DO PERSONALIZED SKILLS HELP CODING AGENTS? AN EMPIRICAL STUDY OF DEVELOPER INTERACTION HISTORIES

Shuyan Huang
UMass Amherst
shuang@umass.edu

Kai Du
OpenRefinery.ai
kai@openrefinery.ai

Andrew Lan
UMass Amherst
andrewlan@umass.edu

ABSTRACT

Large language model (LLM)-powered agents have rapidly evolved from code-completion tools into solvers of complex software engineering (SWE) tasks. As human developers collaborate with these coding agents over time, personalized preferences emerge, which can be used to adapt the agents' coding behavior to better meet the needs of human developers. Capturing and reusing such preferences across interactions may reduce repeated corrections and improve developer-agent collaboration efficiency. Agent skills provide a lightweight mechanism for transferring experience without modifying model parameters and have been proven to be beneficial. However, existing work on learning agent skills mostly focuses on task-specific skills; it remains unclear whether there can be a personalized element, i.e., developer-specific skills distilled from each developer's interaction histories, that effectively generalize to future tasks. Therefore, we propose a two-stage framework for generating personalized agent skills that extract reusable, developer-specific preferences from developer-agent interaction traces: First, we use rule-based bootstrapping and trace-based refinement to extract personalized skills. Second, we design a reproducible replay framework that evaluates the effectiveness of extracted skills through an interactive, LLM-based human developer simulator. We conduct an experiment on 206 real-world, human developer-coding agent interaction sessions from 13 developers and compare personalized skills against no-skill, generic-skill, and other-user-skill baselines. Experimental results show that personalized skills provide only limited and inconsistent improvements over the no-skill baseline, whereas generic skills distilled from interaction traces across developers consistently achieve good performance. Further analysis suggests that personalized skills become more effective when developer preferences manifest frequently, especially when they work on similar tasks over time, making it possible to extract skills from past tasks that are generalizable to future ones. These findings provide empirical insights into when developer-specific personalization is effective and demonstrate that, in practice, broadly transferable procedural knowledge can be more robust than developer-specific preference signals.

1 Introduction

Large language model (LLM) agents have rapidly evolved from code-completion tools into interactive systems for solving complex software engineering (SWE) tasks, such as navigating repositories and iteratively repairing failed solutions [Yang et al., 2024, Jimenez et al., 2024, Wang et al., 2025]. As developers collaborate with these agents, they often have distinct expectations regarding how code changes should be implemented. In practice, these preferences are often implicit at first and gradually emerge through repeated interactions with the agent on multiple tasks. An agent capable of inferring developer preferences from past interaction trajectories has the potential to adapt its behavior accordingly, improving the efficiency of developer-agent collaboration. For example, a developer may consistently prefer minimal, localized changes over broad refactoring, even when this preference is not explicitly stated in every task. An agent can proactively incorporate these preferences into its thinking and implementation processes, thereby reducing the need for developers to repeatedly specify the same preferences and provide similar feedback across tasks.

Agent skills have recently emerged as a promising mechanism for retaining and transferring experience, by packaging reusable procedural knowledge as structured instructions for LLM-based agents [Han et al., 2026, Xu and Yan, 2026]. Current methods distill and optimize the skills from execution trajectories to improve coding-agent performance without updating the underlying model parameters [Ni et al., 2026, Yang et al., 2026a, Wang et al., 2026]. These

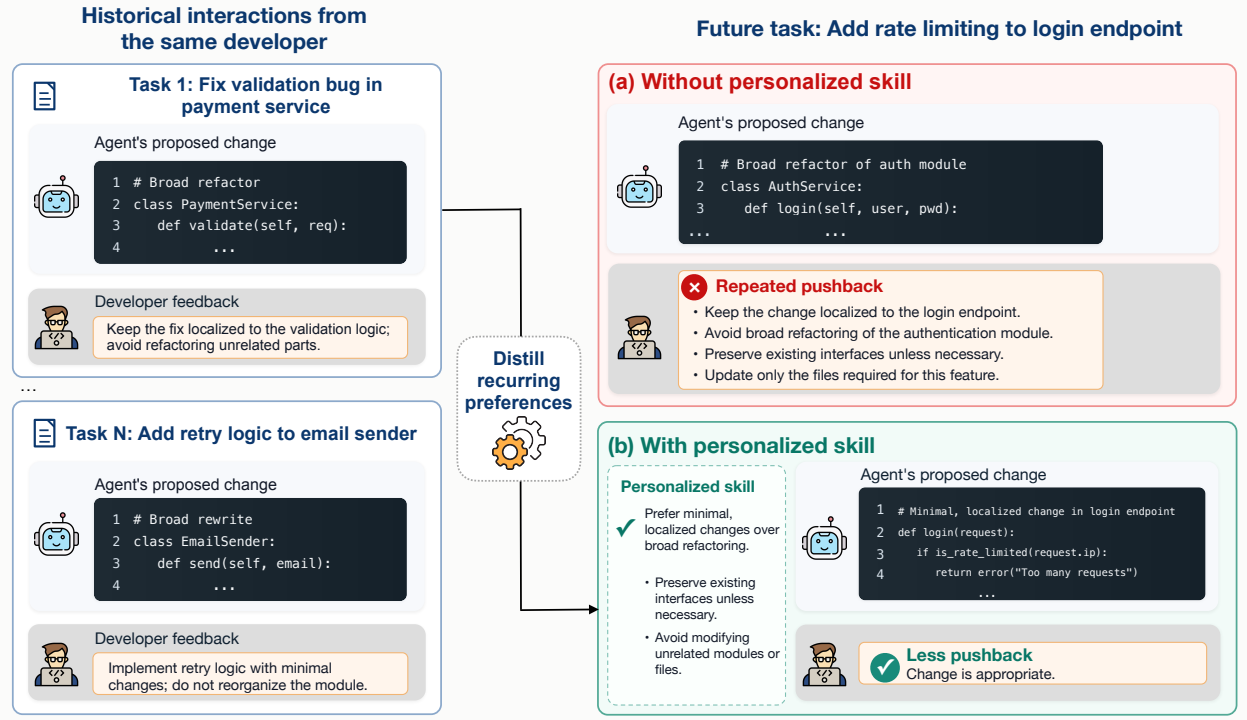

Figure 1: Illustration of personalized skill distillation from prior developer–agent interactions and its application to a future task.

approaches primarily distill domain- or task-specific knowledge to improve agent performance at a specific task. However, in practice, each human developer may work on a diverse collection of tasks that may not have significant overlap. It remains unexplored whether a task-independent, yet developer-specific natural-language skill distilled from a developer's interaction history can generalize across tasks from the same developer.

To bridge this gap, we investigate whether we can learn personalized agent skills that capture transferable developer-specific preferences rather than task-specific behaviors. As illustrated in Figure 1, recurring feedback across a developer's prior tasks can reveal stable implementation preferences, which can be distilled into reusable guidance and applied to subsequent tasks. However, generating and evaluating effective personalized skills present two main challenges. First, a developer's interaction history contains a mixture of transferable preferences, task-specific requirements, and one-off corrections, making it challenging to distinguish reusable preferences from task-specific behaviors. Moreover, these preferences are often conveyed implicitly through feedback on specific implementations rather than explicit instructions, requiring the agent to infer the underlying preferences from contextual signals. Second, fairly evaluating personalized skills depends on comparing different skills under identical interaction conditions. This evaluation requires us to faithfully reconstruct the original developer-agent interaction session while isolating the effect of the injected skill. Accordingly, we investigate the following core research question:

> *Can we distill personalized skills from developer-agent interaction histories, and do they improve coding agent performance and reduce pushback on subsequent tasks?*

Contributions To answer this question, we propose a framework for generating personalized skills from individual developers' interaction traces. For skill construction, we adopt a two-stage procedure. We first construct a bootstrap skill using predefined rules that organize common preference dimensions into an initial structured skill. We then refine the skill based on the developer's historical interaction traces with the coding agent, distilling transferable developer-specific preferences by adding, revising, or removing instructions according to the available evidence. At inference time, the generated skills are incorporated into the agent's prompt to guide its behavior without modifying the underlying model parameters. For reproducible replay, we summarize the developer's requests from each target session into a concise task summary. Conditioned on the summary, an LLM-based user simulator progressively reveals requirements and provides feedback after each coding-agent response. We evaluate the framework on 206 real-world interaction sessions from 13 developers, comparing personalized skills with no-skill, generic-skill, and other-user-skill baselines. Experimental

results show that personalized skills provide limited and inconsistent gains, whereas generic skills distilled across developers perform better consistently. Further analysis suggests that personalization is more effective when preferences recur across similar tasks, indicating that broadly shared procedural guidance may be more robust when developer histories are sparse.

2 Method

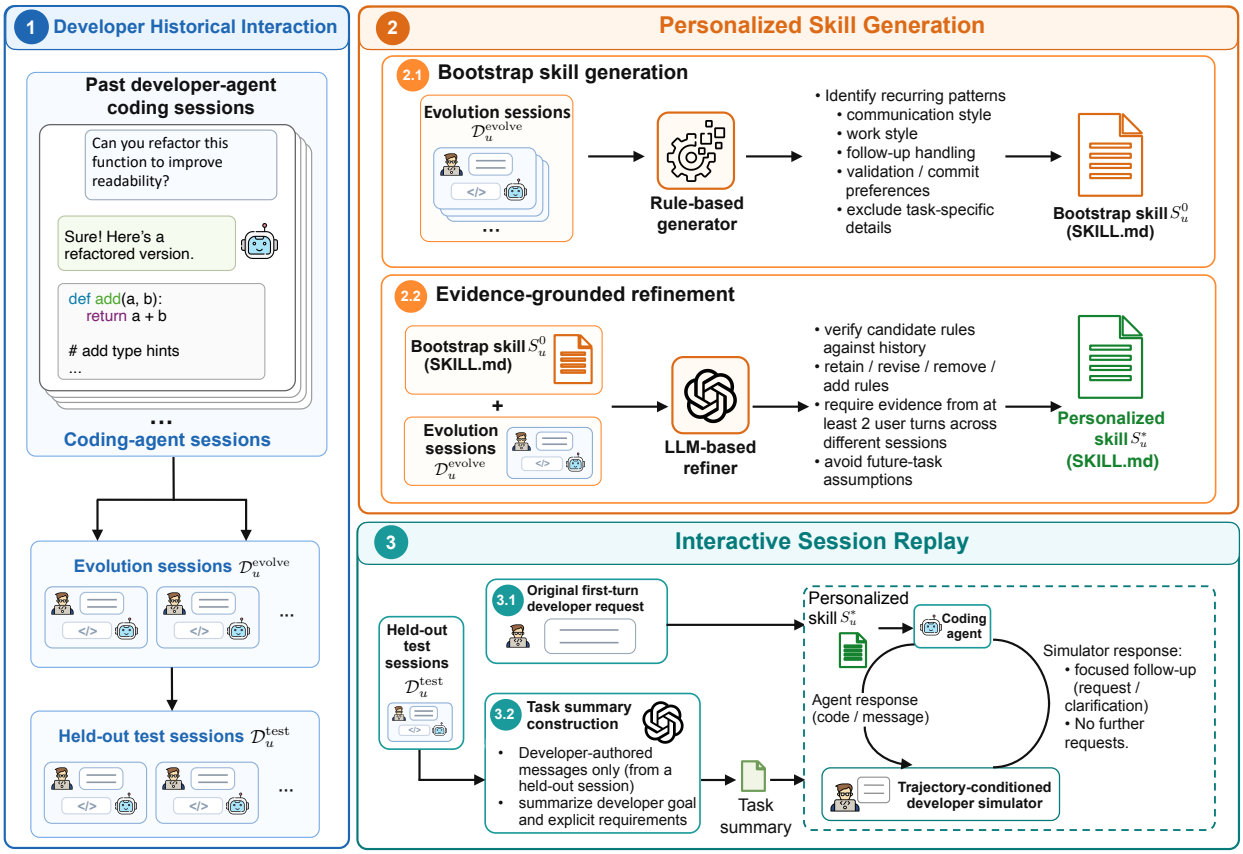

Figure 2: Overview of the personalized skill generation framework.

Figure 2 illustrates the overall framework. Given a developer's historical collaborative coding sessions with the coding agent, we first split them into an evolution set and a held-out test set. We use the evolution sessions to construct a personalized skill by first generating a bootstrap skill and then refining it with evidence extracted from the interaction history. We then evaluate the personalized skill by replaying the held-out test sessions with a trajectory-conditioned user simulator under different skill conditions to evaluate the effectiveness of the personalized skill.

2.1 Problem Formulation

Following prior work, we define a skill S as a human-readable `SKILL.md` document that encodes reusable, natural-language guidance for a coding agent at inference time, without modifying its underlying model parameters [Ni et al., 2026, Yang et al., 2026a]. In this work, a personalized skill captures the recurring preferences and expectations of a particular developer rather than task-specific experience. Formally, let $\mathcal{U}$ denote a set of developers. Each developer $u \in \mathcal{U}$ is associated with a set of coding-agent sessions $\mathcal{D}_u = \{d_{u,1}, \ldots, d_{u,n_u}\}$, where n_u denotes the number of sessions associated with developer u. We split $\mathcal{D}_u$ into an evolution set $\mathcal{D}_u^{\text{evolve}}$ and a disjoint, held-out test set $\mathcal{D}_u^{\text{test}}$. We use $\mathcal{D}_u^{\text{evolve}}$ to construct the personalized skill and reserve $\mathcal{D}_u^{\text{test}}$ for generalization evaluation. Specifically, given the evolution sessions of developer u, we construct a personalized skill $S_u^* = \mathcal{G}(\mathcal{D}_u^{\text{evolve}})$, where $\mathcal{G}$ denotes the personalized skill-generation procedure. We evaluate personalized skills on their performance, i.e., success when the coding agent has these skills, on developer tasks in the held-out test set.

2.2 Personalized Skill Generation

The procedure of personalized skill generation consists of two stages. We first generate a task-independent bootstrap skill from the evolution sessions and then refine it using evidence from the original trajectories:

$$S_u^0 = \mathcal{B}\left(\mathcal{D}_u^{\text{evolve}}\right), \qquad S_u^* = \mathcal{R}\left(S_u^0, \mathcal{D}_u^{\text{evolve}}\right).$$

Here, $\mathcal{B}$ denotes bootstrap construction and $\mathcal{R}$ denotes evidence-grounded refinement, including adding, revising, or removing instructions, which we detail below.

Bootstrap skill generation. To provide a stable initial skill and reduce content drift during subsequent refinement, we first use a fixed pattern analysis template to generate a task-independent bootstrap skill S_u^0 from the developer's evolution sessions $\mathcal{D}_u^{\text{evolve}}$. The template asks a backbone LLM to identify recurring patterns in communication, work style, follow-up handling, validation preferences, and commit behavior, thereby consolidating them into a structured `SKILL.md` file. The bootstrap skill captures how the developer tends to work rather than the specific tasks observed in the evolution sessions. We therefore exclude repository names, file paths, issue identifiers, commands, and other task-specific details.

Evidence-grounded refinement. To improve the reliability of the bootstrap skill and reduce overgeneralization, we next provide the bootstrap skill and the original evolution sessions to a second LLM for refinement. Inspired by prior approaches that distill reusable behavioral guidance from interaction traces and iteratively refine persistent instructions based on observed evidence [Ni et al., 2026, Yang et al., 2026a, Wang et al., 2026], we adopt an evidence-grounded refinement process to improve the bootstrap skill. Unlike the bootstrap stage, which prioritizes coverage, the refinement stage prioritizes evidence-based validation and generalizability. The refiner treats the evolution sessions $\mathcal{D}_u^{\text{evolve}}$ as the primary source of evidence and the bootstrap skill as an initial set of candidate rules. It verifies each candidate rule against the interaction history, retaining, revising, or removing existing rules, meanwhile adding recurring preferences missed during bootstrap generation. To reduce overfitting, the refiner retains a rule only when it is supported by at least two independent user turns from different evolution sessions. The refinement stage focuses primarily on communication style, work style, and follow-up handling while avoiding assumptions about programming languages, frameworks, interfaces, or task types. The resulting skill guides the agent's behavior without overriding the active user request or the current task details. Accordingly, it must not introduce new task requirements, prescribe environment-specific commands, or justify implementing less than the user requested. The final output is a compact, task-independent `SKILL.md` intended to generalize to unseen sessions from the same developer.

2.3 Interactive Session Replay

Faithful evaluation of personalized skills requires the coding agent to use them on a developer's tasks. However, a recorded developer-agent session cannot be replayed verbatim, since the agent, armed with new skills, may follow a different trajectory from the agent in the original interaction. Therefore, using the original developer follow-up messages at fixed turns may not lead to a faithful representation of human behavior. Following prior work [Wu et al., 2026], we replay each held-out test session with an LLM-based developer simulator. The simulator draws on the original developer's task specifications but decides when and how to follow up based on the current session's trajectory. We detail various components of our replay setup below.

Task summary construction. For each held-out test session $d \in \mathcal{D}_u^{\text{test}}$, we generate a task summary from all developer-authored messages in the original session. The summary captures the developer's general goal and the requirements explicitly stated during the interaction. It gives the developer simulator a complete account of the developer's stated requirements while enabling the evaluated agent to discover later requirements through subsequent interaction. Since the summary is extracted from developer messages only, it excludes the original agent's reasoning, actions, and implementation choices. Only the developer simulator receives the task summary, whereas the coding agent receives the original, ground-truth first-turn developer request and encounters later requirements through interactive session replay.

Trajectory-conditioned developer simulation. After each agent response, the developer simulator receives the task summary and the current conversation with the agent. The simulator reviews the agent's latest response and determines whether it has addressed the developer's explicit requests. It then takes one of two actions: issue a focused follow-up for the most important unresolved requirement or return `No further requests.` For each follow-up, the simulator refers to what the evaluated agent has said or done and focuses on any remaining gaps. This setup keeps the interaction faithful to the current trajectory without changing the scope of the recorded session.

2.4 Skill-Conditioned Replay

For each held-out test session $d \in \mathcal{D}_u^{\text{test}}$, we replay the reconstructed interaction under each skill condition. The coding agent receives the corresponding skill before the developer's original first-turn request. The skill is kept unchanged during the session.

3 Experiments

In this section, we conduct experiments to evaluate the effectiveness of our personalized skill generation framework using the proposed replay evaluation setup.

3.1 dataset

Since this work focuses on developer-agent interaction, we evaluate the effectiveness of personalized skills on SWE-chat [Baumann et al., 2026], which contains 8,866 public command line (CLI) interface coding-agent sessions collected through Entire.io from public GitHub repositories between January and June 2026. To enable fully reproducible and verifiable replay, we remove sessions with incomplete interactions, inaccessible or unrecoverable repository states, private dependencies, missing files, or no substantive code edits. For each remaining session, we reconstruct the codebase as it existed before the agent began the task by checking out the parent of the commit containing the agent's final code changes, and create an isolated worktree for execution and validation. We also exclude answer-only, read-only, empty-tool and review-only sessions, and retain only developers with at least three valid tasks. After this filtering, the dataset contains *only 206 sessions from 13 developers*. We note that this strict filtering criterion establishes a rigorous evaluation setup but limits our ability to draw definitive conclusions. For each developer, we randomly split the sessions into an 80% evolution set and a 20% held-out test set to evaluate generalization to unseen tasks from the same developer, resulting in 164 evolution sessions and 42 test sessions. The distribution of task categories in the dataset is shown in Figure 3. The replayable dataset covers a diverse range of software engineering tasks, such as code review and targeted fixes (28.6%), feature implementation (20.9%) and testing/build/DevOps (15.0%).

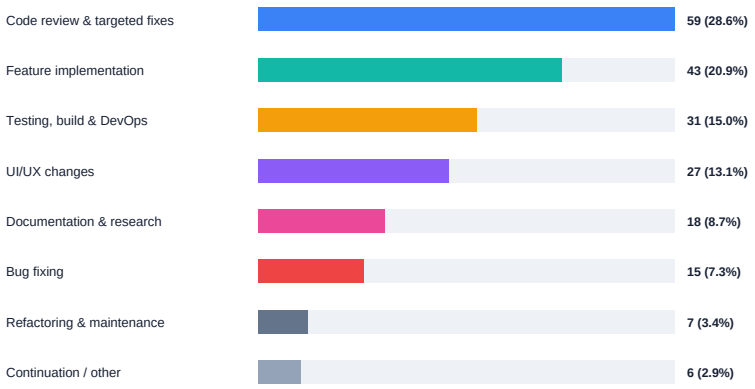

Figure 3: Distribution of task categories in our replayable SWE-chat dataset.

3.2 Experimental Setup

We use Codex [1] with GPT-5.5 for personalized skill generation, coding-agent execution, developer simulator, and task-completion scoring. The prompts used for all components are provided in Appendix C. We replay each held-out task with Codex CLI from the same initial repository commit, using an isolated worktree for every run. To make computational cost consistent across conditions, we limit each replay to at most six developer-agent interaction turns.

We consider four conditions: (A) no skill, (B) the target developer's personalized skill, (C) a personalized skill from another random developer, and (D) a generic skill pooled across all developers. For Condition D, the generic skill is generated from the pooled evolution sessions of all developers, including those of the target developer. All other components including the repository state, initial request, task summary, developer simulator, and agent configuration, remain the same across all experimental conditions. After each replay, we collect the interaction trajectory, code changes, and validation evidence, and assess task completion following the SWE-chat 100-point scoring rubric [Baumann et al.,

[1] https://chatgpt.com/codex/

2026], using LLM-as-a-judge. We repeat the experiment using five random seeds for the within-developer data split to reduce sensitivity to data partitions.

3.3 Quantitative Results

Table 1: Task-completion performance under four skill conditions. Follow-up Rate denotes the percentage of replay instances in which the simulated user issues at least one non-terminal follow-up after the initial request, with multiple follow-ups in the same replay counted once. Win/Tie/Loss denotes the percentage of paired replay instances in which a condition scores higher than, equal to, or lower than the no-skill baseline, respectively. Best results are shown in **bold**.

Condition	Score ↑	Follow-up Rate ↓	Win/Tie/Loss
(A) No skill	$65.02_{\pm 3.24}$	24.76% (52/210)	–
(B) Personalized skill	$65.99_{\pm 2.14}$	30.95% (65/210)	41.43/14.76/43.81% (87/31/92)
(C) Random developer skill	$65.94_{\pm 3.66}$	27.62% (58/210)	43.33/17.62/39.05% (91/37/82)
(D) Generic skill	$\mathbf{68.80_{\pm 2.26}}$	30.00% (63/210)	**50.95/14.76/34.29%** (107/31/72)

Table 1 reports the overall task-completion performance under the four skill conditions.

Generic skills provide the largest and most consistent gains. Generic skills achieve the highest average score (68.80), the largest improvement over the no-skill baseline (+3.78), and the highest win rate (50.95%). Although the improvement over the baseline does not reach the conventional significance threshold (paired t-test, $p = .063$), the consistently stronger performance suggests that pooling interaction histories across developers is more effective: doing so produces more robust and broadly transferable guidance than constructing separate skills from the limited interaction histories of individual developers. Additionally, the generic skill has a follow-up rate of 30.00%, compared with 24.76% for the no-skill baseline. This observation suggests that generic skills help most towards better task completion rather than towards higher developer–agent collaboration efficiency.

Developer-specific skills do not show clear benefit from personalization. Personalized skills yield only a modest improvement in task performance, with an average gain of 0.97 compared to the no-skill baseline. However, their win and tie rates are only 41.43% and 14.76% respectively, indicating that the improvement is inconsistent across held-out task instances. The overall improvement is not statistically significant (paired t-test, $p = .399$). Similarly, a skill distilled from another random developer achieves a comparable average gain of 0.92 with a similar win rate of 43.33% and a tie rate of 17.62% (paired t-test, $p = .451$). The comparable performance of the personalized and mismatched developer skills shows limited evidence that developer-specific information contributes additional benefits beyond generic procedural guidance. We hypothesize that this result may be due to the limited interaction history available for each developer, which makes it difficult to distinguish stable developer preferences from task-specific or one-off feedback and thereby limits generalization performance on held-out tasks.

4 Detailed Analysis

We analyze experimental results, focusing on the impact of relevant interaction history, the effectiveness of the developer simulator, interaction and execution behavior, and skill content. We discuss how limitations in available developer–agent interaction trace data prevent us from drawing more definitive conclusions. We also examine the effectiveness of evidence-grounded skill refinement and how skill effectiveness varies across developers in Appendix A and B.

4.1 Impact of Relevant Interaction History

We examine whether the number of sessions used for skill creation in the evolution set impacts the effectiveness of personalized skills. For each of the 42 held-out tasks in one random seed, we use an LLM to review all evolution sessions from the same developer and identify those that are semantically related to the held-out task. We then group the held-out tasks into four bins (0, 1-2, 3-5, and ≥ 6 relevant sessions) to analyze how the effectiveness of

Table 2: Performance of relevant evolution sessions. A: no skill; B: personalized skill; C: random developer skill; D: generic skill.

Relevant Sessions	N	B − A	B − C	B − D
0	3	−6.33	+15.00	−8.00
1–2	16	0.00	−1.38	−3.81
3–5	11	+0.10	+0.20	−7.20
≥ 6	12	+10.17	+8.92	+5.67

personalized skills varies with the number of relevant historical sessions. As reported in Table 2, the effectiveness of the personalized skill varies with the amount of relevant evolution sessions. When at least six relevant sessions are available, the personalized skill substantially outperforms both the no-skill baseline ($B - A = +10.17$) and the random developer skill ($B - C = +8.92$), and even surpasses the generic skill ($B - D = 5.67$). In contrast, when fewer than six relevant sessions are available, the personalized skill shows little or no advantage over either the no-skill baseline or the random developer skill, while the generic skill performs better in all three groups. These results suggest that personalized skills become effective only when a developer's interaction history contains a sufficient number of examples that are relevant to held-out evaluation tasks. It is possible that as larger-scale developer-agent interaction trace data becomes available, the potential of personalized skills can be fully realized.

4.2 Effectiveness of Developer Simulator

We also examine the effectiveness of the proposed developer simulator. Specifically, we compare the follow-up messages generated by the simulator with the corresponding real developer messages. Each pair of messages is classified by an LLM-as-a-judge evaluator according to a predefined semantic-matching rubric. We define three match levels. An *exact match* indicates that the simulated follow-up expresses all requirements contained in the real developer follow-up, allowing minor differences in wording or the consolidation of multiple turns. A *partial match* indicates that the two follow-ups share at least one requirement, but the simulated follow-up omits or introduces other requirements. A *mismatch* indicates that the two follow-ups contain different requirements with no substantive overlap. Across five seeds, 171 of the 210 replay instances contained substantive follow-ups from both the real developer and the simulator. As shown in Figure 4, among these instances, 59.65% were exact matches, 29.82% were partial matches, and 10.53% were mismatches. Thus, 89.47% were at least partially semantically consistent with the real developer messages. This rate ranged from 86.96% to 92.50% across the four experimental conditions, suggesting that our replay simulator generally captures the requirements expressed by real developers across different skill conditions.

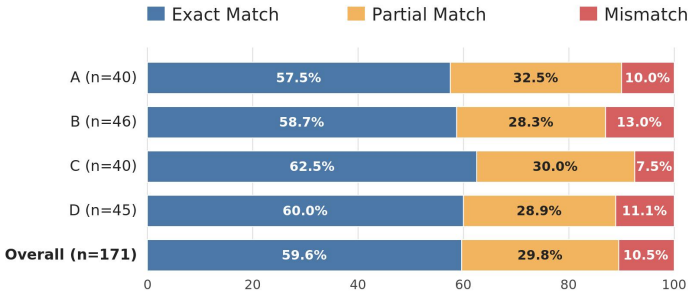

Figure 4: Semantic consistency between real developer messages and simulator-generated follow-ups. A: no skill; B: personalized skill; C: random user skill; D: generic skill.

4.3 Interaction and Execution Behavior

We further investigate how the skill conditions affect the agent's interaction and execution behavior. As shown in Table 3, a skill generally leads to more extensive execution rather than reducing interaction effort. Compared to the no-skill baseline, all three skill conditions produce more agent turns, follow-ups, generated tokens, tool calls, code changes, and validation attempts. More specifically, personalized skills do marginally increase the task-completion score from 65.02 to 65.99, but at the cost of increasing the average number of tool calls from 8.47 to 9.46 and execution time from 91.76 to 106.16 seconds. It changes more files and produces greater patch churn, while the number of unresolved requirement follow-ups increases from 0.29 to 0.37. Thus, personalized skills do not reduce the amount of interaction or computation required to complete a task. Nevertheless, skills lead to more systematic validation. Conditioned on the personalized skills, the average number of test command groups increases from 0.56 to 0.97 and the proportion of runs reporting successful validation rises from 43.1% to 58.9%. The random user and generic skills show similar trends. The generic skill achieves the highest task-completion score, but also uses the most tokens and produces the greatest patch churn. These results suggest that skills primarily encourage more extensive implementation and validation rather than improving execution efficiency.

4.4 Skill Content Analysis

We also compare the 13 personalized skills for each developer and the generic skill across all developers to better understand why the generic skill consistently outperforms personalized skills. As listed in Table 4, the generic skill contains substantially more rules (25 vs. 14.15 on average) and words per skill (383 vs. 236.38) than personalized skills. These two types of skills exhibit similar category distributions, with most rules concerning workflow, follow-up handling, and communication. However, the generic skill contains a larger proportion of commit-related rules (16.0% vs. 7.6%). Personalized skills are not simply specialized versions of the generic skill; their mean TF-IDF similarity to the generic skill is 0.517, while the mean similarity between different developers' skills is 0.443. Under our rule-based lexical criterion, 64.7% of personalized rules are unique to a single developer. There is substantial lexical variation

Table 3: Interaction and execution metrics under the four skill conditions. Values are averaged across evaluated replays. A: no skill; B: personalized skill; C: random user skill; D: generic skill.

Metric	A	B	C	D
Interaction and computation				
Agent turns	1.29	1.37	1.33	1.35
Unresolved-requirement follow-ups	0.29	0.37	0.33	0.35
Command/tool calls	8.47	9.46	8.69	8.98
Agent tokens	442,096	597,120	521,219	643,578
Execution time (s)	91.76	106.16	99.62	104.57
Code changes				
Files changed	1.65	2.22	1.79	1.99
Patch churn	29.08	37.11	32.32	37.98
Testing and validation				
Test command groups	0.56	0.97	0.86	0.93
Validation command groups	1.77	2.23	2.06	2.16
Runs reporting successful validation ↑	43.1%	**58.9%**	54.1%	57.4%

Table 4: Content statistics of the personalized and generic skills. The personalized results are averaged across the skills of 13 developers.

Skill property	Personalized	Generic
Skill size and length		
Rules per skill	14.15	25.00
Words per rule	16.70	15.32
Rule words per skill	236.38	383.00
Rule-category distribution		
Communication rules	23.4%	20.0%
Workflow rules	30.4%	28.0%
Validation rules	10.3%	12.0%
Follow-up rules	28.3%	24.0%
Commit rules	7.6%	16.0%
Similarity and specificity		
Similarity to generic skill	0.517	1.000
Similarity between developers	0.443	–
Unique developer-specific rules	64.7%	–

in personalized skills across developers. Nevertheless, lexical uniqueness alone does not imply that a rule is broadly applicable or relevant to held-out tasks, nor does it guarantee that the coding agent will follow the rule during execution. These observations suggest that the generic skill benefits from broader coverage of reusable developer practices, whereas many developer-specific rules may have limited opportunities to influence held-out tasks. These observations are consistent with our earlier hypothesis in Section 3.3 that limited per-developer interaction histories may constrain the coverage of personalized skills, whereas pooling interactions across developers enables the generic skill to capture a broader range of reusable guidance.

5 Related Work

5.1 Interactive Coding Agents

As LLM-based agents have demonstrated increasing capability in solving complex SWE tasks, many studies have developed benchmarks that provide agents with task specifications and evaluate the resulting repository state [Jimenez et al., 2024, Yang et al., 2024, Wang et al., 2025]. Most recent work has moved toward more realistic interactive settings [Tang et al., 2026]. For example, SWE-chat collects real coding-agent sessions from open-source development environments, providing an empirical understanding of how AI agents perform in real developer workflows [Baumann et al., 2026]. Based on SWE-chat, SWE-Together reconstructs verifiable multi-turn tasks and user feedback from the collected sessions, enabling the evaluation of both final task correctness and the amount of user guidance required

during interaction [Wu et al., 2026]. SWE-Interact transforms existing single-turn software engineering benchmarks into interactive tasks in which a simulated user progressively reveals requirements and provides feedback during execution, evaluating whether agents can discover user intent, adapt to evolving requirements, and build on their own prior work [Raghavendra et al., 2026]. Whereas these studies focus on within-session interaction and adaptation, we investigate cross-session adaptation by examining whether experience accumulated across a developer's prior sessions can guide agent behavior on subsequent tasks.

5.2 Agent Skills and Skill Evolution

Agent Skills encode reusable procedural knowledge in structured packages of instructions, code, and supporting resources [Xu and Yan, 2026, Jiang et al., 2026, Li et al., 2026, Cho et al., 2026, Zhao et al., 2026]. Since skills reside outside the model parameters, they provide a lightweight mechanism for adapting agent behavior without parameter updates. Recent work has explored automatically deriving and improving skills from agents' prior execution experience [Alzubi et al., 2026, Liu et al., 2026, Shen et al., 2026]. For example, Trace2Skill uses inductive reasoning over agent experience to consolidate multiple execution trajectories into a unified skill directory [Ni et al., 2026]. SkillOpt iteratively refines natural-language skill documents using rollout feedback and validation-gated edits [Yang et al., 2026a]. SkillGrad formulates skill improvement as gradient-descent-like optimization, converting trajectory-level diagnoses into textual gradients that guide structured skill revisions [Wang et al., 2026]. Some studies further suggests that skills learned from narrow experience may become overly specialized, whereas skills distilled from more diverse traces can transfer more reliably [Belikova et al., 2026]. These methods primarily aim to extract generalizable knowledge that improves performance across tasks within a domain or benchmark. In contrast, we generate skills from an individual developer's interaction history, focusing on recurring user-specific preferences and expectations rather than broadly applicable task-solving procedures.

5.3 Personalized LLM Agents

Previous work has personalized LLM-based agents using user profiles, persistent memory, and feedback from past interactions [Xu et al., 2026, Westhäußer et al., 2025, Liang et al., 2026, Qiu et al., 2026]. These systems typically summarize user information, retrieve relevant preferences during inference, and update stored memories as new feedback become available. AdaMem further studies what information should be retained over long interaction histories [Chen et al., 2026]. Personalization has also recently attracted attention in software engineering agents. For example, ToM-SWE employs a user-modeling agent to infer developer goals, constraints, and preferences from instructions and interaction history and store them in persistent memory [Zhou et al.]. Hedwig learns behavioral guidelines from developer interactions across sessions to adjust the autonomy of coding agents [Shukla et al., 2026]. TRACE converts user corrections into persistent, developer-specific rules that can be enforced on future tasks [Zhou et al., 2026]. AutoSkill identifies recurring preferences and requirements from dialogue histories and encodes them as reusable skills for future interactions [Yang et al., 2026b]. Unlike these approaches, our work focuses on learning task-independent personalized skills from developers' coding-agent interaction histories and systematically evaluates whether such skills generalize to unseen tasks from the same developer.

6 Conclusions and Future Work

In this paper, we investigate whether personalized skills distilled from prior developer-agent interactions can improve coding-agent performance. The experimental results show limited benefits from developer-specific personalization: personalized skills yield small and inconsistent gains, while a generic skill pooled across developers provides the largest and most consistent improvement. The further analyses suggest that limited per-developer interaction histories may make it difficult to identify stable and transferable developer preferences, although personalization appears more promising when multiple relevant historical examples are available.

Overall, our findings suggest that skill conditioning can improve coding agents, but the current benefits arise primarily from reusable guidance shared across developers rather than from developer-specific personalization. We expect that, as more real-world datasets of developer-agent interaction session trace data become publicly available, there are several promising avenues for future work. First, one can examine whether longer interaction histories, with extensive follow-up specifications after the first turn, support more reliable personalization. Second, one can investigate other methods for integrating skills into coding agents in stead of directly including them as part of the prompt. Since the sessions we analyzed are primarily GitHub-related work, formulating skills as reusable Python functions is not applicable. Third, one can explore more adaptive mechanisms for skill utilization, such as retrieving task-relevant guidance from a repository of skills mined across developers and tasks. Combining it with a small set of developer-specific preferences may be the right combination for a new task.

References

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026.

Julia Belikova, Rauf Parchiev, Evgeny Egorov, Grigorii Davydenko, Gleb Gusev, Andrey Savchenko, and Maksim Makarenko. Managing procedural memory in llm agents: Control, adaptation, and evaluation. *arXiv preprint arXiv:2606.23127*, 2026.

Xingyu Chen, Rui Wang, Zhaopeng Tu, and Liefeng Bo. Adamem: Learning what to remember for personalized long-horizon llm agents. *arXiv preprint arXiv:2606.21144*, 2026.

Hongcheol Cho, Ryangkyung Kang, and Youngeun Kim. Skillret: A large-scale benchmark for skill retrieval in llm agents. *arXiv preprint arXiv:2605.05726*, 2026.

Tingxu Han, Yi Zhang, Wei Song, Chunrong Fang, Zhenyu Chen, Youcheng Sun, and Lijie Hu. Swe-skills-bench: Do agent skills actually help in real-world software engineering? *arXiv preprint arXiv:2603.15401*, 2026.

Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

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

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Kaiqu Liang, Julia Kruk, Shengyi Qian, Xianjun Yang, Shengjie Bi, Yuanshun Yao, Shaoliang Nie, Mingyang Zhang, Lijuan Liu, Jaime Fernández Fisac, Shuyan Zhou, and Saghar Hosseini. Learning personalized agents from human feedback. *arXiv preprint arXiv:2602.16173*, 2026.

Xingyan Liu, Xiyue Luo, Linyu Li, Ganghong Huang, Jianfeng Liu, and Honglin Qiao. Skillforge: Forging domain-specific, self-evolving agent skills in cloud technical support. *arXiv preprint arXiv:2604.08618*, 2026.

Jingwei Ni, Yihao Liu, Xinpeng Liu, Yutao Sun, Mengyu Zhou, Pengyu Cheng, Dexin Wang, Erchao Zhao, Xiaoxi Jiang, and Guanjun Jiang. Trace2skill: Distill trajectory-local lessons into transferable agent skills. *arXiv preprint arXiv:2603.25158*, 2026.

Libin Qiu, Zhirong Gao, Junfu Chen, Yuhang Ye, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Shuo Tang. Autorefine: From trajectories to reusable expertise for continual llm agent refinement. *arXiv preprint arXiv:2601.22758*, 2026.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-INTERACT: Reimagining SWE benchmarks as user-driven long-horizon coding sessions. *arXiv preprint arXiv:2606.30573*, 2026.

Shuaike Shen, Wenduo Cheng, Mingqian Ma, Alistair Turcan, Martin Jinye Zhang, and Jian Ma. Skillfoundry: Building self-evolving agent skill libraries from heterogeneous scientific resources. *arXiv preprint arXiv:2604.03964*, 2026.

Tanjal Shukla, Kevin Feng, Leijie Wang, Mohammad Rostami, and Amy Zhang. Hedwig: Dynamic autonomy for coding agents under local oversight. In *Proceedings of the ACM Conference on AI and Agentic Systems*, pages 1293–1299, 2026.

Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. Programming by chat: A large-scale behavioral analysis of 11,579 real-world ai-assisted ide sessions. *arXiv preprint arXiv:2604.00436*, 2026.

Hanyu Wang, Yifan Lan, Bochuan Cao, Lu Lin, and Jinghui Chen. Skillgrad: Optimizing agent skills like gradient descent. *arXiv preprint arXiv:2605.27760*, 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919, 2025.

Rebecca Westhäußer, Wolfgang Minker, and Sebastian Zepf. Enabling personalized long-term interactions in LLM-based agents through persistent memory and user profiles. *arXiv preprint arXiv:2510.07925*, 2025.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026.

Yue Xu, Qian Chen, Zizhan Ma, Dongrui Liu, Wenxuan Wang, Xiting Wang, Li Xiong, and Wenjie Wang. Toward personalized LLM-powered agents: Foundations, evaluation, and future directions. *arXiv preprint arXiv:2602.22680*, 2026.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026a.

Yutao Yang, Junsong Li, Qianjun Pan, Bihao Zhan, Yuxuan Cai, Lin Du, Jie Zhou, Kai Chen, Qin Chen, Xin Li, et al. Autoskill: Experience-driven lifelong learning via skill self-evolution. *arXiv preprint arXiv:2603.01145*, 2026b.

Xinyu Zhao, Zhen Tan, Vaishnav Tadiparthi, Nakul Agarwal, Kwonjoon Lee, Ehsan Moradi Pari, Hossein Nourkhiz Mahjoub, and Tianlong Chen. Generative skill composition for llm agents. *arXiv preprint arXiv:2606.32025*, 2026.

Xuhui Zhou, Valerie Chen, Zora Zhiruo Wang, Graham Neubig, Maarten Sap, and Xingyao Wang. Tom-swe: User mental modeling for software engineering agents, 2026. *URL https://arxiv. org/abs/2510.21903*.

Yujun Zhou, Kehan Guo, Haomin Zhuang, Xiangqi Wang, Yue Huang, Zhenwen Liang, Pin-Yu Chen, Tian Gao, Nuno Moniz, Nitesh V Chawla, et al. Getting better at working with you: Compiling user corrections into runtime enforcement for coding agents. *arXiv preprint arXiv:2606.13174*, 2026.

A Effectiveness of evidence-grounded skill refinement

Table 5: Direct comparison between the rule-based bootstrap skill and its LLM-refined version. Differences and win/tie/loss counts are computed per session as Condition B − F.

Skill variant	Score ↑	Follow-up Rate ↓	Win/Tie/Loss
(F) Rule-based bootstrap	$65.71_{\pm 1.65}$	30.95% $(65/210)$	–
(B) Bootstrap + LLM refinement	$65.99_{\pm 2.14}$	30.95% $(65/210)$	$42.86/13.33/43.81\%$ $(90/28/92)$

Since we first generates a rule-based bootstrap skill and then uses an LLM to refine it based on the original evolution sessions, we examine whether this refinement improves skill reliability and mitigates overgeneralization. As shown in Table 5, the refined skill achieves a slightly higher average score than the bootstrap skill (65.99 vs. 65.71), corresponding to an improvement of 0.28. Both variants have the same follow-up rate of 30.95%, indicating that refinement does not reduce the need for additional user clarification. Moreover, the refined skill yields a nearly balanced win/tie/loss distribution relative to the bootstrap skill ($90/28/92$), and the difference is not statistically significant (paired t-test, $p = .792$). These results suggest that LLM-based refinement may modestly improve task performance. However, richer interaction histories are needed to determine whether this gain is consistent and statistically reliable.

B Developer-Level Variation in Skill Effectiveness

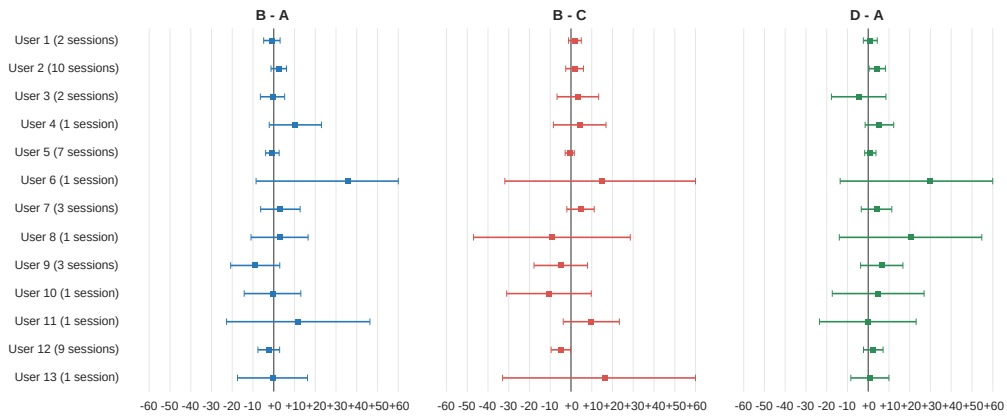

Figure 5: Developer-level differences in task-completion scores across skill conditions (A: no skill; B: personalized skill; C: other-user skill; D: generic skill). Square markers indicate mean score differences, and horizontal bars show descriptive 95% intervals.

We further examine how skill effectiveness varies across developers. As shown in Figure 5, the effect of personalized skills varies substantially. Comparing the personalized-skill and no-skill conditions ($B - A$), only 6 of the 13 developers achieve higher task-completion scores under the personalized-skill condition than under the no-skill baseline. Similarity, comparing the target developer's skill with a skill distilled from another developer ($B - C$) reveals no consistent evidence of a personalization benefit. Although the personalized skill achieves higher task-completion scores than the mismatched skill for 8 of the 13 developers, the largest improvements tend to occur for developers with fewer held-out sessions, whose estimates also exhibit substantially wider confidence intervals. In contrast, the generic skill shows more

consistent improvements, outperforming the no-skill baseline for 11 of the 13 developers. This observation suggests that limited developer-agent interaction session trace data make it difficult to reliably identify developer-specific preferences that generalize to held-out tasks.

C Prompts used in the personalized skill generation framework

The prompts for evidence-grounded skill refinement, coding-agent execution, task summary, developer simulator and task-completion scoring are shown in Figure 6–10.

Evidence-grounded skill refinement prompt

```
You are refining a coding-agent skill profile for a specific SWE-chat user.

Your job is to make the draft skill slightly more accurate and useful while keeping it
task-independent,
compact, and safe for unseen test sessions that may involve different repositories or task types.

## Draft skill (generated by pattern analysis)

{bootstrap_skill}

## User's training sessions ({n_sessions} sessions, {n_turns} user turns)

{session_block}

## Task

Produce a single improved SKILL.md.

Rules:
- Preserve the existing YAML front-matter format (--- name: ... description: ... ---).
- Preserve the Scope section's safety intent: the skill adapts behavior only, and the active user
request, local repository, and replay worktree remain authoritative.
- Treat the session block as the ground truth. The draft skill is a heuristic starting point — accept,
revise, or delete each rule based on whether it is confirmed by recurring evidence in the session
block.
- Remove any bootstrap rule that is not supported by at least two independent user turns across
different sessions. Unsupported rules cause more harm than a shorter skill.
- Add rules for clear patterns visible in the session block that the draft skill missed.
- Focus refinement primarily on Communication, Work Style, and Follow-Up Handling.
- Only adjust Validation or Commits when the training sessions show a recurring user preference, and
keep those rules conditional on the active request.
- Stay task-independent: capture *how* the user works, not *what* they asked for in training.
- Capture the user's *style* preferences: communication tone, directness, edit scope, claim precision,
follow-up handling, commit hygiene when requested, and validation style when relevant.
- Generalize only stable behavior patterns supported by multiple user turns or clearly recurring
interaction style.
- Do NOT infer that future test sessions will use the same repo, files, UI surfaces, tools, issues,
frameworks, languages, domains, or task type as training sessions.
- Do NOT mention specific repos, filenames, paths, issue IDs, commit hashes, exact labels, exact user
examples, URLs, logs, metrics, or task content from above.
- Do NOT add product requirements, feature requirements, validation requirements, or delivery
requirements unless they are explicitly conditional on the active request.
- Do NOT prescribe specific CLI commands, tool names, or environment-dependent steps.
Validation guidance must be conditional ("prefer the narrow relevant check when available", "if
tests exist and are relevant, run them")
rather than prescriptive ("always run X before finishing", "run the repository's final check
command").
- Do NOT generate guidance that causes the agent to stop or report failure merely because a tool,
file, repo layout, or CLI is unavailable. The execution environment may be incomplete; missing
resources should lead to scoped investigation, fallback to available evidence, or a precise
blocker only when the task truly cannot proceed.
- Do NOT generate guidance that could cause the agent to implement less than what the user explicitly
requested. Any "scoped" or "minimal" edit guidance must mean "don't add unrelated changes beyond
the request" — never "minimize what was asked for". Never produce rules that could justify
under-delivering on the task scope when the user has clearly stated what they want.
- Do not expand the skill with environment setup instructions, project-specific workflows, or
task-domain playbooks.
- Avoid long or redundant skills. Keep only guidance that would help on an unseen task for this user.
- Training-Split Examples, if included, must be short and sanitized. They must illustrate interaction
style only, not preserve task details.

Output only the final SKILL.md content, no commentary.
```

Figure 6: Prompt for evidence-grounded skill refinement.

Coding-agent execution prompt

```
You are replaying a SWE-chat coding session in an isolated local worktree.

Your goal is to complete the user's task in this repository as a coding agent.

Replay rules:
- Complete the full task described below in a single coherent pass.
- Treat the local worktree as the task's starting source state.
- Use repository evidence as the authority: inspect files, existing patterns, scripts, tests, and git
history as needed.
- Do not use hidden assumptions, private local paths, external authenticated state, or information not
present in the task description.
- Do not push, open pull requests, publish packages, or mutate remote services.
- If a command would require unavailable credentials or external state, report that limitation and
continue with local evidence.
- Do not claim success for work that is not supported by local evidence.
- Keep the replay bounded: avoid broad formatting, linting, or test commands when a narrow
repository-native check or direct file evidence is enough.
- If verification is blocked or slow, record the exact limitation instead of repeatedly trying
environment repairs.

First user message:
{first_user_message}
```

Figure 7: Prompt for coding-agent execution.

Task summary generation prompt

```
You are given all messages a user sent to a coding agent during one coding session.
Write a concise task summary focused on what the agent must deliver and how to verify it.

Instructions:
- Do not copy the user's messages verbatim.
- Do not include personal style preferences or incidental chat mechanics.
- Preserve task-relevant anchors such as the affected subsystem, feature area, command/tooling
surface, document type, or route/component category.
- Omit brittle details such as exact line numbers, long absolute paths, random IDs, commit hashes, and
exact numeric counts unless they are central to the task.
- Focus on the actionable task the user expected the agent to complete by the end of the session;
include important constraints and corrections, but do not turn exploratory discussion into mandatory
work.
- Do not reference the original agent's approach, intermediate steps, or solution details.
- Do not turn exploratory questions, tentative options, or agent-proposed ideas into mandatory
requirements unless the user clearly adopted them.
- Treat commit, push, PR creation, deployment, or publishing as required only when the user explicitly
requested them.
- Treat verification as required when the user explicitly requested it or when it is necessary to
confirm the implemented behavior; otherwise phrase it as an expected check when feasible.
- If a requirement depends on unavailable repository/tooling/remote access, phrase the success
criterion as completion when feasible or a precise blocker report when blocked.
- Keep success criteria focused on the minimum observable outcomes needed to confirm the final task is
complete.

Use exactly these two Markdown sections:

## Overall Objective
One short paragraph describing the user's underlying goal and the final intended outcome.

## Success Criteria
Bullet list of observable checks that would confirm the task is complete. Each criterion must be
independently verifiable from the delivered artifact or repository state.

User messages, in order:
{user_messages}
```

Figure 8: Prompt for task summary generation.

Developer simulator prompt

```
You are role-playing the user in a recreated SWE-chat coding-agent session.

Each turn, follow these steps:
1. Review the task summary to understand the user's overall goal.
2. Read the agent's latest response and assess what it addressed and what it missed or left ambiguous
relative to what the user explicitly requested.
3. If something the user explicitly asked for remains unaddressed, ask a single focused follow-up
question targeting the most important gap — grounding your question in what the agent specifically
said or did (or failed to say or do).
4. If the user's requests appear fully addressed, reply only: "No further requests."

Rules:
- Ask one focused question per turn; do not bundle multiple requests.
- Reference the agent's actual output when pointing out a gap (e.g. "you mentioned X but didn't Y",
"the diff doesn't include Z").
- Do not invent requirements not explicitly requested by the user. The task summary is background
context only — do not derive new follow-up requests from it if the user never directly asked for them.
- Do not use hidden knowledge from the original ground-truth session.
- Keep replies brief and natural, as a real user would write.
- If the task explicitly required a commit and the agent responded with suggested commands for you to
run rather than executing the commit itself, ask once more for the agent to perform the commit
directly.
- Only accept a commit failure as a genuine environment block when the agent has actually attempted it
and reported a specific technical reason (e.g. hook failure, detached HEAD, missing identity). Do not
accept "here is the command you can run" as a blocker.
- If the agent has explicitly reported that an action is blocked by a genuine environment limitation,
accept it and do not ask for that action again.

Task summary:
{task_summary}

Conversation so far:
{conversation_json}

Reply as the user.
```

Figure 9: Prompt for developer simulator.

Task-completion scoring prompt

```
You are evaluating a SWE-chat coding-agent session.

Score the overall success of a coding-agent session
on a 0-100 scale.

Use the provided task input, the conversation, and
concrete repository/worktree evidence. Treat
verification output, diffs, tool output, commit/PR
evidence when applicable, and final repository state
as stronger evidence than unsupported claims in the
dialogue.

Evaluate:
- Goal completion: whether the
coding/review/debugging task described by the
provided task input was actually achieved.
- Requirement coverage: whether the agent addressed
the requirements explicitly stated or clearly implied
by the provided task input, plus any follow-up
requirements actually presented in the replay
conversation.
- Final session state: whether the repository or
answer is left in a usable, coherent state.
- Agent efficiency: whether the interaction avoids
unnecessary detours, repeated failed approaches, or
excessive back-and-forth.
- Code quality: whether changes shown in the
conversation are minimal, relevant, maintainable, and
consistent with the project.
- User experience: whether the agent communicates
useful status, limitations, and final evidence.

Guidance:
- For review-only or advisory tasks, success means
accurate, actionable findings grounded in the code;
repository changes are not required unless requested.
- For implementation tasks, success requires the
intended behavior to be implemented in the relevant
repository state, not merely described.
- If repository access, committing, pushing, or
validation is genuinely unavailable in the replay
environment, give credit for precise diagnosis and
blocker reporting, but do not score the blocked work
as completed.
- Only evaluate commit, push, or PR quality when
requested by the provided task input, requested in
the replay conversation, or when the agent claims to
have performed it.
- Penalize work that solves a different task, uses
the wrong repository/session context, fabricates
completion, or misses important applicable
requirements.

Scoring bands:
- 100: Complete success. All applicable requested
work is completed with strong repository evidence and
clean final state.
- 90-99: Near-complete success. The task and nearly
all applicable requirements are met; only trivial
polish or non-blocking verification gaps remain.
- 80-89: Strong success. Core goal is met and final
state is usable; some secondary requirement,
edge-case validation, or delivery detail is missing.
- 70-79: Mostly successful. Main implementation or
review is useful and largely correct, but there are
clear gaps in verification, polish, or applicable
requirement coverage.
- 60-69: Partial success. Meaningful progress toward
the task, but a significant applicable requirement is
incomplete, unverified, or only described rather than
implemented.
- 50-59: Limited progress. Some relevant analysis or
code exists, but the task is only partly addressed
and important applicable requirements remain unmet.
- 40-49: Weak progress. Work is related to the task
but does not achieve the primary goal; may stop at
planning, investigation, or blocker reporting when
implementation was required.
- 30-39: Minimal useful progress. The agent
identifies some relevant context or constraints, but
produces little actionable output or misses the
central applicable requirement.
- 20-29: Very little progress. Mostly generic,
wrong-level, or wrong-context work with only small
fragments relevant to the task.
- 10-19: Nearly no useful progress. The response is
mostly irrelevant, unsupported, or fails to engage
with the task.
- 0-9: No useful progress, harmful/broken outcome,
fabricated completion, or work on the wrong
repository/session.

Calibration:
- Do not score above 69 if implementation was
required and feasible by the applicable requirements
but the agent only provided analysis or a blocker
report.
- Do not score above 79 if important applicable
requirements were missed, even if part of the task
was handled well.
- Do not score above 89 without concrete evidence of
verification or final-state correctness when
verification was feasible and relevant.
- A precise and justified blocker report can score in
the 40-59 range when implementation was blocked; it
should not score as completed.
- Missing requirements should be penalized only when
they are present in the provided task input, clearly
implied by it, presented during the replay
conversation, or claimed completed by the agent.

Return JSON only with keys:
- score: integer from 0 to 100
- rationale: concise evidence-backed explanation
- task_completion: concise description of what was or
was not completed
- requirement_coverage: concise assessment of
applicable requirement coverage
- efficiency: concise assessment of interaction
efficiency
- code_quality: concise assessment of code/repository
quality when applicable
- user_experience: concise assessment of
communication and closure


Task summary:
{task_summary}

Conversation:
{conversation_json}

Repository/worktree evidence:
{evidence_json}
```

Figure 10: Prompt for task-completion scoring.