additional hidden judge call that scores adherence to the *previous* expert guidance (call $k{-}1$) based on the subsequent agent messages up to the current expert invocation (call k), producing `follow_score`$\in [0,1]$ that is logged as `expert_follow_score` (not shown to the agent). A key semantic consequence is that the *final* expert guidance call has no later call to "carry" its follow-through score; therefore, in our modified SkyRL training harness we add an out-of-loop, post-hoc follow judge that replays the same follow-judge logic after rollout completion and writes a separate `final_expert_follow_score` field into `trajectory.info` (used by the RL reward without altering per-call expert penalties). These expert-derived scores are then weighted in the composite reward (e.g., via `generator.expert_cfg.weight` and `generator.follow_cfg.weight` in the SkyRL launcher).

You are an expert software engineer solving SWE-bench–style tasks.

Provide the most direct and precise guidance to the agent's question.
Treat any provided conversation or repository details strictly as background context—they are *not* your prior messages.

You are also given a ground-truth patch.
This patch represents the correct solution: do **not** copy it verbatim, but use it to direct the agent toward the correct solution as clearly and quickly as possible.

If the agent's framing is inaccurate, restate the real issue clearly and redirect them to the correct solution.
Keep your answer concise, actionable, and technically accurate.

Figure 13: System prompt used for Claude Sonnet 3.7 acting as the expert during expert-augmented trajectory generation. The expert is instructed to provide high-level corrective guidance grounded in a hidden ground-truth patch, without revealing the solution verbatim.

You are an expert software engineer assisting an agent solving SWE-bench–style tasks.

Provide clear, direct, and technically precise guidance that helps the agent make concrete progress (e.g., reproducing the bug, isolating the faulty logic, or implementing a minimal fix).
You may have strong intuitions about the root cause or solution, but focus on communicating actionable steps the agent should take in the next few steps.

Treat any provided conversation or repository details strictly as background context—they are the agent's own prior messages, not yours.
If you believe the agent is on the wrong track (e.g., it has not found the right file, diagnosis, fix direction, or even the right question), redirect it to the correct line of investigation. More generally, respond as you would if you were solving the task at this moment.

Keep your answer concise, actionable, and technically accurate.
As the agent is solving SWE-bench–style tasks, do not mention pull requests, commit messages, or other GitHub workflow artifacts.

Figure 14: System prompt used by the expert model under our standard expert-collaboration setting. The expert is instructed to provide concise, actionable guidance and to correct the agent when it diverges from the most promising debugging path.

(Expert-judge extension; appended to the expert prompt)

Return **ONLY** a single valid JSON object (no Markdown, no extra text) with exactly the following schema:

```
{
  "guidance": string,
  "warranted_score": float,
}
```

where `warranted_score` is in $[0.0, 1.0]$:
- Values below 0.5 indicate that the expert call was generally unnecessary.
- A value of 1.0 indicates the expert call was clearly necessary and well-timed.
- Be strict in your scoring.

Example:

```
{
  "guidance": "Do X...\nThen verify Y...",
  "warranted_score": 0.8
}
```

Important:
If `guidance` spans multiple lines, encode newlines as `\n`.

Figure 15: Expert-judge prompt used to evaluate whether an expert call was warranted. This prompt is appended to the standard expert prompt (Fig. 14) and instructs the judge to both provide corrective guidance and assign a strict scalar score indicating whether the agent's expert query was appropriate.

System prompt (follow judge).

You are evaluating whether an agent followed prior expert guidance.
Return **ONLY** a single valid JSON object (no Markdown, no extra text) with exactly the following schema:

```
{
  "follow_score":  float,
  "rationale":  string
}
```

where `follow_score` is in $[0.0, 1.0]$:
- Values below 0.5 indicate that the agent generally did not follow prior expert guidance.
- Be strict in your scoring.

Scoring guidance.
If the agent was instructed to perform a specific action (e.g., view a file and apply an edit) but only partially followed it (e.g., viewed without editing), this should receive a low score. Requesting clarification or assistance in order to complete a specific instructed action should be considered as following that guidance for that action. If there is no clear instruction to follow, set `follow_score` to `null`.

User input to the judge.

```
<prior_expert_guidance>
<expert guidance text>
</prior_expert_guidance>

<agent_followup_conversation>
<subsequent agent messages (JSON-encoded)>
</agent_followup_conversation>

<agent_current_question_to_expert>
<current agent question>
</agent_current_question_to_expert>
```

Figure 16: Follow-judge prompt used to assess whether the agent followed prior expert guidance. The judge receives the previous expert advice, the agent's subsequent interaction trace, and the agent's current expert query, and outputs a strict scalar score indicating adherence to guidance along with a brief rationale.

E Phase II Reward Modeling and Shaping Details

This appendix defines the reward terms referenced in Phase II (§2.3). The goal is to make (i) degenerative stalling and (ii) non-collaborative expert usage explicitly suboptimal, while keeping rewards stable for RL post-training.

E.1 Correctness and Similarity Terms

These apply to the code patch (output artifact) generated by the coding agent. Both terms are computed similarly to (Copet et al., 2025).

Correctness. Let τ denote the full agent interaction trajectory (interleaved sequence of agent thoughts/actions and tool responses), and let $p(\tau)$ denote the final patch the agent submits at the end of τ. We compute $R_{\text{correct}}(\tau, x) \in \{0, 1\}$ by re-evaluating $p(\tau)$ in a fresh environment identical to the task backend used for rollout generation:

$$R_{\text{correct}}(\tau, x) = \mathbb{1}\{p(\tau) \text{ passes verification}\}.$$

A patch is considered correct (resolved) if all unit tests pass (e.g., as in SWE-Bench). Otherwise, it is considered unresolved. Here, x denotes the task instance metadata (e.g., the dataset record), which includes the gold patch used below and any information needed to re-run verification.

Similarity fallback (unresolved only). When unresolved ($R_{\text{correct}} = 0$), we compute a similarity score $\text{sim}(\tau, x) \in [0, 1]$ between the model patch and a gold patch after filtering diff noise. Concretely, we drop diffs for newly-added files (e.g., "`new file mode`" blocks or diffs involving `/dev/null`) and drop the git noise line `\No newline at end of file`, then compute string similarity on the remaining unified-diff text. With threshold θ (we use $\theta = 0.5$ in practice),

$$R_{\text{sim}}(\tau, x) = \begin{cases} 0, & \text{sim}(\tau, x) \geq \theta, \\ -1, & \text{sim}(\tau, x) < \theta. \end{cases}$$

This discrete fallback discourages unrelated patches while preserving reward stability. We use a thresholded signal to avoid instability under policy updates and prevent similarity from dominating optimization. Intuitively, when a trajectory fails verification, we still want a stable learning signal: clearly unrelated patches are penalized (-1), while patches that substantially overlap the gold fix receive a neutral score (0). This can improve learning on difficult instances where the policy cannot yet produce a fully test-passing patch, but can still learn to localize and edit the right region. If the model patch is missing (or becomes empty after filtering), we treat similarity as 0 and assign $R_{\text{sim}} = -1$.

E.2 Stall Penalty

Base-command extraction. To detect stalled progress (often manifesting as action loops), we convert each tool/action step into a *base command* and form a sequence $c_1, \ldots, c_M$. We normalize commands to make superficial variants count as the same action, then measure repetition on the normalized sequence. Our normalization mirrors the reward implementation:

- Strip leading environment-variable assignments (e.g., `FOO=bar python -m pytest` $\mapsto$ `python -m pytest`).
- For chained shell commands joined by `&&` or `;`, keep the *last* subcommand (e.g., `cd repo && pytest` $\mapsto$ `pytest`).
- Define the base command as the first token (e.g., `pytest`, `ls`), except for a small set treated as two-word bases (e.g., `git status`, `git diff`, `str_replace_editor view`).
- Collapse "navigation-like" operations into one equivalence class so mixed navigation streaks count as repetition: e.g., `grep`, `find`, and `str_replace_editor view` are treated as identical for stall detection.

This normalization follows the instantiation in SWE-smith (Yang et al., 2025b), supplemented by inspecting post-SFT trajectories until it covered the repetition patterns we observed. For example, repeated `git diff` actions are counted as identical, and a sequence like `grep` $\rightarrow$ `find` $\rightarrow$ `str_replace_editor view` is treated as a repeated navigation streak.

Let $s_1, s_2, \ldots$ denote lengths of maximal consecutive identical-command streaks *in temporal order* as we scan the trajectory (i.e., s_1 is the first streak length, s_2 the next streak after the command changes, etc.).

Triggered, capped stall penalty. We use a triggered rule: the penalty only activates once any streak is long enough; after the first trigger, subsequent streaks use a lower threshold $k_2 < k_1$. The charged exceedance matches the implementation, which uses a small "$+1$" offset so that a streak that just reaches the threshold is already penalized:

$$R_{\text{loop}}(\tau) = \max\Big(c_{\text{loop}},\ -\lambda_{\text{loop}}\Big(\max(0, s_1 - k_1 + 1) + \sum_{j \geq 2} \max(0, s_j - k_2 + 1)\Big)\Big), \qquad (4)$$

where $\lambda_{\text{loop}} > 0$ and $c_{\text{loop}} < 0$ caps the magnitude. Operationally, the first term contributes when $s_1 \geq k_1$; after the first trigger, later streaks contribute when $s_j \geq k_2$ for $j \geq 2$. This makes stalling *sparse but decisive*: short repeats are tolerated, while true degeneracy is sharply penalized.

E.3 Expert-as-Judge Collaboration Terms

These terms train the agent to collaborate with the expert in a multi-turn, pair-programming-like manner: they discourage unnecessary escalation, and they directly supervise the agent's behavior *after* receiving advice (executing the requested steps and reporting back).

Let $\mathcal{E}(\tau) = \{e_i\}_{i=1}^{N}$ denote the ordered set of expert-call events (as in §2.3), where each event $e_i = (t_i, q_i, g_i, \tilde{s}_i)$ records the timestep t_i, the agent query q_i sent to the expert, the expert guidance g_i, and the compact context packet $\tilde{s}_i$ provided to the expert. In our implementation, $\tilde{s}_i$ is a compact *bundle of recent context*, not a single message: it includes the problem statement and a truncated tail of recent processed conversation messages (a fixed-size window for token control) from τ, serialized for the expert. Concretely, we pass (at most) the last 5 processed messages, then drop the leading system prompt and the first user message that repeats the full problem statement, and remove action/thought fields from these messages before serialization. For follow-through, we associate each guidance g_i with the subsequent response segment $\Delta_i(\tau)$, i.e., the agent's behavior after receiving g_i up to and including the next expert call (or trajectory end if there is no subsequent call).

We log two judge scores computed by the prompts shown earlier in this appendix (warrant judge: Fig. 15; follow judge: Fig. 16):

$$u_i := J_{\text{warrant}}(q_i, \tilde{s}_i) \in [0,1], \qquad f_i := J_{\text{follow}}(g_i, \Delta_i(\tau)) \in [0,1].$$

In the logged trajectories, these scores appear as per-call numeric fields (warrant: `expert_score`; follow: `expert_follow_score`) that the reward function consumes directly. Note that online follow judging stores the follow score on the *next* expert call (evaluating adherence to the previous guidance), so the final guidance may optionally contribute an additional terminal follow score computed post-hoc.

Human validation of the judges. We developed both judge prompts iteratively, manually inspecting judge outputs until they met our quality bar. To validate the final judges, we sampled 140 warrant events and 160 follow events uniformly from Phase II rollouts and labeled them manually against the rubrics in Figs. 15–16, blind to the judge's score. Agreement with the judge is 84% (κ=0.66) for warrant and 89% (κ=0.77) for follow, with inter-annotator agreement of κ=0.81. The residual disagreements (16% of warrant and 11% of follow events) affect only the training reward, not the reported metric, as the first property below makes explicit. Three further properties bound the risk of reward hacking. First, the reported metric is judge-independent: judge scores shape training only, while SWE-bench Verified Pass@1 is computed by unit tests (App. E.1); a policy that games the judges would show judge scores rising while Pass@1 stagnates, whereas we observe Pass@1 improving across all expert backends as total tokens fall by ≈40%. Second, the follow judge asks a largely factual question (did the agent execute the instructed action and report back?) and assigns `null` when there is no clear instruction to follow; reporting a negative result counts as following through, so the reward does not require agreement with the expert. Judge outputs are hidden from the agent throughout. Third, the training dynamics do not show the signature of warrant-judge gaming: mean expert reward stays within 0.82–0.88 across all 160 steps (App. B) rather than climbing.

Invocation quality (R_{expert}). We convert the per-call warrant scores $\{u_i\}$ into a single trajectory-level term that discourages low-value escalation while keeping scale stable as the number of expert calls N varies:

$$\phi(u) = \begin{cases} u, & u \geq \tau_{\text{low}}, \\ p_{\text{low}}, & u < \tau_{\text{low}}, \end{cases} \qquad p_{\text{low}} \leq 0, \qquad R_{\text{warrant}}(\tau) = \text{Agg}\big(\{\phi(u_i)\}_{i=1}^{N}\big),$$

where Agg is typically mean (or min for stricter budgeting). Here, u is a dummy variable used to define the scalar transform $\phi(\cdot)$, which is then applied to each u_i.

To prevent expert spam, let n_{b2b} be the number of back-to-back expert calls. We add a capped penalty:

$$R_{\text{b2b}}(\tau) = \max(-1,\ \lambda_{\text{b2b}}\, n_{\text{b2b}}), \qquad \lambda_{\text{b2b}} \leq 0.$$

Optionally, with an expert budget Q, let $n_{\text{over}} = \max(0, N - Q)$ and $R_{\text{quota}}(\tau) = p_{\text{over}}\, n_{\text{over}}$ with $p_{\text{over}} \leq 0$. We do *not* enable this quota term in our runs; it is included for completeness. We combine:

$$R_{\text{expert}}(\tau) = R_{\text{warrant}}(\tau) + R_{\text{b2b}}(\tau) + \mathbb{1}[\text{quota enabled}]\, R_{\text{quota}}(\tau).$$

Follow-through (R_{follow}). We shape follow-through similarly:

$$\psi(f) = \begin{cases} f, & f \geq \tau_{\text{follow}}, \\ p_{\text{follow-low}}, & f < \tau_{\text{follow}}, \end{cases} \qquad p_{\text{follow-low}} \leq 0, \qquad R_{\text{follow}}(\tau) = \text{Agg}\big(\{\psi(f_i)\}_{i \in \mathcal{I}}\big),$$

where $\mathcal{I}$ indexes events for which follow-through is defined (e.g., excluding terminal calls).

E.4 Gating Functions

We gate only the auxiliary term R_{other} to prevent correctness or similarity from compensating for pathological behavior; loop and follow-through terms are always applied:

$$g_{\text{loop}}(\tau) = \begin{cases} 0, & R_{\text{loop}}(\tau) \le a_2, \\ 0.5, & R_{\text{loop}}(\tau) \le a_1, \\ 1, & \text{otherwise}, \end{cases} \qquad g_{\text{follow}}(\tau) = \begin{cases} 0, & R_{\text{follow}}(\tau) \le b_2, \\ 0.5, & R_{\text{follow}}(\tau) \le b_1, \\ 1, & \text{otherwise}, \end{cases}$$

for thresholds $a_2 < a_1 \le 0$ and $b_2 < b_1 \le 0$. The main text uses $R_{\text{total}} = R_{\text{loop}} + w_{\text{follow}} R_{\text{follow}} + g_{\text{loop}} g_{\text{follow}} R_{\text{other}}$ (Eq. 2). Notably, R_{follow} is never gated out; gating applies solely to R_{other}, while loop and follow-through penalties remain active throughout training. In our reward-shaping schedule below, we apply the follow gate only in Stage II by setting $g_{\text{follow}} \equiv 1$ during Stage I (i.e., the follow gate is effectively inactive in Stage I).

E.5 Reward Shaping Schedule

While Phase I SFT teaches the SLM to imitate expert interaction patterns, it does not reliably induce the behaviors required for effective collaboration. In particular, we observe two persistent failure modes: (i) the model fails to escalate when progress stalls, leading to long degenerative loops; and (ii) even when advice is obtained, the model often fails to follow it or to report back appropriately. We therefore apply a two-stage shaping curriculum that progressively tightens constraints to first suppress stalling and then enforce follow-through.

Stage I: Loop aggressive shaping (escalation induction). We make stalling strongly suboptimal by (i) increasing loop/stall penalty magnitude via a more negative cap c_{loop} and/or larger λ_{loop} in Eq. 4, and (ii) setting the loop gate g_{loop} to downweight R_{other} more aggressively. In this stage, we keep follow-through shaping mild: w_{follow} is small and we keep the follow gate inactive (i.e., $g_{\text{follow}} \equiv 1$). This reliably shifts the policy from "repeat failed actions" to "seek help when stuck."

Stage II: Loop+follow aggressive shaping (pair programming). After stalling is largely suppressed, we increase the strength of follow-through shaping by (i) increasing w_{follow}, (ii) setting $p_{\text{follow-low}}$ more negative, and/or (iii) making g_{follow} more aggressive (lower thresholds b_1, b_2), so that failing to execute and report back on expert guidance becomes strongly suboptimal. In addition, Stage II enforces a hard no-expert gate: trajectories with zero expert calls incur a fixed negative penalty, making unguided execution explicitly suboptimal. This stage converts one-shot escalation into multi-turn collaboration.

Reporting. In §4, we report ablations over this schedule (Stage I only vs. Stage I+II) and show that Stage I primarily reduces stagnation while Stage II improves expert-guidance adherence and end-to-end solve rate. The stall-penalty and shaping-schedule hyperparameters are given in §3.1. The remaining values are $(a_1, a_2) = (-1.0, -2.0)$, $(b_1, b_2) = (-0.5, -1.0)$, $(\tau_{\text{low}}, p_{\text{low}}) = (0.5, -1.0)$, and $\lambda_{\text{b2b}} = -0.1$ (with R_{b2b} capped at -1); we use mean aggregation for both Agg terms and $w_{\text{sim}} = 1.0$.

F Example Trajectory Snippets

For completeness, we include partial, representative trajectory snippets (left to right) corresponding to the three examples in Fig. 1(b) for the SWE-bench Verified instance `sympy__sympy-14976`. In the first example (SWE-agent-LM-7B; Fig. 17), the SLM stalls by repeatedly reading different segments of `lambdify.py`, forming tentative hypotheses, and continuing localization until it aborts due to runtime limits (we omit the remaining steps for brevity). In the second example (SWE-Protégé-7B P1, Sonnet 3.7; Fig. 18), the SLM requests expert help but fails to follow the expert's instruction to focus on the `lambdify` function in `/testbed/sympy/utilities/lambdify.py` (though it manages to follow the suggestion to inspect `MPMATH_TRANSLATIONS`). It then stalls by repeatedly issuing similar commands to view different portions of the `Rational` class, rather than re-engaging the expert. Finally, in the third example (SWE-Protégé-7B P2, Sonnet 3.7; Fig. 19), the SLM follows the expert's instructions end-to-end, reports progress back, and ultimately converges to the correct fault localization in `pycode.py` via this collaboration.

(a) SWE-agent-7B-LM. Instance `sympy__sympy-14976`.

Let's continue reading the function:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 250 300`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
250 values or numpy arrays as arguments.  However, in some cases
251 the generated function relies on the input being a numpy array:
252
253 >> from sympy import Piecewise
[Additional output omitted for brevity.]
```

Let's continue reading the function:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 300 350`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
300 Matrix([[1, 3]])
301
302 Tuple arguments are handled and the lambdified function should
303 be called with the same type of arguments as were used to create
[Additional output omitted for brevity.]
```

Let's continue reading the function:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 350 400`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
350 array([[ 1., 2.],
351 [ 3., 4.]], dtype=float32)
352
353 """
[Additional output omitted for brevity.]
```

Let's continue reading the function:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 400 450`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
400 from sympy.printing.pycode import MpmathPrinter as Printer
401 elif _module_present('numpy', namespaces):
402 from sympy.printing.pycode import NumPyPrinter as Printer
403 elif _module_present('numexpr', namespaces):
[Additional output omitted for brevity.]
```

Let's continue reading the function:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 450 500`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
450
451 funcname = '_lambdifygenerated'
```

```
452
453 if _module_present('tensorflow', namespaces):
[Additional output omitted for brevity.]
```

Now let's look at the '_TensorflowEvaluatorPrinter' and '_EvaluatorPrinter' classes which are responsible for generating the code for different modules:
str_replace_editor view /testbed/sympy/printing/lambdarepr.py

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/lambdarepr.py:
1 from __future__ import print_function, division
2 from distutils.version import LooseVersion as V
3
4 from .str import StrPrinter
[Additional output omitted for brevity.]
```

Let's also look at the PycodePrinter class which is likely responsible for generating the code for the 'mpmath' module:
str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 1 50

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
1 """
2 Python code printers
3
4 This module contains python code printers for plain python as well as NumPy &
SciPy enabled code.
[Additional output omitted for brevity.]
```

Let's look at the MpmathPrinter class which is specifically for the 'mpmath' module:
grep -r "MpmathPrinter" /testbed

```
/testbed/sympy/printing/pycode.py:class MpmathPrinter(PythonCodePrinter):
/testbed/sympy/printing/pycode.py:for k in MpmathPrinter._kf:
/testbed/sympy/printing/pycode.py:  setattr(MpmathPrinter, '_print_%s' % k,
_print_known_func)
/testbed/sympy/printing/pycode.py:  setattr(MpmathPrinter, '_print_%s' % k,
_print_known_const)
/testbed/sympy/printing/tests/test_pycode.py:  MpmathPrinter, NumPyPrinter,
PythonCodePrinter, pycode, SciPyPrinter
[Additional output omitted for brevity.]
```

Let's examine the MpmathPrinter class:
grep -A 50 "class MpmathPrinter" /testbed/sympy/printing/pycode.py

```
class MpmathPrinter(PythonCodePrinter):
"""
Lambda printer for mpmath which maintains precision for floats
"""
printmethod = "_mpmathcode"
[Additional output omitted for brevity.]
```

Let's also look at the '_known_functions_mpmath' dictionary:
grep -A 20 "_known_functions_mpmath" /testbed/sympy/printing/pycode.py

```
_known_functions_mpmath = dict(_in_mpmath, **{
'sign':  'sign',
})
_known_constants_mpmath = {
'Pi':  'pi'
[Additional output omitted for brevity.]
```

Let's see where '_known_functions_mpmath' is defined:
grep -r "_known_functions_mpmath =" /testbed

```
/testbed/sympy/printing/pycode.py:_known_functions_mpmath = dict(_in_mpmath, **{
```

Let's see what '_in_mpmath' contains:
`grep -A 20 "_in_mpmath" /testbed/sympy/printing/pycode.py`

```
_not_in_mpmath = 'log1p log2'.split()
_in_mpmath = [(k, v) for k, v in _known_functions_math.items() if k not in
_not_in_mpmath]
_known_functions_mpmath = dict(_in_mpmath, **{
'sign':  'sign',
})
[Additional output omitted for brevity.]
```

Let's see where '_known_functions_math' is defined:
`grep -r "_known_functions_math =" /testbed`

```
/testbed/sympy/printing/pycode.py:_known_functions_math = {
```

Let's see the definition of '_known_functions_math':
`str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 1 50`

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
1 """
2 Python code printers
3
4 This module contains python code printers for plain python as well as NumPy &
SciPy enabled code.
[Additional output omitted for brevity.]
```

Let's continue reading the definition of '_known_functions_math':
`str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 50 100`

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
50 'log2':  'log2',
51 'sin':  'sin',
52 'sinh':  'sinh',
53 'Sqrt':  'sqrt',
[Additional output omitted for brevity.]
```

Let's continue reading the PythonCodePrinter class:
`str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 100 150`

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
100 super(PythonCodePrinter, self).__init__(settings)
101 self.module_imports = defaultdict(set)
102 self.known_functions = dict(self._kf, **(settings or {}).get(
103 'user_functions', {}))
[Additional output omitted for brevity.]
```

Figure 17: Example trajectory snippet from SWE-agent-7B-LM, instance `sympy__sympy-14976`. We omit repeated runtime warnings and truncate long tool outputs for readability.

(b) SWE-Protégé-7B P1 (Sonnet 3.7). Instance `sympy__sympy-14976`.

```
ask_expert_llm 'Based on the PR description, there seems to be an issue with
mpmath not properly handling rational numbers in lambdified expressions.  Looking
at the code in /testbed/sympy/utilities/lambdify.py, where should I focus my
attention to fix this issue?'
```

```
<expert_llm_guidance>
The issue is in how rational numbers are handled when lambdifying expressions
with the mpmath module.  Based on the PR description, when a rational number like
'S(1)/3' appears in an expression that's lambdified with 'modules='mpmath'', it
```

```
remains as a Python fraction (‘232/3‘) rather than being converted to an mpmath
object with full precision.

You should focus on these specific areas:

1.  The ‘lambdify‘ function in ‘/testbed/sympy/utilities/lambdify.py‘ (around
line 166)
2.  The ‘MPMATH_TRANSLATIONS‘ dictionary (around line 45)

The key issue is that when rational numbers appear in expressions lambdified
with mpmath, they need to be converted to mpmath’s high-precision representation
rather than remaining as Python fractions.

The solution likely involves modifying how rational numbers are handled in the
code generation process for mpmath specifically.  You’ll need to ensure that
when the mpmath module is used, rational numbers like ‘S(1)/3‘ are converted
to ‘mpmath.mpf(1)/mpmath.mpf(3)‘ or an equivalent mpmath representation that
preserves full precision.

This would fix the issue where ‘nsolve‘ gets reduced precision results because
the rational constant isn’t evaluated at full precision.
</expert_llm_guidance>
```

Based on the expert's guidance, I need to focus on how rational numbers are handled when lambdifying expressions with the mpmath module. Let's examine the relevant parts of the code:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 45 71`

```
Here’s the result of running ‘cat -n‘ on /testbed/sympy/utilities/lambdify.py:
45 MPMATH_TRANSLATIONS = {
46 "Abs":  "fabs",
47 "elliptic_k":  "ellipk",
48 "elliptic_f":  "ellipf",
[Additional output omitted for brevity.]
```

Now let's look at how rational numbers are handled in general:
`grep -r "S\(" /testbed/sympy/core/`

```
grep:  Unmatched ( or \(
```

Let's try a different approach:
`grep -r "S(" /testbed/sympy/core/`

```
/testbed/sympy/core/numbers.py:  >> mod_inverse(S(2)/7, S(5)/2)
/testbed/sympy/core/numbers.py:  return (S.NegativeOne**num[0]*
    num[1]*S(2)**num[2])

/testbed/sympy/core/numbers.py:  >> S('.[3]') # repeating digits in brackets
/testbed/sympy/core/numbers.py:  >> S('3**2/10') # general expressions
/testbed/sympy/core/numbers.py:  >> (S(-3)/2).as_content_primitive()
[Additional output omitted for brevity.]
```

Let's look for files related to rational numbers:
`find /testbed -type f -name "*.py" | grep -i "number" | head -10`

```
/testbed/sympy/polys/numberfields.py
/testbed/sympy/polys/tests/test_numberfields.py
/testbed/sympy/functions/combinatorial/numbers.py
/testbed/sympy/functions/combinatorial/tests/test_comb_numbers.py
/testbed/sympy/core/numbers.py
[Additional output omitted for brevity.]
```

Let's examine the ‘numbers.py‘ file which likely contains the rational number implementation:
`str_replace_editor view /testbed/sympy/core/numbers.py -view_range 1 50`

```
Here's the result of running 'cat -n' on /testbed/sympy/core/numbers.py:
1 from __future__ import print_function, division
2
3 import decimal
4 import fractions
[Additional output omitted for brevity.]
```

Let's look at the rational number implementation:
`grep -n "class Rational" /testbed/sympy/core/numbers.py`

```
1388:class Rational(Number):
2469:class RationalConstant(Rational):
```

Let's look at the Rational class:
`str_replace_editor view /testbed/sympy/core/numbers.py -view_range 1388 1450`

```
Here's the result of running 'cat -n' on /testbed/sympy/core/numbers.py:
1388 class Rational(Number):
1389 """Represents rational numbers (p/q) of any size.
1390
1391 Examples
[Additional output omitted for brevity.]
```

Let's continue looking at the Rational class:
`str_replace_editor view /testbed/sympy/core/numbers.py -view_range 1450 1500`

```
Here's the result of running 'cat -n' on /testbed/sympy/core/numbers.py:
1450
1451 Access numerator and denominator as .p and .q:
1452
1453 >> r = Rational(3, 4)
[Additional output omitted for brevity.]
```

Let's continue looking at the Rational class:
`str_replace_editor view /testbed/sympy/core/numbers.py -view_range 1500 1550`

```
Here's the result of running 'cat -n' on /testbed/sympy/core/numbers.py:
1500 p = p.replace(' ', '')
1501 pq = p.rsplit('/', 1)
1502 if len(pq) == 2:
1503 p, q = pq
[Additional output omitted for brevity.]
```

Let's continue looking at the Rational class:
`str_replace_editor view /testbed/sympy/core/numbers.py -view_range 1550 1600`

```
Here's the result of running 'cat -n' on /testbed/sympy/core/numbers.py:
1550 obj = Expr.__new__(cls)
1551 obj.p = p
1552 obj.q = q
1553 return obj
[Additional output omitted for brevity.]
```

Figure 18: Example trajectory snippet from SWE-Protégé-7B P1 (Sonnet 3.7), instance `sympy__sympy-14976`. We show the agent thought and the corresponding tool output for each thought/action pair; long tool outputs are truncated, while expert guidance blocks are shown in full.

(c) SWE-Protégé-7B P2 (Sonnet 3.7). Instance `sympy__sympy-14976_0`.

Let's look at the specific part of the code that handles the 'mpmath' module, since that's where the issue is occurring according to the PR description:
`ask_expert_llm 'In the Sympy lambdify.py file, where would the handling of rational numbers specifically for the '"'"'mpmath'"'"' module be`

```
implemented?  I’"’"’m looking for the code that processes expressions when using
modules=’"’"’mpmath’"’"’ and why rationals aren’"’"’t being properly wrapped.’
```

```
<expert_llm_guidance>
You need to look at how rational numbers are handled in the lambdify function.
The issue is that when using the ’mpmath’ module, rational numbers like 232/3
aren’t being converted to mpmath’s high-precision format.

First, examine the lambdify function implementation in detail, located in:
/testbed/sympy/utilities/lambdify.py

Then look at how the printer handles different types of expressions, also located
in:
/testbed/sympy/utilities/lambdify.py

The issue is likely in how rational numbers are printed when the mpmath module is
selected.  Look for code that handles Rational objects or fractions, and check if
there’s special handling for the mpmath module that’s missing.  You’ll probably
need to modify the _EvaluatorPrinter class to ensure that rational numbers are
converted to mpmath.mpf() calls when the mpmath module is used.
</expert_llm_guidance>
```

Let me try again with the correct format:

```
str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 704 744
```

```
Here’s the result of running ‘cat -n‘ on /testbed/sympy/utilities/lambdify.py:
704 def _preprocess(self, args, expr):
705 """Preprocess args, expr to replace arguments that do not map
706 to valid Python identifiers.
707
[Additional output omitted for brevity.]
```

Now let’s look at how subexpressions are processed, which is likely where the rational numbers should be handled:

```
str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 746 763
```

```
Here’s the result of running ‘cat -n‘ on /testbed/sympy/utilities/lambdify.py:
746 def _subexpr(self, expr, dummies_dict):
747 from sympy.matrices import DeferredVector
748 from sympy import sympify
749
[Additional output omitted for brevity.]
```

Let’s also check how functions are printed, particularly focusing on the mpmath module:

```
str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 654 688
```

```
Here’s the result of running ‘cat -n‘ on /testbed/sympy/utilities/lambdify.py:
654 def doprint(self, funcname, args, expr):
655 """Returns the function definition code as a string."""
656 from sympy import Dummy
657
[Additional output omitted for brevity.]
```

Now, let’s create a script to reproduce the issue mentioned in the PR description:

```
str_replace_editor create /testbed/reproduce_issue.py -file_text ’#!/usr/bin/env
python3
from sympy import symbols, Eq, rf, S, Float, nsolve
import inspect
# Define the variable
x = symbols(’"’"’x’"’"’)
# Create the equation from the PR description
eqn = Eq(rf(18, x), 77 + S(1)/3)
# Print the equation
print("Original equation:", eqn)
```

```
# Lambdify the expression with mpmath module
f = lambda x:  eval(inspect.getsource(lambda y:  (rf(18, y) - (77 +
S(1)/3)).evalf())[10:-2])
print("
textbackslashnLambda function source:")
print(inspect.getsource(f))
# Test with nsolve
x0 = nsolve(eqn, Float('"'"'1.5'"'"', 64), prec=64)
print("
textbackslashnSolution x0:", x0)
print("rf(18, x0).evalf(64):", rf(18, x0).evalf(64))
print("Expected:
textasciitilde77.3333333333333")'
```

```
File created successfully at:  /testbed/reproduce_issue.py
```

Let's run the script to see the issue:
`cd /testbed && python reproduce_issue.py`

```
/testbed/sympy/solvers/diophantine.py:3188:  SyntaxWarning:  "is" with a literal.
Did you mean "=="?
if feasible is 1:  # it's prime and k == 2
Original equation:  Eq(RisingFactorial(18, x), 232/3)
Lambda function source:
f = lambda x:  eval(inspect.getsource(lambda y:  (rf(18, y) - (77 +
S(1)/3)).evalf())[10:-2])
[Additional output omitted for brevity.]
```

Now I understand the issue. When using 'lambdify(modules='mpmath')', the rational number '232/3' is not being properly wrapped in mpmath's high-precision format. Let's ask the expert for guidance on how to fix this:
`ask_expert_llm 'I'"'"'ve identified that the issue is in the _EvaluatorPrinter class in lambdify.py. When using modules='"'"'mpmath'"'"', rational numbers like 232/3 aren'"'"'t being converted to mpmath.mpf() calls. How should I modify the code to ensure that rational numbers are properly wrapped in mpmath.mpf() when the mpmath module is selected?'`

```
<expert_llm_guidance>
You're on the right track.  The issue is that when using the mpmath module,
rational numbers need to be wrapped in mpmath.mpf() calls to maintain precision.

Let's look at the _EvaluatorPrinter class which handles how expressions are
printed in the lambdify function:

1.  First, check the _EvaluatorPrinter class implementation, located in:
/testbed/sympy/utilities/lambdify.py

2.  Then look for how rational numbers are handled in the printer.  There should
be a method like _print_Rational or similar.

3.  The fix will likely involve modifying how Rational objects are printed when
the module is 'mpmath'.  You'll need to wrap the rational in mpmath.mpf() instead
of just using Python's division operator.

After identifying the specific method, modify it to check if the module is
'mpmath' and if so, wrap the rational in mpmath.mpf().
</expert_llm_guidance>
```

Let's examine the _EvaluatorPrinter class more closely to find how rational numbers are handled:
`str_replace_editor view /testbed/sympy/utilities/lambdify.py -view_range 400 450`

```
Here's the result of running 'cat -n' on /testbed/sympy/utilities/lambdify.py:
400 from sympy.printing.pycode import MpmathPrinter as Printer
401 elif _module_present('numpy', namespaces):
```

```
402 from sympy.printing.pycode import NumPyPrinter as Printer
403 elif _module_present('numexpr', namespaces):
[Additional output omitted for brevity.]
```

Now let's look for how rational numbers are handled in the printers. Based on the PR description, we need to focus on the '_EvaluatorPrinter' class which seems to handle the actual printing of expressions for evaluation:
`grep -r "_print_Rational" /testbed/sympy/`

```
/testbed/sympy/polys/numberfields.py:  def _print_Rational(self, expr):
/testbed/sympy/polys/numberfields.py:  return "mpi('
grep:  /testbed/sympy/polys/__pycache__/numberfields.cpython-39.pyc:  binary file
matches
/testbed/sympy/printing/ccode.py:  def _print_Rational(self, expr):
/testbed/sympy/printing/mathml.py:  def _print_Rational(self, e):
[Additional output omitted for brevity.]
```

Let's check if there's a specific implementation of '_EvaluatorPrinter' in the lambdify module:
`grep -A 10 "_EvaluatorPrinter" /testbed/sympy/utilities/lambdify.py`

```
funcprinter = _EvaluatorPrinter(printer, dummify)
funcstr = funcprinter.doprint(funcname, args, expr)
funclocals = {}
global _lambdify_generated_counter
filename = '<lambdifygenerated-%s>' % _lambdify_generated_counter
[Additional output omitted for brevity.]
```

Now I'll look at the '_print_Rational' method in the '_EvaluatorPrinter' class. Since we don't see it explicitly defined there, it likely inherits it from the parent class. Let's check the base class:
`ask_expert_llm 'The _EvaluatorPrinter class in lambdify.py doesn'"'"'t seem to have a _print_Rational method defined.  It inherits from LambdaPrinter.  Where should I look for the implementation of how rational numbers are handled during lambdification, especially for the mpmath module case?'`

```
<expert_llm_guidance>
You need to look at the MpmathPrinter class since that's what's used when
modules='mpmath'.  Let's examine the implementation:

1.  First, check the MpmathPrinter in sympy/printing/pycode.py:

2.  Then look for the _print_Rational method in this class or its parent classes.
If MpmathPrinter doesn't override _print_Rational, check its parent class (likely
PythonCodePrinter):

3.  The issue is likely that the _print_Rational method in PythonCodePrinter (or
whatever parent class) is not properly wrapping rational numbers in mpmath.mpf()
when used with the mpmath module.  You'll need to modify the MpmathPrinter class
to override _print_Rational to use mpmath.mpf() for proper precision.
</expert_llm_guidance>
```

Based on the expert's advice, I need to check the MpmathPrinter class in the pycode.py file:
`str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 1 100`

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
1 """
2 Python code printers
3
4 This module contains python code printers for plain python as well as NumPy &
SciPy enabled code.
[Additional output omitted for brevity.]
```

Let's search for the MpmathPrinter class and the _print_Rational method:
`grep -n "class MpmathPrinter" /testbed/sympy/printing/pycode.py`

```
314:class MpmathPrinter(PythonCodePrinter):
```

(no thought logged)
grep -n "_print_Rational" -A 5 /testbed/sympy/printing/pycode.py | grep -A 5 MpmathPrinter

```
Your command ran successfully and did not produce any output.
```

Let's look at the MpmathPrinter class implementation:
str_replace_editor view /testbed/sympy/printing/pycode.py -view_range 314 350

```
Here's the result of running 'cat -n' on /testbed/sympy/printing/pycode.py:
314 class MpmathPrinter(PythonCodePrinter):
315 """
316 Lambda printer for mpmath which maintains precision for floats
317 """
[Additional output omitted for brevity.]
```

Figure 19: Example trajectory snippet from SWE-Protégé-7B P2 (Sonnet 3.7), instance `sympy__sympy-14976_0`. We omit repeated runtime warnings, truncate long tool outputs, and show expert guidance blocks in full.

G Broader Impacts

SWE-Protégé aims to make long-horizon agentic systems more practical by enabling small language models to remain the primary decision-makers while selectively collaborating with stronger expert models when progress stalls. This design targets a more economical deployment model for agentic AI, where large models are used sparingly rather than continuously. In software engineering workflows, this can reduce latency and compute costs, and may make agentic assistance more accessible in settings where infrastructure or budget constraints preclude always-on frontier models.

SWE-Protégé is not intended to replace human developers or engineering judgment. Instead, it supports a layered assistance paradigm in which a small model handles routine exploration and tool use, while expert models are invoked selectively to overcome uncertainty or stagnation. This mirrors common practice in production engineering, where automated systems rely on fallbacks, escalation paths, and checks before changes are merged or deployed.

Like other LLM-based code agents, SWE-Protégé inherits risks associated with incorrect, misleading, or incomplete outputs. In our setting, these risks include improper deferral decisions (e.g., escalating too late, too frequently, or inappropriately) and propagating incorrect guidance from the expert model. Misguided edits could introduce defects, regressions, or security vulnerabilities if deployed without adequate review. We therefore recommend deploying SWE-Protégé with automated verification (e.g., unit tests, linters, static analysis) and human oversight, particularly in safety-critical or production environments.

As with other LLM-based code agents, SWE-Protégé is subject to biases arising from benchmark composition, training data, and model interaction dynamics. Our evaluation focuses primarily on Python-based repositories, reflecting SWE-bench's task composition. To mitigate expert interaction bias, expert models are treated as fallible collaborators rather than oracles: they have no access to ground-truth patches or tests during evaluation/inference, and the SLM remains the principal decision-maker. We further mitigate hidden biases through the use of public datasets (e.g., SWE-smith), standard evaluation protocols (i.e., from SWE-bench), and transparent reporting of training procedures, reward design, and limitations, which we explicitly document to avoid overgeneralization of results.

Our experiments and analyses are conducted on open-source repositories and public benchmarks. Lowering the cost of code generation and repair may increase the volume of automated changes, which can amplify downstream risks if used irresponsibly. Careful access control, auditability, and review processes remain important. Except for the SWE-agent-LM 7B and 32B variants and the Claude Sonnet 3.7, Sonnet 4.5, and Opus 4.1 models, which we evaluate as baselines or experts in this work, all other entries in Table 1 are taken directly from (Yang et al., 2025b) and were not evaluated by us.

Overall, we view SWE-Protégé as a step toward more responsible and sustainable agentic systems: ones that use scale selectively, incorporate explicit escalation mechanisms, and emphasize controlled interaction with stronger models rather than ubiquitous reliance on them.

Asset	Source/version used	License or terms
SWE-smith tasks and trajectories	SWE-smith task dataset and trajectories (SWE-bench, 2026a,b; Yang et al., 2025b); training trajectories sampled to ≈4.8K accepted tasks	MIT License for the SWE-smith released assets. The dataset is derived from public GitHub repositories; SWE-smith lists the upstream repository licenses exhaustively in its Table 6, reporting that the licenses allow the public, nonprofit use exercised in that work.
SWE-bench Verified	500-instance SWE-bench Verified test split (Chowdhury et al., 2024)	SWE-bench code/data are released under the MIT License; underlying task repositories retain their original upstream licenses.
SWE-Gym subset	100-task subset drawn from SWE-Gym/SWE-Gym and bundled in SkyRL (SumanthRH, 2026; Pan et al., 2024)	Apache License 2.0 for SWE-Gym. Underlying task repositories retain their original upstream licenses.
Qwen2.5-Coder-Instruct	Qwen2.5-Coder-Instruct 7B and 32B model checkpoints (Hui et al., 2024)	Apache License 2.0, as listed on the Hugging Face model cards.
SWE-agent	SWE-agent framework (Yang et al., 2024), augmented only with `ask_expert`	MIT License.
vLLM, torchtune, and SkyRL	vLLM for inference, torchtune for SFT (meta-pytorch, 2026), and SkyRL for RL (NovaSky-AI, 2026)	vLLM and SkyRL are Apache License 2.0; torchtune is BSD 3-Clause.
Claude expert models	Claude Sonnet 3.7 (Anthropic, 2025), Sonnet 4.5, and Opus 4.1 accessed through AWS Bedrock	Used through AWS Bedrock under AWS Bedrock and Anthropic commercial terms for third-party Bedrock models.

Table 4: Existing assets used by SWE-Protégé, with source attribution and license or terms of use.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and §1 state the paper's main claims about expert–protégé collaboration, sparse expert use, SWE-bench Verified performance, and cost. These claims are supported by the method description in §2, experimental setup in §3, and results in §4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses limitations and future work in §5, including the focus on SWE-bench/SWE-agent, limited exploration of hyperparameters and collaboration designs, treating the expert as a fixed black-box backend, the hybrid dependence on a frontier

expert, the train/inference gold-patch asymmetry, diminishing returns at 32B, and untested sensitivity to the RL task subset, reward heuristics, and alternative scaffolds.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not present theoretical results or formal proofs; its contributions are methodological and empirical.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper describes the agent system, models, datasets, evaluation protocol, SFT/RL settings, reward design, and expert-tool details in §3, §3.1, §2, App. C, App. D, and App. E.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The submission describes the data sources, training procedure, and implementation details needed to understand and partially reproduce the results, but due to institutional restrictions we are unable to publicly release the code, generated training data, or model checkpoints. To support reimplementation, the paper and appendix specify the prompts, reward functions, and all training hyperparameters.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.

- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The experimental setup in §3 and training details in §3.1 specify the benchmark, model variants, inference framework, rollout limits, SFT settings, RL algorithm, data subsets, rollout counts, batch size, number of steps, and reward shaping schedule.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The main comparison table reports standard errors over three runs (Table 1), and the appendix reports 95% bootstrap uncertainty bands with 400 resamples and a fixed random seed for looping diagnostics (App. B).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: §3 states that experiments used a single node with 8 NVIDIA A100/H100 80G GPUs, such as AWS `p4de.24xlarge`, for both training and inference.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research uses public software-engineering benchmarks and model APIs/checkpoints, does not involve human-subject experiments, and discusses risks and mitigations in App. G.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The paper discusses positive impacts, risks, and mitigations in App. G.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release new high-risk models, scraped datasets, or other high-risk assets with the submission. Deployment risks and recommended safeguards for use are discussed in App. G.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper credits the datasets, benchmarks, codebases, model checkpoints, and commercial expert-model services used in the experiments. Table 4 summarizes the relevant source, version or split, and license or terms of use for these assets; for SWE-smith-derived data, we rely on the exhaustive upstream repository license listing provided by SWE-smith.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

Justification: The paper does not release new assets.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing experiments or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing experiments or research with human subjects, so IRB approval or equivalent review is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are central to the method and experiments; §2, §3, App. C, and App. D describe the student model, expert models, expert invocation mechanism, and LLM-based judging used for reward shaping.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.