TEST-TIME ADAPTATION OF SCAFFOLDS FROM UNLABELED TRAJECTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

A scaffold is a workflow that orchestrates how a coding agent observes its environment, uses tools, and produces outputs. Existing scaffolds typically follow a uniform workflow across different models and tasks. In this study, we reveal that such uniform workflows may fail to accommodate diverse model capabilities and varying coding complexities. To address this limitation, we propose *AdaScaffold*, a test-time **ada**ptation framework that adapts **scaffold**s to diverse model capabilities and coding complexities. AdaScaffold uses unlabeled trajectories to estimate when to intervene and inject behavioral guidelines into the original workflow via two core modules: (i) *Intervention Timing Estimation*, which optimizes the timing of each intervention by maximizing the *empirical covariation* between agent mis-behaviors and risks; and (ii) *Guideline Generation*, which prompts the coding agent itself with the estimated timings and predefined mis-behaviors to synthesize model- and task-specific guidelines. Extensive evaluation demonstrates that AdaScaffold improves resolve rates across various model scales while reducing token consumption to 45%–85%. By grounding intervention timing in a closed-form statistical criterion over unlabeled trajectories, AdaScaffold offers an interpretable test-time scaffold-level adaptation mechanism for coding agents. Our code is available at: `https://anonymous.4open.science/r/AdaScaffold`.

1 INTRODUCTION

Large language models (LLMs) have rapidly become the foundation of automated software engineering (SWE), powering coding agents that can read repositories, edit source files, run tests, and submit patches in a closed interaction loop (Jimenez et al., 2024; Yang et al., 2024; Wang et al., 2025; Jain et al., 2025). A coding agent is not the LLM alone: an external *scaffold* decides what tools the model can call, how observations are formatted, how the trajectory is bounded by step, time, and token budgets, and when the trajectory is allowed to terminate.

As shown in Figure 1(a), in general modern systems this scaffold is a uniform workflow shared by every model and every task. Serving as the operational backbone for coding tasks, the scaffold guides the agent's workflow via structured prompts, manages environment interactions to return observations, and enables the LLM to invoke tools for retrieving, modifying, and executing code, ultimately capturing the outputs to produce the final generated patch (Wang et al., 2025).

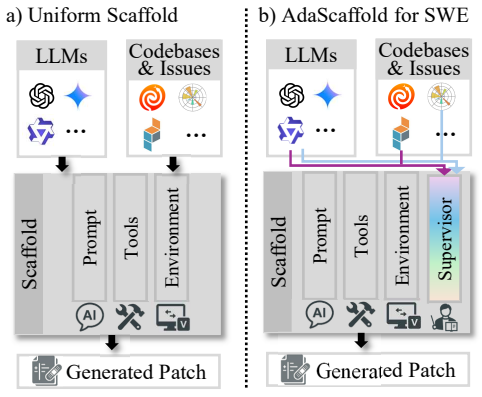

Figure 1: Uniform Scaffold vs. AdaScaffold.

However, this one-fits-all design is increasingly hard to justify. Figure 2 highlights two complementary failure modes of fixed scaffolds. On the model side (as shown in Figure 2(a)), smaller models exhaust the step budget on redundant exploration and time out before producing an edit, while larger models tend to over-verify an already-correct

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

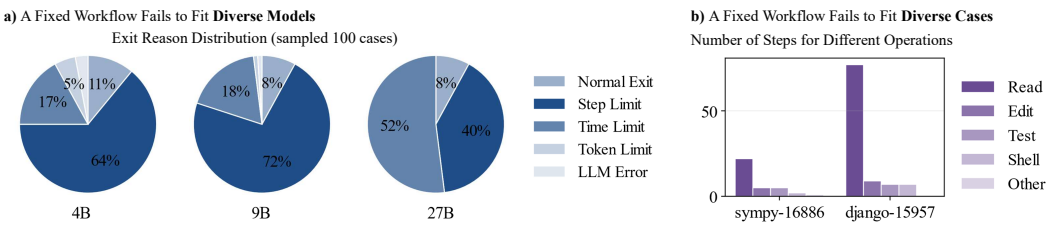

Figure 2: Limitations of Uniform Scaffolds: (a) cannot adapt to different models: smaller models tend to exceed step budgets, while larger models are more likely to incur excessive runtime, requiring different process control; and (b) cannot adapt to different tasks: some require only a few code-reading steps, whereas others spend most of the budget on code exploration.

patch and run out of wall-clock budget; the two regimes call for opposite kinds of process control. On the task side (as shown in Figure 2(b)), some issues are solved with only a handful of code-reading steps, whereas others legitimately need most of the budget for exploration before any edit is possible, so a fixed workflow cannot fit both. Motivated by these observations, we ask whether the scaffold itself can be made adaptive to the underlying model and the incoming task without retraining the LLM or relying on human intervention at test time. Our answer is AdaScaffold: an *adaptive scaffold* that assigns the agent an additional supervisor role to provide guidelines for common misbehaviors before tasks, estimates when these guidelines should be activated from unlabeled test-time trajectories, and steers the agent's reasoning through timely interventions with behavioral guideline injections. The difference introduced by AdaScaffold is shown in Figure 1.

In this paper, we first formalize the setting of coding tasks in Section 3. We begin by establishing a formulation for coding tasks using LLM coding agents, where we explicitly define the utility of the scaffold alongside the task objectives and evaluation protocols. Subsequently, we present the rationale behind the adaptive scaffold, followed by its rigorous formulation.

Section 4 details the method. AdaScaffold consists of two modules: (i) *Intervention Timing Estimation* estimates when each intervention should be activated by maximizing the *empirical covariation* between predefined agent mis-behaviors and their associated risks over unlabeled test-time trajectories. (ii) *Guideline Generation* prompts the coding agent itself with the estimated intervention timings and mis-behaviors to synthesize model- and task-specific natural-language guidelines. The generated guidelines are then injected into the original scaffold and deterministically activated at the estimated timings during evaluation, without introducing additional LLM calls at each step.

Section 5 presents results on SWE-Bench-Verified (Jimenez et al., 2024) using Qwen-3.5 (Yang et al., 2025) agents at 4B, 9B, and 27B, as well as a GLM-5.2 (Zeng et al., 2026) agent at 753B. We compare AdaScaffold against a uniform-scaffold baseline (i.e., original scaffold) and two test-time experience-based competitors, and conduct ablations of intervention timing estimation and guideline generation. We further study the transferability of estimated timings across trajectory pools, the effect of calibration sample size, and the quality of guidelines generated by the coding agent itself compared with those generated by stronger external supervisors, including GPT-5.5 (Singh et al., 2025) and Claude Opus 4.6 (Anthropic, 2025). AdaScaffold consistently improves resolve rates across different model scales while reducing token consumption to 45%–85%.

Beyond these empirical gains, AdaScaffold demonstrates that scaffold adaptation can be performed at test time using unlabeled agent trajectories, without requiring reward signals or additional training. In particular, intervention timing can be determined by a closed-form statistical criterion based on the empirical covariation between mis-behaviors and risks, while the coding agent itself can translate predefined mis-behaviors and estimated timings into interpretable behavioral guidelines. This provides a simple and interpretable mechanism for adapting an existing scaffold to different model capabilities and coding complexities, offering scaffold-level adaptation as a complementary axis for improving the reliability and efficiency of coding agents.

2 Related Work

Coding Agents and Scaffolds. Large language models have become central to code generation and software engineering. Codex showed that LLMs trained on code can solve natural-language

programming tasks (Chen et al., 2021), while AlphaCode demonstrated competition-level program synthesis (Li et al., 2022). CodeT5 studied unified pretraining for code understanding and generation (Wang et al., 2021b), and CodeGen explored multi-turn program synthesis (Nijkamp et al., 2023). Recent open code models such as StarCoder, Code Llama, and DeepSeek-Coder further improved code-specialized foundation models (Li et al., 2023; Roziere et al., 2023; Guo et al., 2024). Recent work has shifted from single-shot generation to interactive coding agents. SWE-bench introduced repository-level evaluation on real GitHub issues (Jimenez et al., 2024). SWE-agent emphasized the importance of agent-computer interfaces (Yang et al., 2024), and OpenHands provides an open platform for generalist software development agents (Wang et al., 2025). Agentless, MarsCode Agent, AutoCodeRover, and CodeAct study alternative repair pipelines, bug-fixing agents, autonomous program improvement, and executable actions (Xia et al., 2024; Liu et al., 2024; Zhang et al., 2024; Wang et al., 2024).

Prompting and scaffolding methods shape how LLMs reason and use tools. Chain-of-thought prompting improves reasoning through intermediate steps (Wei et al., 2022); self-consistency aggregates multiple reasoning paths (Wang et al., 2023); and least-to-most prompting decomposes complex tasks into simpler subproblems (Zhou et al., 2023). Toolformer studies tool use through self-supervised signals (Schick et al., 2023), while program-aided language models use executable programs as reasoning interfaces (Gao et al., 2023). Reflexion and Self-Refine further use feedback to revise agent behavior or outputs (Shinn et al., 2023; Madaan et al., 2023). For coding agents, scaffolds are important because the agent must manage exploration, editing, testing, and submission under limited steps and tokens. Prior work mainly improves models, interfaces, workflows for uniform scaffolds. AdaScaffold instead performs *offline, deterministic scaffold-level adaptation* from unlabeled trajectories, while keeping the agent fixed and requiring no extra LLM calls at each step.

Meta-Learning and Test-Time Adaptation. Meta-learning studies how models can adapt quickly to new tasks. MAML learns an initialization suitable for fast gradient adaptation (Finn et al., 2017); Matching Networks perform few-shot learning through metric comparison (Vinyals et al., 2016); and memory-augmented meta-learning uses external memory for rapid adaptation (Santoro et al., 2016). Later work analyzes whether MAML gains come from rapid learning or feature reuse (Raghu et al., 2020). In language modeling, GPT-3 popularized in-context learning, where models adapt through prompt context without parameter updates (Brown et al., 2020). These methods typically rely on training-time task distributions, support examples, gradient updates, or prompt demonstrations.

Test-time adaptation improves model behavior during evaluation using information available at test time. Test-time training adapts models with self-supervised objectives (Sun et al., 2020); TENT uses entropy minimization (Wang et al., 2021a); and TTT++ studies when self-supervised test-time training succeeds or fails (Liu et al., 2021). EATA, MEMO, and SAR improve efficiency, augmentation-based robustness, and stability under dynamic shifts (Niu et al., 2022; Zhang et al., 2022; Niu et al., 2023). Recent LLM work also explores self-generated reasoning or feedback, such as STaR and self-rewarding language models (Zelikman et al., 2022; Yuan et al., 2024). These methods adapt predictions, reasoning traces, or model parameters. In contrast, AdaScaffold performs test-time adaptation at the scaffold level: it estimates intervention timing from unlabeled coding trajectories, generates supervisor guidelines offline, and applies deterministic guideline injection during final agent evaluation.

3 Preliminary and Insights

Task and Scaffold. We consider a benchmark of n software engineering tasks $\mathcal{D} = \{(x_i, \mathcal{T}_i)\}_{i=1}^{n}$, where each task is defined as $x_i = (\mathcal{R}_i, q_i, \mathcal{E}_i)$. Here, $\mathcal{R}_i$ denotes the initial repository, q_i is the natural-language issue, $\mathcal{T}_i$ is the test set, and $\mathcal{E}_i$ is the execution environment. Given a patch Δ_i, task success is determined by $f_{\text{verify}}(\mathcal{R}_i \oplus \Delta_i, \mathcal{T}_i | \mathcal{E}_i) \in \{0, 1\}$, where f_{verify} denotes the verification procedure that executes the test set $\mathcal{T}_i$ in the environment $\mathcal{E}_i$ on the patched repository, $\oplus$ denotes applying the patch to the repository, and the value '1' implies that all test cases in $\mathcal{T}_i$ pass.

A scaffold $S \in \mathcal{S}$ specifies how an LLM is prompted, what tools it can use, and how model outputs are converted into actions. Together with an LLM parameterized by Θ_{LLM}, it induces an agent policy $\pi_{\Theta_{\text{LLM}}, S}$. At step t, o_t denotes the observation returned by the environment, such as file contents, command outputs, or test results, and a_t denotes the agent action, such as reading files, editing code, running commands, or submitting the final answer.

Table 1: Pre-defined Interventions for mis-behaviors.

	Intervention Description	Parameters
A	Break stuck loop; force a different action.	`min_repeats`
B	Soft nudge to make first edit.	`step_frac`
C	Hard push: demand first edit NOW.	`step_frac`
D	Write/verify checkpoint; branch on patch presence.	`step_frac`
E	Submit checkpoint; emit <function=submit>.	`step_frac`
F	Force stop if patch exists near budget end.	`step_frac`
G	Recover from consecutive errored observations.	`threshold`
H	Stop running tests without editing.	`threshold`

The agent conditions on the interaction history $h_t = (q_i, o_0, a_0, o_1, a_1, \ldots, o_t)$ and samples an action

$$a_t \sim \pi_{\Theta_{\mathrm{LLM}},S}(\cdot \mid h_t),$$

where

$$\pi_{\Theta_{\mathrm{LLM}},S}(a_t \mid h_t) = S(a_t \mid h_t, \Theta_{\mathrm{LLM}}).$$

After T interaction steps for sample x_i, the final patch is extracted from the terminal history as $\Delta_i = f_{\mathrm{extract}}(h_T)$.

It is necessary that $T \leq \hat{T}$, where $\hat{T}$ denotes the maximum steps. The benchmark performance is measured by the average verification success $\mathrm{Score}(\Theta_{\mathrm{LLM}}, S)$, where:

$$\mathrm{Score}(\Theta_{\mathrm{LLM}}, S) = \frac{1}{n}\sum_{i=1}^{n} f_{\mathrm{verify}}(\mathcal{R}_i \oplus \Delta_i, \mathcal{T}_i | \mathcal{E}_i). \tag{1}$$

Design of AdaScaffold. The above formulation treats the scaffold S as a fixed element in $\mathcal{S}$. However, a fixed workflow may fail to fit diverse LLMs with different capabilities, and may also fail to fit diverse SWE cases with different debugging, localization, and editing requirements, as shown in Section 1. This motivates an adaptive scaffold that fits itself to both the task and the model.

Formally, we define a scaffold generator Φ that maps a task-model pair to a task- and model-specific scaffold:

$$S_{\mathrm{ada}} \triangleq \Phi(x_i, \Theta_{\mathrm{LLM}}) \in \mathcal{S}.$$

The agent then acts according to the scaffold induced by this adaptive scaffold: $a_t \sim S_{\mathrm{ada}}(\cdot \mid h_t, \Theta_{\mathrm{LLM}})$.

However, obtaining such an adaptive scaffold without incurring excessive overhead remains challenging. In the following section, we describe how our method instantiates this idea by augmenting a standard uniform scaffold with model- and task-specific behavioral guidelines, whose activation timings are estimated from unlabeled test-time trajectories (i.e., without evaluating their outcomes or accessing test results), thereby enabling adaptive scaffold behavior with minimal additional cost.

4 METHOD

Test-Time AdaScaffold. We consider a test-time setting where only an available subset of instances $\mathcal{D}_{\mathrm{sub}} = \{x_i\}_{i=1}^{m}$ is given. This construction does not use the hidden test signal $\mathcal{T}_i$. For each instance, we are given a fixed model Θ_{LLM} and a uniform scaffold S_{fix}. We first run Θ_{LLM} with S_{fix} on $\mathcal{D}_{\mathrm{sub}}$ and collect the resulting trajectories $[h_T]_i = (q_i, o_0, a_0, \ldots, o_T)$, with the exit reason $e([h_T]_i)$. The instances in $\mathcal{D}_{\mathrm{sub}}$ may be drawn from the final evaluation set; throughout adaptation, we use only their task inputs and unlabeled agent trajectories, and do not access verifier outcomes or hidden tests until all adapted rollouts are completed.

AdaScaffold uses these observed trajectories to construct a guideline-augmented scaffold. It keeps the original scaffold unchanged and adds a intervention guideline set generated from test-time behavior:

$$S_{\mathrm{ada}} = S_{\mathrm{fix}} \oplus f_{\mathrm{guideline}}(\mathcal{D}_{\mathrm{sub}}, \Theta_{\mathrm{LLM}}). \tag{2}$$

We then evaluate the same model Θ_{LLM} under S_{ada} and report $\mathrm{Score}(\Theta_{\mathrm{LLM}}, S_{\mathrm{ada}})$ as the final result. Thus, the method adapts the scaffold at test time without updating model parameters and without accessing ground-truth tests during guideline construction.

Intervention Timing Estimation. We define a family of candidate interventions I_θ for mis-behaviors, where each intervention type is listed in Table 1 (with more details in Appendix A.1) and θ denotes its timing parameter. For a trajectory h_T, the intervention indicator is

$$I_\theta(h_T) \in \{0, 1\},$$

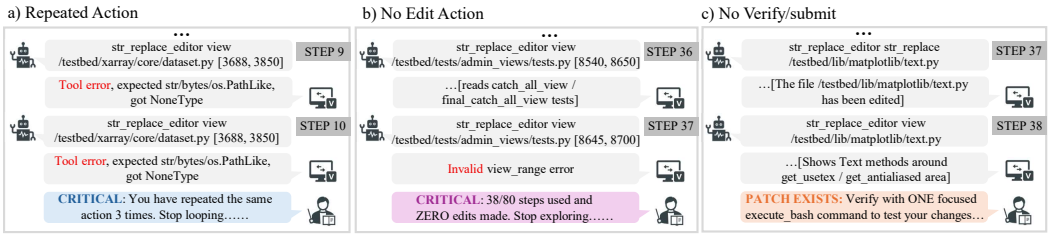

Figure 3: Three examples of supervisor interventions: a) repetitive-action guidelines break failed loops; b) delayed-editing guidelines drive exploration toward code changes; c) completion guidelines promote focused verification and timely submission. Guidelines are inserted immediately after each step's observation.

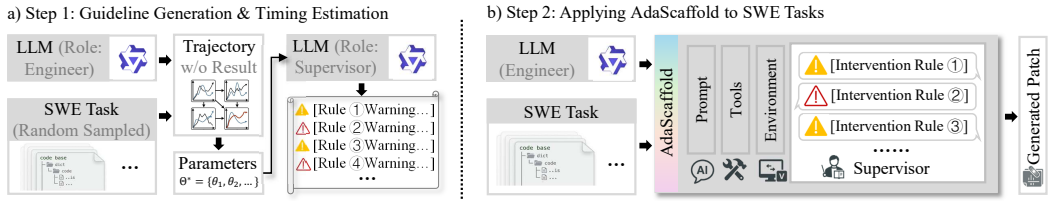

Figure 4: Overview of AdaScaffold. In the first step, the LLM runs with S_{fix} on a randomly sampled SWE task subset to collect unlabeled trajectories, which are used to estimate intervention timing and generate supervisor guidelines. In the second step, these guidelines are merged into S_{fix} to form S_{ada}, and the LLM is finally evaluated under our AdaScaffold.

where $I_\theta(h_T) = 1$ means that the intervention would be triggered on that trajectory, and $I_\theta(h_T) = 0$ otherwise.

We estimate intervention timing using only trajectory-level behavior. A trajectory is treated as risky if it terminates with an undesirable exit reason or if its length is close to the step budget:

$$Y(h_T) = 1 \Leftrightarrow e(h_T) \neq \texttt{Normal} \vee T \geq \rho \cdot \hat{T}.$$

Here $e(h_T)$ is the exit reason, T is the trajectory length, $\hat{T}$ is the maximum allowed step budget, and ρ controls the long-trajectory threshold.

For each candidate timing parameter θ, we measure how strongly the intervention event is associated with risky trajectories through function $l(\theta)$:

$$\begin{aligned} l(\theta) =& P(Y(h_T) = 1 \wedge I_\theta(h_T) = 1) \\ & - P(Y(h_T) = 1) \cdot P(I_\theta(h_T) = 1), \end{aligned}$$

which is the *Covariation* between the trigger event and the risk event, measuring whether an intervention timing is statistically associated with risky trajectories beyond its marginal firing frequency. See further details in Appendix B.1. In practice, we can estimate this quantity empirically on $\mathcal{D}_{\text{sub}}$ and choose the timing parameter by

$$\begin{aligned} \theta^* =& \arg\max_\theta \sum_{i=1}^m I_\theta([h_T]_i) \cdot Y([h_T]_i) \\ & - \frac{1}{m} \sum_{i=1}^m I_\theta([h_T]_i) \sum_{i=1}^m Y([h_T]_i), \end{aligned} \tag{3}$$

which is proved in Appendix B.2. Since the steps are discrete, θ is also discrete; hence, determining θ^* only involves simple frequency counting. The resulting θ^* determines when each intervention should be activated in the guideline-augmented scaffold.

Guideline Generation. After estimating intervention timing, we generate the guideline content using the same LLM model. Each selected timing parameter θ specifies when a guideline should fire, while the language model writes the corresponding natural-language instruction. Formally,

Algorithm 1 Test-Time AdaScaffold

Require: LLM Θ_{LLM}, fixed scaffold S_{fix}, subset $\mathcal{D}_{\text{sub}} = \{x_i\}_{i=1}^{m}$
Ensure: Final results under S_{ada}
1: # Step 1: Guideline Generation for AdaScaffold
2: Collect trajectories $\{[h_T]_i\}_{i=1}^{m}$ with S_{fix}
3: # Intervention Timing Estimation
4: Estimate timing parameters Θ^* from $\{[h_T]_i\}_{i=1}^{m}$ using equation 3.
5: # Guideline Generation
6: Prompt Θ_{LLM} as a supervisor with Θ^* to generate $f_{\text{guideline}}(\mathcal{D}_{\text{sub}}, \Theta_{\text{LLM}})$ using equation 4.
7: # Step 2: Applying AdaScaffold to SWE Tasks
8: $S_{\text{ada}} \leftarrow S_{\text{fix}} \oplus f_{\text{guideline}}(\mathcal{D}_{\text{sub}}, \Theta_{\text{LLM}})$.
9: Compute Score$(\Theta_{\text{LLM}}, S_{\text{ada}})$ using equation 1.
10: **return** Score$(\Theta_{\text{LLM}}, S_{\text{ada}})$

given the optimized parameter set Θ^*, the guideline generator is defined as

$$f_{\text{guideline}}(\mathcal{D}_{\text{sub}}, \Theta_{\text{LLM}}) = \bigcup_{\theta \in \Theta^*} \Theta_{\text{LLM}}(\texttt{Prompt}(\theta)). \tag{4}$$

The prompt contains the estimated intervention timing, trajectory statistics from $\mathcal{D}_{\text{sub}}$, and the required guideline format. Please refer to Appendix A.2 for the full prompt templates. The generated guidelines are attached to S_{fix} to form S_{ada}, as shown in equation 2. During evaluation, these guidelines are executed deterministically: they inspect the current trajectory state and inject the generated instruction when their trigger condition is satisfied.

When a supervisor guideline is triggered, its instruction is inserted immediately after the observation returned by the previous action, thereby conditioning the agent's next decision step. Figure 3 shows three representative classes of intervention guidelines: repetitive-action guidelines prompt the agent to break loops and try alternative operations, delayed-editing guidelines push the agent from exploration to implementation when many steps have passed without code changes, and completion guidelines guide the agent toward lightweight verification and submission after a patch is produced.

By injecting these instructions at observation boundaries, AdaScaffold regularizes the agent's workflow and reduces unnecessary token expenditure on redundant exploration or stalled action patterns.

Overall Framework. Figure 4 illustrates the overall two-stage workflow of our AdaScaffold construction. In the first step, we ask the LLM to act as a software engineering agent and sample a subset $\mathcal{D}_{\text{sub}}$ at test time. For each instance, the base original scaffold S_{fix} is used to generate an initial trajectory $[h_T]_i$. Since hidden test cases are not accessible during this process, these trajectories do not contain correctness labels.

Given the collected trajectories $\{[h_T]_i\}$, we estimate the parameter set Θ^* according to equation 3. We then prompt the LLM in a supervisor role, conditioning on the estimated parameters, to generate intervention guidelines as in equation 4. These guidelines are task- and model-specific, and are merged into the original scaffold following equation 2, yielding S_{ada}.

In the second step, the LLM returns to the software engineering role and solves each instance using the constructed AdaScaffold. The final predictions are then produced by applying S_{ada} in the inference procedure defined in equation 1. The detailed procedure is summarized in Algorithm 1.

5 Experiments

Benchmark and Metrics. We evaluate on SWE-Bench-Verified (Jimenez et al., 2024). Each task is launched in an isolated Docker container via the R2E-Gym (Jain et al., 2025) harness; the agent edits files, runs shell commands, and submits a final patch under the OpenHands (Wang et al., 2025) scaffold, and is graded by the hidden test suite. We report four complementary metrics: (1) *Resolve Rate*: the fraction of tasks whose final patch passes all hidden tests; the primary quality signal (higher is better). (2) *Normal-Exit Rate*: the fraction of runs that terminate with a voluntary submit rather than by hitting a step, time, or token budget, capturing whether the agent stays in control of its own termination (higher is better). (3) *Average Steps*: the average number of tool calls per trajectory, used

Table 2: Main results of the baseline (i.e., the original scaffold), comparative methods guided by instances or subtasks, and our method (i.e., AdaScaffold) on SWE-Bench-Verified. AdaScaffold results are highlighted ; values show mean ± std over 3 seeds. Positive gains relative to the baseline are highlighted in green, while negative ones are marked in red.

Model	Method	Resolve Rate (%)↑	Δ↑	Normal Exit (%)↑	Δ↑	Avg Steps↓	Budget↓	Avg Tokens (M)↓	Budget↓
4B	Original	34.73 ± 1.03	-	7.00 ± 1.11	-	76.46 ± 0.28	-	2.17 ± 0.03	-
	Instance	36.07 ± 0.75	+1.33	5.60 ± 0.28	-1.40	76.92 ± 0.07	100.60%	2.01 ± 0.01	92.63%
	Subtask	33.73 ± 0.09	-1.00	7.13 ± 0.19	+0.13	77.58 ± 0.51	101.46%	2.18 ± 0.01	100.46%
	AdaScaffold	39.33 ± 1.04	+4.60	33.93 ± 0.75	+26.93	66.81 ± 0.32	87.38%	1.69 ± 0.03	77.88%
9B	Original	48.13 ± 0.95	-	7.13 ± 0.42	-	84.71 ± 0.34	-	2.76 ± 0.03	-
	Instance	48.20 ± 1.41	+0.07	6.67 ± 0.09	-0.46	83.78 ± 0.36	98.90%	2.76 ± 0.01	100.00%
	Subtask	44.33 ± 1.79	-3.80	9.07 ± 0.09	+1.94	82.43 ± 0.29	97.31%	2.64 ± 0.04	95.65%
	AdaScaffold	50.67 ± 0.34	+2.54	52.67 ± 0.98	+45.54	68.88 ± 0.28	81.31%	1.97 ± 0.01	71.38%
27B	Original	60.00 ± 0.72	-	17.80 ± 0.80	-	89.16 ± 0.46	-	2.77 ± 0.01	-
	Instance	62.60 ± 1.13	+2.60	7.87 ± 0.19	-9.93	76.98 ± 0.17	86.34%	2.17 ± 0.01	78.34%
	Subtask	63.27 ± 1.04	+3.27	13.93 ± 0.75	-3.87	74.65 ± 0.33	83.73%	2.13 ± 0.02	76.90%
	AdaScaffold	65.40 ± 0.75	+5.40	91.53 ± 0.34	+73.73	54.36 ± 0.04	60.97%	1.29 ± 0.00	46.57%

as a step-budget cost proxy (lower is better). (4) *Average Tokens*: the average number of completion tokens per trajectory, used as a compute cost proxy (lower is better). Given the LLM, a good scaffold will improve task performance while consuming fewer computational resources.

Implementation Details. We evaluate three Qwen-3.5 (Yang et al., 2025) models at 4B, 9B, and 27B parameters, as well as GLM-5.2 (Zeng et al., 2026) at 753B. Main experiments with Qwen-3.5 models are served by vLLM (Kwon et al., 2023) and rolled out through the R2E-Gym (Jain et al., 2025) Kubernetes backend with 16 parallel workers on a 4×A100-80GB node. Every configuration is repeated with 3 random seeds over the full test split, and we report mean and standard deviation. Baseline and AdaScaffold share identical agent hyper-parameters: at most 80 agent steps with a hard cap of 100, a context window of 65,536 tokens per completion, a per-task wall-clock budget of 7,200s and a per-task LLM budget of 1,200s, a 90s cap per tool execution, sampling temperature 0.6 and top-p 0.95. Function calling is omitted to guarantee generalization performance. For the *guideline generation* part, in addition to using each student as its own supervisor, we also use Claude Opus 4.6 (Anthropic, 2025) and GPT-5.5 (Singh et al., 2025) as external supervisors.

Comparative Methods. We compare AdaScaffold with two test-time methods: (1) Adapting with *Instance*-level experience (Ouyang et al., 2026), which distills each trajectory into a whole-task experience containing the problem, solution path, and lessons learned. At test time, similar experiences are retrieved using the new problem statement and prepended to the system prompt. (2) Adapting with *Subtask*-level experience (Shen et al., 2026), which decomposes each trajectory into localized subtasks and stores a separate experience for each. At test time, recent context retrieves relevant subtask experiences step-by-step as procedural hints. Both methods use the same test-time setting as AdaScaffold, described in Section 4.

Main Results. Table 2 compares the fixed scaffold with AdaScaffold on three Qwen-3.5 models (4B, 9B, 27B), with each model serving as its own supervisor. AdaScaffold consistently improves *Resolve Rate* (+4.60 / +2.54 / +5.40 points) and substantially increases *Normal-Exit Rate* (+26.93 / +45.54 / +73.73 points). Meanwhile, it uses only 87.4% / 81.3% / 61.0% of baseline steps and 77.8% / 71.4% / 46.6% of completion tokens. Notably, the 27B model achieves the largest quality gain at less than half the token cost. Results on GLM-5.2 are provided in Appendix C.2.

Compared with instance- and subtask-level experience adaptations, AdaScaffold is more effective, particularly for smaller models, where experience-based methods provide virtually no *Resolve Rate* improvement. Their limited efficiency gains may also be offset by the overhead of processing injected experiences. In contrast, AdaScaffold provides explicit supervision throughout reasoning without extensive experience injection or retrieval, improving both performance and efficiency.

Figure 5 shows that AdaScaffold's gains for both 4B and 27B mainly come from reliably completing easy and medium-difficulty tasks, suggesting that it acts primarily as a scheduler that converts existing model capability into completed solutions. Its mechanism also substantially increases normal termination, indicating that interventions are well aligned with agent progress.

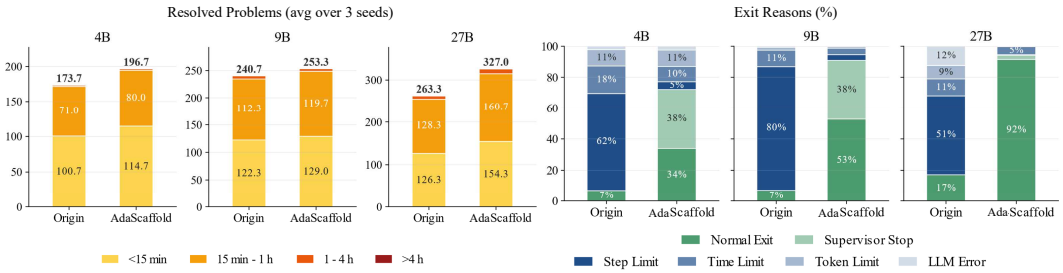

Figure 5: Distribution of difficulty levels of resolved tasks (left) and trajectory exit reasons (right) on SWE-bench-Verified, comparing 'Original' and AdaScaffold (averaged over 3 seeds).

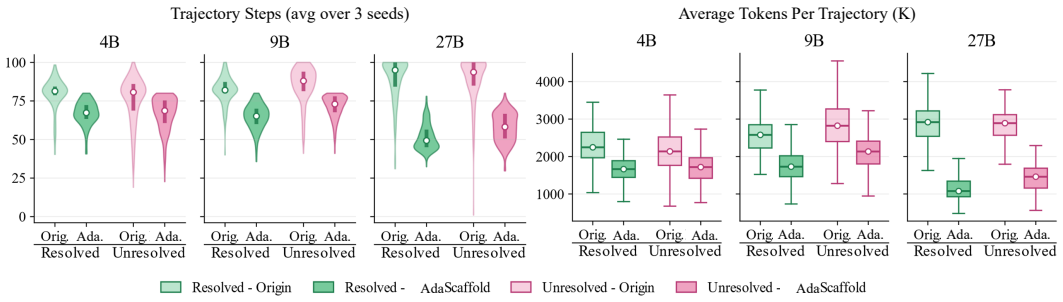

Figure 6: Distribution of trajectory step counts (left) and average tokens consumed per trajectory (right) for resolved and unresolved tasks, comparing 'Original' and AdaScaffold.

Figure 6 further shows that AdaScaffold shifts step and token distributions downward across model scales. For 27B, the median steps on resolved tasks decrease from ~ 95 to ~ 50, while the median token cost drops by roughly $2\times$. Thus, AdaScaffold achieves substantial compute savings through both fewer interaction rounds and shorter, more focused interactions.

Ablation Studies. Results of the ablation studies are shown in Table 3. *W/o Time Estimation* removes our *Covariation*-based parameter estimation and replaces it with random parameters for intervention, while *W/o Guideline Generation* keeps guideline triggering unchanged but removes LLM-based guideline generation. When triggered, the system only points out the mistake without providing actionable guidance. These two settings therefore examine the importance of (1) when to intervene and (2) what content to provide.

The *w/o Time Estimation* results show that random guideline injection weakens effectiveness. Although it reduces resource usage more aggressively than AdaScaffold (e.g., 4B: Steps / Tokens 66.81 / 1.69M $\rightarrow$ 60.93 / 1.47M; 27B: 54.36 / 1.29M $\rightarrow$ 49.20 / 1.03M), the performance gain over Baseline nearly disappears: *Resolve Rate* reaches only 35.73% (4B, +1.00) and 60.53% (27B, +0.53), compared with AdaScaffold's +4.60 and +5.40 gains. *Normal-Exit Rate* is also substantially lower (4B: 17.80% vs. 33.93%; 27B: 55.20% vs. 91.53%). This confirms that appropriate intervention timing is crucial for translating guidelines into task success.

The *w/o Guideline Generation* setting causes an even more severe degradation. *Resolve Rate* falls below Baseline across all model sizes (4B / 9B / 27B: 32.47% / 32.13% / 54.93% vs. 34.73% / 48.13% / 60.00%), while *Normal-Exit Rate* also drops below Baseline (5.00% / 6.67% / 6.87% vs. 7.00% / 7.13% / 17.80%). Efficiency likewise deteriorates, with 27B increasing from 54.36 steps and 1.29M tokens in AdaScaffold to 73.65 steps and 1.99M tokens. These results show that merely pointing out errors without actionable corrective guidance can lead to repeated failures, making *Guideline Generation* essential for both effectiveness and efficiency.

Case Studies. Figure 7 presents two SWE-bench-Verified case studies using the 4B agent, comparing 'Original' and 'AdaScaffold' on the same instances. Each row shows step-indexed trajectories, with actions grouped into *Explore*, *View*, *Edit*, *Run / Test*, and *Submit*. Colored bubbles mark AdaS-

Table 3: Ablation studies of different modules, and ours are highlighted; showing mean ± std over 3 seeds. Positive gains relative to the baseline (i.e., original scaffold) are in green, while negative ones are in red. 'TE' means 'Time Estimation', while 'GG' means 'Guideline Generation'.

	Method	Resolve Rate (%)↑	Δ↑	Normal Exit (%)↑	Δ↑	Avg Steps↓	Budget↓	Avg Tokens (M)↓	Budget↓
4B	AdaScaffold	39.33 ± 1.04	+4.60	33.93 ± 0.75	+26.93	66.81 ± 0.32	87.38%	1.69 ± 0.03	77.88%
	w/o TE	35.73 ± 1.55	+1.00	17.80 ± 1.30	+10.80	60.93 ± 0.65	79.77%	1.47 ± 0.01	67.74%
	w/o GG	32.47 ± 2.38	-2.26	5.00 ± 0.75	-2.00	70.92 ± 0.34	92.76%	1.90 ± 0.02	87.56%
9B	AdaScaffold	50.67 ± 0.34	+2.54	52.67 ± 0.98	+45.54	68.88 ± 0.28	81.31%	1.97 ± 0.01	71.38%
	w/o TE	43.07 ± 0.84	-5.06	16.07 ± 1.18	+8.94	56.54 ± 0.23	66.74%	1.40 ± 0.01	50.72%
	w/o GG	32.13 ± 0.74	-16.00	6.67 ± 0.19	-0.46	75.50 ± 0.37	89.09%	2.01 ± 0.06	72.83%
27B	AdaScaffold	65.40 ± 0.75	+5.40	91.53 ± 0.34	+73.73	54.36 ± 0.04	60.97%	1.29 ± 0.00	46.57%
	w/o TE	60.53 ± 0.36	+0.53	55.20 ± 1.56	+37.40	49.20 ± 0.56	55.19%	1.03 ± 0.01	37.18%
	w/o GG	54.93 ± 0.68	-5.07	6.87 ± 0.62	-10.93	73.65 ± 0.13	82.61%	1.99 ± 0.01	71.84%

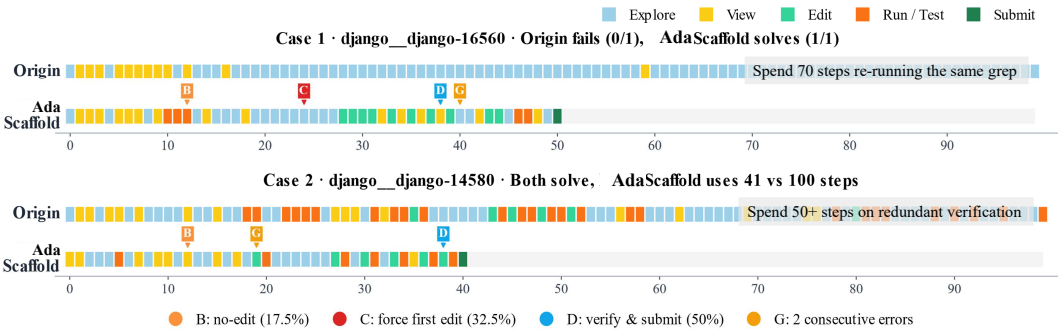

Figure 7: Two SWE-bench-Verified case studies comparing 'Original' and AdaScaffold (4B model); each strip shows per-step actions, with guideline firings marked above AdaScaffold. Top (`django-16560`): AdaScaffold breaks Origin's exploration loop and enables a correct patch. Bottom (`django-14580`): both succeed, but AdaScaffold uses fewer steps.

caffold's guideline interventions: B (no edit by 17.5% budget), C (force first edit by 32.5%), D (verify-and-submit at 50%), and G (fix root cause after two consecutive errors).

The top row shows an instance that 'Original' fails to solve (reward 0/1): after locating the relevant file, it enters a prolonged exploration loop without editing and reaches the 100-step limit. AdaScaffold instead triggers B at step 12 and C at step 24 to initiate editing, followed by G and D for verification and submission, successfully solving the task at step 50. This demonstrates how timely interventions can rescue otherwise unsolved instances.

The bottom row shows an instance solved by both agents but with different costs. 'Original' finds a correct patch at step 48 but spends 50+ additional steps on redundant verification before hitting the limit. AdaScaffold triggers D after verification and submits at step 40, avoiding this unnecessary tail. Thus, AdaScaffold can preserve correctness while substantially reducing trajectory length and token consumption.

Other Experiments. We further evaluate AdaScaffold under diverse settings in Appendix C.1–C.4. We examine different LLMs to generate guidelines, larger and more capable models (GLM-5.2), cross-dataset estimation of intervention parameters, and the sensitivity to the number of trajectories used for parameter estimation. These results further demonstrate the robustness, transferability, and stability of AdaScaffold across supervisors, model scales, datasets, and calibration sizes.

6 Conclusion

AdaScaffold addresses the bottleneck of one-fits-all scaffolds by adapting agent workflows from unlabeled test-time trajectories. Its Timing Estimation and Guideline Generation modules provide model- and task-specific supervision, consistently improving resolve rates while reducing step and token budgets. These results demonstrate that scaffold-level adaptation is a promising direction for building more reliable and efficient coding agents.

AI USE STATEMENT

In this work, we used generative AI tools only to improve the readability and language of the manuscript. We **have not** used generative AI tools for any of the required disclosure tasks, including generating synthetic data, developing theoretical models or conceptual frameworks, formulating mathematical claims, designing or providing feedback on research methodology or experiments, implementing methods, interpreting results, or other research-related tasks. The use of generative AI for other recommended disclosure tasks is not applicable to this work.

We have reviewed all AI-assisted text and verified its accuracy and consistency with the original work. We take full responsibility for the final content of this work, including all text, claims, and results.

ETHICS STATEMENT

We have carefully considered the ethical implications of this work and have no specific ethical concerns to declare. Our study **does not** involve human subjects, sensitive personal data, or the collection of new data from individuals. We also do not identify any potential risks related to discrimination, privacy, security, or harmful applications beyond those inherent to general-purpose AI systems. The experiments are conducted using publicly available benchmarks and standard research practices.

REPRODUCIBILITY STATEMENT

To facilitate reproducibility, we provide the source code and implementation at `https://anonymous.4open.science/r/AdaScaffold`. Detailed implementation settings and experimental configurations are provided in Section 5 and Appendix A.1. All datasets used in our experiments are publicly available and can be downloaded from their original sources. These resources together provide the necessary information to reproduce our experimental results.

REFERENCES

Anthropic. Introducing claude 4. Anthropic News Blog, May 2025. URL `https://www.anthropic.com/news/claude-4`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *NeurIPS*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, 2023.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, YK Li, et al. Deepseek-coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 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. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. *ICLR*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, 2023.

Raymond Li, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, et al. Starcoder: may the source be with you! *TMLR*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 2022.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*, 2024.

Yuejiang Liu, Parth Kothari, Bastien Van Delft, Baptiste Bellot-Gurlet, Taylor Mordan, and Alexandre Alahi. Ttt++: When does self-supervised test-time training fail or thrive? *NeurIPS*, 2021.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *NeurIPS*, 2023.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*, 2023.

Shuaicheng Niu, Jiaxiang Wu, Yifan Zhang, Yaofo Chen, Shijian Zheng, Peilin Zhao, and Mingkui Tan. Efficient test-time model adaptation without forgetting. In *ICML*, 2022.

Shuaicheng Niu, Jiaxiang Wu, Yifan Zhang, Zhiquan Wen, Yaofo Chen, Peilin Zhao, and Mingkui Tan. Towards stable test-time adaptation in dynamic wild world. In *ICLR*, 2023.

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *ICLR*, 2026.

Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of maml. In *ICLR*, 2020.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *ICML*, 2016.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *NeurIPS*, 2023.

Kangning Shen, Jingyuan Zhang, Chenxi Sun, Wencong Zeng, and Yang Yue. Structurally aligned subtask-level memory for software engineering agents. *arXiv preprint arXiv:2602.21611*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS*, 2023.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *ICML*, 2020.

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. *NeurIPS*, 2016.

Dequan Wang, Evan Shelhamer, Shaoteng Liu, Bruno Olshausen, and Trevor Darrell. Tent: Fully test-time adaptation by entropy minimization. In *ICLR*, 2021a.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *ICML*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *ICLR*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2023.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, 2021b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*, 2022.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Josh Clymer, Jai Dhyani, et al. Re-bench: Evaluating frontier ai r&d capabilities of language model agents against human experts. *arXiv preprint arXiv:2411.15114*, 2024.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

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. *NeurIPS*, 2024.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. In *ICML*, 2024.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *NeurIPS*, 2022.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Marvin Zhang, Sergey Levine, and Chelsea Finn. Memo: Test time robustness via adaptation and augmentation. *NeurIPS*, 2022.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *ISSTA*, 2024.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *ICLR*, 2023.

A More Details About AdaScaffold

A.1 Mis-Behavviors and Parameter Estimation Details

Regarding mis-behavior specification, the predefined mis-behaviors in our framework can be specified manually or generated by an LLM.

In this work, we choose to manually identify a small set of representative mis-behavior categories commonly observed in software engineering agents and associate each category with one or more interventions, as summarized in Table 4. This simple specification already achieves strong performance and generalization across different models and tasks, indicating that effective scaffold adaptation does not require exhaustive coverage of agent mis-behaviors.

The selected categories cover three recurring aspects of agent behavior: *progress*, *verification and termination*, and *tool use*. Specifically, *stagnation* captures cases where the agent repeatedly takes the same action without making progress (A), while *delayed editing* captures excessive hesitation before modifying the code (B, C). *Ineffective verification and termination* covers failing to establish useful checkpoints, failing to submit an available solution, or continuing unnecessary actions near the budget limit (D, E, F). Finally, *ineffective tool use* captures repeated erroneous observations or redundant test execution without subsequent editing (G, H). These categories correspond to distinct and observable failure patterns in coding-agent trajectories and therefore provide natural targets for scaffold-level intervention.

This simple specification already enables effective scaffold adaptation, suggesting that AdaScaffold does not require an exhaustive or automatically discovered set of mis-behaviors to be effective. The framework can be readily extended with additional mis-behaviors generated by an LLM or identified from other sources.

Table 4: Mis-behavior categories and their corresponding interventions.

Mis-behavior	**Description**	**Interventions**
Stagnation	Repeatedly taking the same action without making progress.	A
Delayed editing	Spending excessive steps reasoning or inspecting without making an initial code edit.	B, C
Ineffective verification	Failing to establish useful checkpoints or verify whether a viable patch has been produced.	D
Ineffective termination	Failing to submit an available solution or continuing unnecessary actions after a patch is ready, especially near the budget limit.	E, F
Ineffective tool use	Repeatedly producing erroneous observations or running tests without making corresponding edits.	G, H

A.1.1 Trajectory features.

For every trajectory $h_T \in \mathcal{D}_{\text{sub}}$ we precompute step-indexed features that are reused across all candidate θ:

- the action-type sequence $\{a_t^{\text{type}}\}_{t=1}^{T}$, mapping each tool call to one of

  $$\{\text{VIEW}, \text{EDIT}, \text{CREATE}, \text{INSERT}, \text{RUN_TEST}, \text{EXPLORE_FS}, \text{SEARCH}, \text{SUBMIT}, \ldots\};$$

- a canonicalised action signature $\sigma(a_t)$, obtained by collapsing whitespace, masking quoted strings, replacing integers with a placeholder, and truncating to 200 characters;
- the indicators $\textsc{Edited}(t) = \mathbb{1}\{\exists s \le t : a_s^{\text{type}} \in \{\text{EDIT}, \text{CREATE}, \text{INSERT}\}\}$ and $\textsc{HasPatch}(t)$, the latter additionally gated by a non-empty diff returned by the environment;

- the run-length counters TESTNOEDIT(t) (consecutive RUN_TEST steps since the last edit) and ERRSTREAK(t) (consecutive steps whose observation matches a canonical error pattern: `Traceback`, `No replacement was performed`, `FileNotFoundError`, `SyntaxError`, ...).

A.1.2 RISK PROXY AND CALIBRATION CONSTANTS

The risk label $Y(h_T)$ uses $\hat{T} = 80$, $\rho = 0.85$, and treats $e(h_T)$ as undesirable when $e(h_T) \in \{$MAX_STEP_LIMIT, AGENT_MAX_STEP_LIMIT, ABS_STEP_LIMIT, TRAJ_TIME_LIMIT, TOKEN_LIMIT$\}$. Calibration uses $m = 100$ trajectories drawn from the available pool by reservoir sampling, and a minimum trigger support of $N_\theta \geq 3$ to suppress noise. Conditional risks are Laplace-smoothed with $\alpha = \beta = 1$.

A.1.3 INDICATOR I_θ AND SEARCH GRID PER SLOT

Each slot is associated with a closed-form indicator $I_\theta(h_T)$ that is 1 on the first step where its trigger condition holds. The search grid for θ is fixed in advance:

- **Slot A.** $\theta = (w, m_{\min}, t_{\min})$. Fires at the smallest t such that $\sigma(a_{t-w+1}) = \cdots = \sigma(a_t)$ with $w = m_{\min}$ and $t \geq \max(t_{\min}, m_{\min})$. Grid: $m_{\min} \in \{2, 3, 4, 5\}$, $t_{\min} \in \{4, 6, 8, 10, 12\}$.
- **Slots B / C.** $\theta = \phi$. Fires at $t = \lfloor \phi \hat{T} \rfloor$ iff EDITED$(t) = 0$. Grid: even $t \in [0.10\hat{T},\ 0.66\hat{T}]$.
- **Slots D / E.** $\theta = \phi$. Fires at $t = \lfloor \phi \hat{T} \rfloor$, with patch-conditioned variants $\{$HASPATCH$(t) = 0\}$ used for D and $\{$HASPATCH$(t) = 1\}$ for E. Grid: even $t \in [0.50\hat{T},\ 0.96\hat{T}]$.
- **Slot F.** $\theta = \phi$. Fires at $t = \lfloor \phi \hat{T} \rfloor$ iff HASPATCH$(t) = 1$. Grid: even $t \in [0.85\hat{T},\ \hat{T}]$.
- **Slot G.** $\theta = k$. Fires at the first t with ERRSTREAK$(t) \geq k$. Grid: $k \in \{2, 3, 4, 5\}$.
- **Slot H.** $\theta = k$. Fires at the first t with TESTNOEDIT$(t) \geq k$ and $a_t^{\text{type}} =$ RUN_TEST. Grid: $k \in \{2, 3, 4, 5\}$.

The empirical covariation $\hat{l}(\theta)$ is computed for every grid point as in equation 3 and equivalently rewritten as

$$\hat{l}(\theta) = N_\theta \left[\hat{P}(Y = 1 \mid I_\theta = 1) - \hat{P}(Y = 1)\right].$$

A.1.4 SLOT-LEVEL CONSTRAINTS ON $\theta^\star$

maximizing $\hat{l}(\theta)$ alone can produce non-monotone schedules across slots, so we impose deterministic phase constraints when picking $\theta^\star$:

- **B vs. C.** On no-edit candidates with $\hat{l}$ above its median, B selects the earliest ϕ; C is then chosen on $\{\phi : \phi \geq \phi^\star_{\text{B}} + 5/\hat{T}\}$ to enforce a strict ordering between the soft and hard nudges.
- **D / E.** D uses the patch-free candidates with the 60-th-percentile $\hat{l}$ cut-off and the earliest fraction; E uses the patch-conditional candidates with the 75-th-percentile cut-off and $\phi^\star_{\text{E}} \geq \phi^\star_{\text{D}} + 0.10$.
- **F.** F uses the patch-conditional late candidates with the 50-th-percentile cut-off and a preference for the latest fraction, subject to $\max(0.95,\ \phi^\star_{\text{E}} + 0.05) \leq \phi^\star_{\text{F}} \leq 0.98$ to leave a margin under the absolute step cap.

If the support filter eliminates every candidate of a slot, we fall back to fixed defaults so the guideline set always covers all slots: $w = 3$, $m_{\min} = 3$, $t_{\min} = 6$ (A); $\phi = 0.25, 0.50, 0.70, 0.90, 0.98$ (B–F); $k = 3$ (G/H).

A.1.5 COST

For each $(\Theta_{\mathrm{LLM}}, \mathcal{D}_{\mathrm{sub}})$ pair, the calibration above runs in seconds on a single CPU; guideline firing at inference is pure state matching and adds no additional LLM calls.

A.2 GUIDELINE-GENERATION PROMPT TEMPLATE

Once the timing parameters $\theta^\star$ have been estimated as in Appendix A.1, they are inserted into a single, fixed prompt template that asks the supervisor LLM $\mathcal{T} = \Theta_{\mathrm{LLM}}$ to emit the final guideline set $f_{\mathrm{guideline}}(\mathcal{D}_{\mathrm{sub}}, \Theta_{\mathrm{LLM}})$ in JSON form. The template is identical across all $(\Theta_{\mathrm{LLM}}, \mathcal{D}_{\mathrm{sub}})$ pairs; only the calibration statistics, the numeric parameters, and the target model name are interpolated. Table 5 summarizes its structure. The supervisor is instructed to copy the estimated $\theta^\star$ values verbatim into the trigger fields, to fill in the eight slots in a fixed order, and to return one valid JSON object without surrounding prose. The returned JSON is then validated against the trigger schema, slot ordering, and per-slot contract; guidelines that violate the schema are dropped, and a force-done safety net is added if missing. The validated JSON is the guideline set used by the guideline engine at inference time.

B MORE THEORETICAL SUPPORT

B.1 COVARIATION AND INDEPENDENCE

Let $I_\theta(h_T), Y(h_T) \in \{0, 1\}$ be the trigger indicator and the risk indicator defined in the main text, viewed as Bernoulli random variables under the data-generating distribution of trajectories produced by Θ_{LLM} on S_{fix}.

Covariation equals covariance for $0/1$ variables. Since $\mathbb{E}[I_\theta] = P(I_\theta{=}1)$, $\mathbb{E}[Y] = P(Y{=}1)$, and $\mathbb{E}[I_\theta\, Y] = P(I_\theta{=}1 \wedge Y{=}1)$, the quantity in the main text satisfies

$$\begin{aligned} l(\theta) &= P(Y{=}1 \wedge I_\theta{=}1) - P(Y{=}1)\, P(I_\theta{=}1) \\ &= \mathbb{E}[I_\theta\, Y] - \mathbb{E}[I_\theta]\, \mathbb{E}[Y] \\ &= \mathrm{Cov}(I_\theta, Y). \end{aligned} \tag{5}$$

Hence $l(\theta)$ is exactly the covariance between the trigger event and the risk event.

Interpretation of covariation. Factoring equation 5 gives

$$l(\theta) = p_\theta \big[\, P(Y{=}1 \mid I_\theta{=}1) - q \,\big],$$

where $p_\theta = P(I_\theta{=}1)$ and $q = P(Y{=}1)$. Thus, maximizing $l(\theta)$ favors intervention timings that are associated with a positive risk *lift* while also occurring sufficiently often in the calibration trajectories. This criterion deliberately balances conditional risk enrichment against trigger support: a trigger that is highly predictive but extremely rare receives limited weight, as does a frequent trigger with little risk enrichment. We therefore use covariation as a practical, label-free criterion for selecting intervention timings; it is not intended to establish a causal effect of an intervention.

B.2 PROOF OF EQUATION 3

Let $\mathcal{D}_{\mathrm{sub}} = \{[h_T]_i\}_{i=1}^m$ be an i.i.d. sample of trajectories. The plug-in (empirical) estimators for the three quantities appearing in $l(\theta)$ are

$$\hat{P}(Y{=}1 \wedge I_\theta{=}1) = \frac{1}{m} \sum_{i=1}^{m} I_\theta([h_T]_i)\, Y([h_T]_i),$$

$$\hat{P}(I_\theta{=}1) = \frac{1}{m} \sum_{i=1}^{m} I_\theta([h_T]_i),$$

$$\hat{P}(Y{=}1) = \frac{1}{m} \sum_{i=1}^{m} Y([h_T]_i),$$

Table 5: Block-level structure of the guideline-generation prompt sent to the supervisor LLM. The actual template is reused across all supervisor–student pairs; only the calibration statistics and the estimated parameters $\theta^\star$ are interpolated per run.

Block	Content (abridged)
Role	"You are a senior engineer who has read hundreds of failed trajectories from small coding agents." Frames the LLM as a supervisor that writes the literal message text the student will receive, plus the trigger that decides when to send it.
Student / scaffold spec	Student model name, parameter count, architecture, and step budget $\hat{T}$. The fixed scaffold S_{fix} exposes three XML tools: `str_replace_editor`, `execute_bash`, `submit`; a canonical XML call is shown verbatim.
Phase strategy	Early / middle / late / final phases, mapped to the desired supervisor behaviour: allow exploration unless looping, push for a first concrete edit, redirect late exploration into verification or submission, and force termination when the budget is nearly exhausted.
Calibration data	Behavior-only statistics computed on $\mathcal{D}_{\text{sub}}$ (no reward used): risk proxy Y, global risk rate, exit-reason distribution, trajectory-length distribution, first-edit-step distribution, and the top high-risk state buckets.
Numeric parameters	The estimated $\theta^\star$ values are pasted as a JSON block with the explicit instruction "use VERBATIM". A field-level mapping ties each slot to its parameter (e.g. Slot C `step_fraction` = `no_edit_hard_fraction`).
Trigger schema	The closed set of supported trigger types and required fields: `action_repeat`, `no_edit_by_fraction`, `step_fraction_exact`, `force_done_at_fraction`, `error_streak`, `test_without_edit`, `obs_bloat_streak`. The list of allowed message placeholders is also given.
Slot contract	Required ordering A–F (G/H optional, last). Per-slot constraints, e.g. Slot D must define both `message_if_patch` and `message_no_patch` with different next actions; Slot E both branches must contain a literal `<function=submit></function>`; Slot F must use top-level `"action": "force_done"` with no message.
Example	A fully written Slot C rule (`no_edit_by_fraction`) is shown to fix the bar for textual messages: short diagnosis, named tool and parameter, concrete next action.
Style rules	Every textual message must start with `[Supervisor]`, name at least one tool, give a concrete next action, be at least 25 words, and must not recommend `git reset / checkout / stash`, file deletion, or running the full test suite.
Output format	A skeleton JSON with a `meta` block (version, source, `uses_reward: false`, sample size, seed) and an ordered `rules` array. Hard validity rules: exact slot order, slot-C trigger type, slot-D dual messages, slot-E literal submit snippet, slot-F `force_done`, every numeric trigger value drawn from $\theta^\star$, and a single JSON object as output (no fences, no prose).

each of which is unbiased and consistent for the corresponding population probability. Substituting into equation 5 gives the empirical covariation

$$\hat{l}(\theta) = \frac{1}{m}\sum_{i=1}^{m} I_\theta([h_T]_i)\, Y([h_T]_i) - \frac{1}{m^2}\left(\sum_{i=1}^{m} I_\theta([h_T]_i)\right)\left(\sum_{i=1}^{m} Y([h_T]_i)\right). \tag{6}$$

Since $m > 0$ does not depend on θ, multiplying equation 6 by m preserves the argmax:

$$\begin{aligned}
\arg\max_\theta \hat{l}(\theta) &= \arg\max_\theta m\,\hat{l}(\theta) \\
&= \arg\max_\theta \left[\sum_{i=1}^{m} I_\theta([h_T]_i)\, Y([h_T]_i) - \frac{1}{m}\sum_{i=1}^{m} I_\theta([h_T]_i)\sum_{i=1}^{m} Y([h_T]_i)\right] \\
&= \theta^\star,
\end{aligned}$$

which is exactly equation 3. Hence empirically maximizing the covariation between I_θ and Y on $\mathcal{D}_{\text{sub}}$ is equivalent to the timing-selection guideline used in the main text. By the law of large numbers, $\hat{l}(\theta) \xrightarrow{m\to\infty} l(\theta)$ for every fixed θ, and since the candidate set defined in Appendix A.1 is

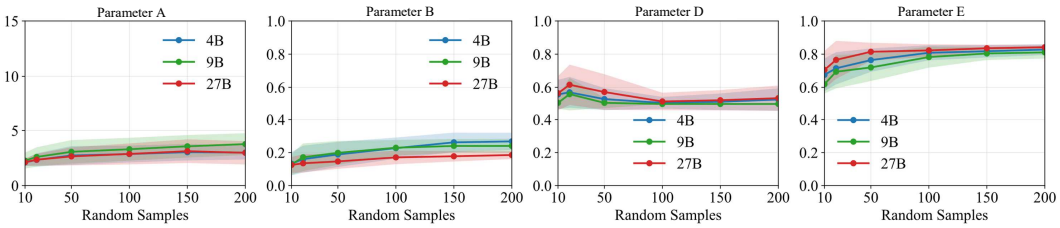

Figure 8: Value of estimated parameters according to the number of random samples used. Four parameters among Θ are shown above as examples.

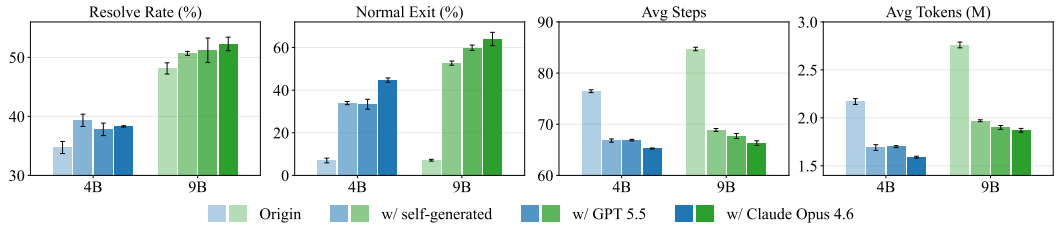

Figure 9: Comparison of different LLMs as supervisors (i.e., using different LLMs to generate guidelines). *Origin* is the base model without AdaScaffold; *w/ self-generated*, *w/ GPT-5.5*, and *w/ Claude Opus 4.6* use the corresponding LLM as the supervisor.

finite, $\theta^\star \xrightarrow{m\to\infty} \arg\max_\theta l(\theta)$ as well. The empirical procedure of equation 3 therefore inherits the association-maximizing interpretation established in Appendix B.1.

C More Quantitative Results

C.1 Results of Using Different LLMs to Generate Guideline

Figure 9 studies the effect of using different LLMs as the supervisor that produces the intervention guidelines. For each base agent (Qwen-3.5 4B and Qwen-3.5 9B) we compare four configurations: the 'Original' run with no supervisor, and three AdaScaffold variants whose guidelines are instantiated from estimated parameters Θ by (i) the agent model itself, (ii) GPT-5.5, and (iii) Claude Opus 4.6. The four shades of each colour, from lightest to darkest, denote these four configurations; error bars show the standard deviation, and the four panels report resolve rate, normal-exit rate, average steps and tokens.

Overall the gap between using a much stronger supervisor (GPT-5.5 or Claude Opus 4.6) and using the agent's own model to generate the guidelines is small. For instance on the 4B agent the resolve rate moves from 39.3% (self) to 37.8% / 38.3% (GPT / Claude), and the normal-exit rate of 34% changes only marginally; the step and token savings are nearly identical across the three supervisors. This indicates that, under the AdaScaffold design, a more capable LLM is not necessary to play the supervisor role–the agent model itself suffices, which substantially reduces deployment cost and external API overhead. Secondly, a more nuanced observation comes from the resolve-rate panel: replacing the self-supervisor with GPT-5.5 or Claude Opus 4.6 yields a small gain for the 9B agent (50.7% $\rightarrow$ 51.2% / 52.3%) but a small drop for the 4B agent (39.3% $\rightarrow$ 37.8% / 38.3%). A plausible explanation is that when the capability gap between supervisor and agent becomes too large, the guidelines produced by the stronger supervisor no longer match what the weaker agent can actually execute, so its advice fails to translate into additional resolved tasks.

Table 6 reports a controlled comparison of three AdaScaffold variants that differ only in *which* LLM is used as the supervisor that generates the intervention guidelines. For each base agent (Qwen-3.5 4B, 9B and 27B) we evaluate four configurations: the *Original* run, *w/ self-generated* (the base

Table 6: Comparison of different LLMs as supervisors. *Origin* is the base model without AdaScaffold; *w/ self-generated*, *w/ GPT-5.5*, and *w/ Claude Opus 4.6* use the corresponding LLM as the supervisor.

	Method	Resolve Rate (%)↑	Δ↑	Normal Exit (%)↑	Δ↑	Avg Steps↓	Budget↓	Avg Tokens (M)↓	Budget↓
4B	Original	34.73 ± 1.03	-	7.00 ± 1.11	-	76.46 ± 0.28	-	2.17 ± 0.03	-
	w/ self-generated	39.33 ± 1.04	+4.60	33.93 ± 0.75	+26.93	66.81 ± 0.32	87.37%	1.69 ± 0.03	77.88%
	w/ GPT-5.5	37.80 ± 1.06	+3.07	33.40 ± 2.31	+26.40	66.89 ± 0.15	87.49%	1.70 ± 0.01	78.24%
	w/ Claude Opus 4.6	38.30 ± 0.12	+3.57	44.67 ± 1.05	+37.67	65.26 ± 0.10	85.35%	1.59 ± 0.01	73.27%
9B	Original	48.13 ± 0.95	-	7.13 ± 0.42	-	84.71 ± 0.34	-	2.76 ± 0.03	-
	w/ self-generated	50.67 ± 0.34	+2.54	52.67 ± 0.98	+45.54	68.88 ± 0.28	81.31%	1.97 ± 0.01	71.38%
	w/ GPT-5.5	51.20 ± 2.08	+3.07	59.87 ± 1.27	+52.74	67.70 ± 0.49	79.92%	1.90 ± 0.02	68.94%
	w/ Claude Opus 4.6	52.27 ± 1.15	+4.14	64.00 ± 3.12	+56.87	66.30 ± 0.46	78.27%	1.87 ± 0.02	67.58%
27B	Original	60.00 ± 0.72	-	17.80 ± 0.80	-	89.16 ± 0.46	-	2.77 ± 0.01	-
	w/ self-generated	65.40 ± 0.75	+5.40	91.53 ± 0.34	+73.73	54.36 ± 0.04	60.97%	1.29 ± 0.00	46.57%
	w/ GPT-5.5	67.53 ± 1.42	+7.53	79.33 ± 0.61	+61.53	63.25 ± 0.09	70.94%	1.61 ± 0.01	58.13%
	w/ Claude Opus 4.6	65.93 ± 1.50	+5.93	92.13 ± 1.10	+74.33	54.17 ± 0.46	60.76%	1.27 ± 0.02	45.74%

Table 7: Results of baseline (i.e., original scaffold) and our method (i.e., AdaScaffold) on SWE-Bench-Verified, using GLM-5.2, and ours are highlighted. Positive gains relative to the baseline are highlighted in green, while negative ones are marked in red.

Model	Method	Resolve Rate (%)↑	Δ↑	Normal Exit (%)↑	Δ↑	Avg Steps↓	Budget↓	Avg Tokens (M)↓	Budget↓
GLM-5.2 (753B)	Original Scaffold	69.60 ± 0.28	-	31.73 ± 0.94	-	75.24 ± 0.04	-	1.44 ± 0.03	-
	AdaScaffold	69.13 ± 0.13	-0.47	52.83 ± 1.18	+21.10	66.09 ± 0.13	87.84%	1.19 ± 0.07	82.64%

model itself acts as the supervisor), *w/ GPT-5.5*, and *w/ Claude Opus 4.6*. We report resolve rate, normal-exit rate, average steps and average tokens (mean ± std over 3 seeds), together with the absolute deltas Δ over 'Original' for accuracy-style metrics and the *budget*–the average number of steps / tokens expressed as a fraction of the corresponding 'Original'–for efficiency-style metrics.

Overall the gap between using a stronger external supervisor (GPT-5.5 or Claude Opus 4.6) and letting the agent generate its own guidelines is small. Across all three model sizes the three AdaScaffold variants land within $\sim$2 absolute points of each other on resolve rate, all reduce the average step count to roughly 60%–87% of the Original configuration, and all push the normal-exit rate from single digits up to 30%–90%. This indicates that, under the AdaScaffold design, a more capable LLM is not necessary to play the supervisor role: the agent's own model already suffices, which substantially reduces deployment cost and external API overhead.

A more nuanced observation comes from the resolve-rate column on the 4B and 9B agents. Replacing the self-supervisor with GPT-5.5 or Claude Opus 4.6 yields a small *gain* for the 9B agent ($50.67\% \rightarrow 51.20\% / 52.27\%$) but a small *drop* for the 4B agent ($39.33\% \rightarrow 37.80\% / 38.30\%$). A plausible explanation is that when the capability gap between supervisor and agent becomes too large, the guidelines produced by the stronger supervisor no longer match what the weaker agent can actually execute, so its advice fails to translate into additional resolved tasks. Interestingly, on the 4B agent Claude Opus 4.6 still delivers the most aggressive efficiency profile (normal exit 44.67% vs. 33.93% for self, token budget 73.27% vs. 77.88%), suggesting that even when its guidelines cost the agent a small amount of accuracy, they push the agent to terminate cleaner trajectories more often.

For the 27B agent the picture changes again. Both Claude Opus 4.6 and the self-generated guidelines drive almost all trajectories to a clean normal exit ($91.53\% / 92.13\%$) and cut the step budget to $\sim 61\%$ and the token budget to $\sim 46\%$, while GPT-5.5 produces a higher resolve rate (67.53% vs. $65.40\% / 65.93\%$) at the cost of noticeably weaker efficiency (normal exit 79.33%, step budget 70.94%, token budget 58.13%). In other words, at 27B the choice of supervisor mainly trades a small amount of resolve rate for a large amount of efficiency, but the self-generated supervisor remains Pareto-competitive on every axis. Combined with the trend on 4B/9B, this confirms that AdaScaffold scales naturally with model capability and that using the agent's own model as the supervisor is a robust default across model sizes.

Table 8: Results of estimating Θ on different datasets (showing the results of 4B model as the example). The best results are marked in **bold**.

	Resolve Rate (%)↑	Normal Exit (%)↑	Avg Steps↓	Avg Tokens (M)↓
Original Scaffold	34.73 ± 1.03	7.00 ± 1.11	76.46 ± 0.28	2.17 ± 0.03
AdaScaffold	**39.33** ± 1.04	33.93 ± 0.75	**66.81** ± 0.32	**1.69** ± 0.03
w/ R2E-Gym	38.47 ± 1.47	**34.40** ± 1.41	69.92 ± 0.39	1.85 ± 0.03
w/ Rebench	38.33 ± 1.13	33.20 ± 2.68	67.63 ± 0.29	1.75 ± 0.01

C.2 Results on Larger-scale and More Capable Models

For GLM-5.2 (Zeng et al., 2026), we adopt the same hyperparameter settings as in the main experiments: at most 80 agent steps with a hard cap of 100, a context window of 65,536 tokens per completion, a per-task wall-clock budget of 7,200 seconds, a per-task LLM budget of 1,200 seconds, a 90-second timeout for each tool execution, a sampling temperature of 0.6, and top-p of 0.95. Function calling is disabled.

To better align with the reasoning profile of frontier-scale models, AdaScaffold naturally accommodates different intervention intensities. For a model with strong intrinsic problem-solving capabilities like GLM-5.2, rigid behavioral constraints are not required; instead, lightweight, advisory interventions (i.e., Interventions B, D, G, and H) are sufficient to steer the agent's execution flow.

As shown in Table 7, AdaScaffold achieves comparable resolve rate to the baseline while substantially enhancing autonomous task completion, boosting the normal-exit rate from 31.73% to 52.83%. Crucially, this is accomplished with marked efficiency gains, trimming both the average trajectory steps and overall token consumption to roughly 85% of the baseline. This demonstrates that AdaScaffold functions effectively as an efficiency-oriented scheduler for frontier models, streamlining redundant exploration without compromising solution quality.

C.3 Using Different Datasets to Estimate Θ

Table 8 reports an ablation that probes the generalization of our *Intervention Timing Estimation* procedure. We take the 4B agent as a representative case and, instead of fitting Θ on SWE-bench-Verified itself, we estimate it on two out-of-distribution trajectory pools–R2E-Gym (Jain et al., 2025) and Rebench (Wijk et al., 2024)–and then deploy the resulting parameters as the intervention guidelines during SWE-bench-Verified evaluation. This setting simulates the realistic situation in which the test distribution is entirely unseen during scaffold design, and additionally lets us measure how well the estimated parameters transfer across datasets. For reference, we report the baseline and AdaScaffold with parameters estimated on the matching SWE-bench-Verified trajectories.

Two findings emerge. First, transferring Θ across datasets incurs only a small performance loss: resolve rate drops from 39.33% (in-distribution) to 38.47% / 38.33% (R2E-Gym / Rebench), the normal-exit rate stays essentially unchanged (33.93% $\rightarrow$ 34.40% / 33.20%), and the step or token savings are slightly attenuated (66.81 $\rightarrow$ 69.92 / 67.63 steps; 1.69 M $\rightarrow$ 1.85 M / 1.75 M tokens). Second, all three AdaScaffold variants remain substantially better than the Original Scaffold baseline (e.g. +3.6-4.6 absolute points in resolve rate, 4-5× normal-exit rate, 9-13% fewer steps and 15-22% fewer tokens). This shows that AdaScaffold is effective even when the test distribution is completely unseen, and that the estimated parameters generalize well across heterogeneous trajectories.

C.4 Using Different Numbers of Samples to Estimate Θ

Figure 8 shows how the four risk-guideline parameters estimated by our pipeline vary with the number of trajectories used for estimation, separately for the Qwen-3.5 4B, 9B and 27B agents. From left to right the results correspond to: (1) Parameter A in Table 1, the look-back window (in steps) used by Intervention A to break a stuck loop; (2) Parameter B in Table 1, the fraction of the step budget after which the supervisor warns the agent that no edit has been produced; (3) Parameter

D in Table 1, the budget fraction after which the supervisor forces the agent to start writing a patch if no patch exists; and (4) Parameter E in Table 1, the budget fraction after which the supervisor forces submission. For each model we sweep the estimation sample size $m \in \{10, 20, 50, 100, 150, 200\}$, drawing trajectories uniformly at random; lines show the mean and shaded bands the 1σ envelope over repeated samples.

Two observations emerge. First, the mean estimates are remarkably stable across m: even at $m = 10$ the curves already approximate the values obtained at $m = 200$. The benefit of more samples is therefore not a shift in the central estimate but a shrinkage of the standard deviation, which yields more reproducible risk parameters. Based on this trade-off between estimation cost and stability, we adopt $m = 100$ throughout the paper, which lies on the knee of the variance curve for all three model sizes. Second, the parameter values change systematically with model scale: the larger 27B agent plans in longer, more deliberate editing sequences, so its estimated Parameter A and Parameter B are smaller (loops are flagged sooner and the no-edit warning fires earlier), while its Parameter E is larger (the supervisor allows more budget before forcing submission, leaving room for the agent's longer edit phase). This is consistent with the qualitative picture that the supervisor should intervene earlier on cheap behaviours (looping, idling) but give more capable models additional time to finish editing before being pushed to submit.

D Discussion and Future Work

D.1 What AdaScaffold actually changes

Our experiments suggest that AdaScaffold improves coding agents not by augmenting their underlying reasoning ability, but by reshaping the trajectory itself: it reallocates the agent's step and token budget away from unproductive looping and over-verification, and toward earlier editing and timely submission. This is consistent with two recurring patterns observed across model scales. The gains on the resolve rate concentrate on tasks of low-to-medium difficulty rather than fundamentally harder ones, and the dramatic improvement in the normal-exit rate shows that most of the win comes from converting budget-induced terminations into clean, model-issued submissions. Adaptive scaffolding is therefore best viewed as a *scheduler* that helps an agent realize its existing capability under finite budgets, complementary to model-, data- and prompt-level improvements.

D.2 Why a behavior-only criterion is enough

A perhaps surprising finding is that intervention timing can be reliably estimated from a handful of unlabeled trajectories using only a behavior-derived risk proxy and a closed-form covariation criterion. The criterion is interpretable, has well-understood statistical properties, and transfers across heterogeneous trajectory pools (e.g., from R2E-Gym or Rebench to SWE-Bench-Verified) with only minor performance loss. This indicates that, for a fixed scaffold and a fixed model, "risky" trajectory patterns are stable enough to be learned without reward signals, which is a desirable property in realistic deployments where ground-truth tests are unavailable, expensive, or proprietary.

D.3 Future work

Several directions naturally extend this work. First, the intervention family itself could be discovered automatically from trajectory clusters, rather than designed by hand, turning AdaScaffold into a fully data-driven scaffold-generation pipeline. Second, the timing estimation could be extended from trajectory-level statistics to per-instance statistics, yielding a truly per-instance scaffold rather than a model-level one. Third, AdaScaffold is orthogonal to model fine-tuning, RL-based agent training, and inference-time search (e.g., best-of-n, tree search): combining adaptive scaffolds with these techniques is a promising route to further reduce the cost-quality gap of coding agents. Finally, it would be interesting to study whether the same covariation criterion generalises beyond coding agents to other long-horizon tool-using settings such as web agents and scientific-discovery agents.