RepairChemist: CROSS-LANGUAGE REPAIR SKILL DISTILLATION FOR LOW-RESOURCE PROGRAM REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have made substantial progress in program repair for high-resource programming languages (PLs) such as Java, yet a clear performance gap remains for languages with comparatively scarce repair data, such as Rust. Existing approaches use fine-tuning or program translation to alleviate this gap, but may introduce additional adaptation costs or semantic inconsistencies. In this work, we propose *RepairChemist*, a cross-language repair knowledge transfer framework that abstracts repair knowledge from high-resource PLs into reusable repair skills and guides program repair in low-resource PLs. Specifically, *RepairChemist* distills defect patterns and repair actions from bug-fix pairs in high-resource PLs, and organizes the resulting skills into skill groups based on statement types, forming a structured skill library. During low-resource program repair, *RepairChemist* retrieves applicable skills for the current defect based on its statement types and defect patterns to guide patch generation. Experiments on xCodeEval and SWE-bench Multilingual show that *RepairChemist* consistently improves repair performance across low-resource PLs and LLMs in both procedural and agentic repair settings, without parameter updates or program translation.

1 INTRODUCTION

Program repair plays an important role in software engineering. In recent years, large language models (LLMs) and agents have substantially advanced automated program repair (APR) through their strong code understanding and reasoning capabilities (Le Goues et al., 2019; Xia & Zhang, 2022; Joshi et al., 2023). Meanwhile, AI coding tools such as Claude Code (Anthropic, 2026), Codex (Chen et al., 2021), and GitHub Copilot (GitHub, 2023) are increasingly integrated into real-world software development workflows, further driving the demand for APR.

However, code LLMs exhibit substantially different repair performance across programming languages (PLs) (Hossain et al., 2024; Luo et al., 2025; 2026; Yang et al., 2026). They perform well on high-resource PLs such as Python, Java, and C++, where abundant training data are available (Xia & Zhang, 2022; 2024; Yuan et al., 2022). In contrast, their performance remains limited on low-resource PLs such as Ruby and Rust, where training data are much scarcer (Baltaji et al., 2025; Wang et al., 2023; Wong et al., 2025).

To mitigate this gap, prior work has explored cross-language knowledge transfer to improve program repair performance in low-resource PLs. For example, HELO-APR (Wang et al., 2026) constructs repair data from high-resource PLs to fine-tune models for low-resource PLs, while LANTERN (Luo et al., 2025) translates defective programs into better-supported PLs for repair. However, fine-tuning requires additional model adaptation, and program translation may introduce semantic inconsistencies.

Instead of fine-tuning or program translation, we explore directly transferring repair knowledge shared across PLs. This approach is motivated by a key observation: despite differences in code syntax, certain defect patterns and their corresponding repair actions are shared across PLs (Yuan et al., 2022; Wang et al., 2026). For example, out-of-bounds accesses can often be repaired by adding bounds checks; although implementations vary across PLs, the underlying repair knowledge remains the same. The use of skills has become increasingly widespread. A skill encapsulates reusable

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.

procedural knowledge, specifying when and how to apply it (Anthropic, 2026; Jiang et al., 2026), making it suitable for representing repair knowledge shared across similar defect. We therefore consider distilling repair knowledge into reusable repair skills. However, repair knowledge is often tightly coupled with language-specific implementations, making it difficult to distill skills reusable across PLs. Moreover, diverse defect patterns give rise to numerous skills, making it challenging to retrieve the appropriate skills for a target defect.

In this work, we propose *RepairChemist*, a repair knowledge transfer framework for low-resource program repair. Specifically, *RepairChemist* first distills language-agnostic defect patterns (when to apply) and repair actions (how to repair) from bug-fix pairs in high-resource PL, thereby decoupling repair knowledge from concrete implementations. To facilitate skill retrieval, *RepairChemist* organizes skills into groups. Since different PLs share common statement types, such as conditional and iterative statements (Roziere et al., 2020; Bui et al., 2021; Zügner et al., 2021), *RepairChemist* groups skills belonging to the same statement type into skill groups, narrowing the search space for identifying applicable skills. As more bug-fix pairs are incorporated, *RepairChemist* abstracts new skills from emerging defect patterns and refines existing skills with recurring defect patterns, enabling the continual evolution of the skill library. During the repair process for low-resource PL, *RepairChemist* first retrieves relevant skill groups from the constructed skill library, then selects applicable skills from these groups, and finally uses them to guide patch generation.

We systematically evaluate *RepairChemist* across multiple low-resource PLs and benchmarks, covering two repair scenarios: procedural repair workflows and agentic repair. The results show that *RepairChemist* consistently improves repair performance across PLs and models, demonstrating that repair skills distilled from high-resource PLs can effectively enhance low-resource program repair.

2 Related Work

2.1 LLM-Based and Agentic Program Repair

From the perspective of repair control, existing LLM-based APR methods can be broadly divided into procedural workflows and agentic repair approaches (Yang et al., 2025a; Ouyang et al., 2025). Procedural workflows embed LLMs into predefined repair pipelines, where repair steps are executed in a fixed order. For example, ChatRepair (Xia & Zhang, 2024) iteratively refines patches using test feedback, ThinkRepair (Yin et al., 2024) leverages reasoning knowledge from historical repairs, and Agentless (Xia et al., 2024) adopts a three-phase workflow of fault localization, patch generation, and validation. In contrast, agentic repair approaches use LLMs as the decision-making core to autonomously plan repair actions based on the current code state and environment feedback, while dynamically invoking tools for code retrieval, file editing, compilation, and testing. Representative methods include SWE-agent and mini-SWE-agent (Yang et al., 2024). Despite substantial progress, existing LLM-based APR research has largely focused on high-resource PLs.

2.2 Program Repair for Low-resource PLs

Program repair for low-resource PLs is mainly limited by the scarcity of high-quality bug-fix data (Zhang et al., 2023; Gharibi et al., 2026; Luo et al., 2025; Gharibi et al., 2023). Existing work addresses this issue through fine-tuning, data synthesis, and program translation. Baltaji et al. (2025) fine-tune code LLMs on high-resource bug-fix data and transfer them to low-resource PLs, while Wang et al. (2026) construct low-resource repair data through translation and bug injection, followed by model training. These training-based methods require additional training and adaptation for new target languages. Wong et al. (2025) synthesize low-resource bug-fix pairs with LLMs, but synthetic bugs may differ from real-world defects. Luo et al. (2025) uses a translate–repair–translate-back pipeline, which may introduce semantic inconsistencies or lose language-specific features. In contrast to these approaches, *RepairChemist* distills transferable repair skills from high-resource PLs to directly guide program repair in low-resource PLs.

2.3 Repair Knowledge Reuse

Prior studies have explored how historical repairs within the same PL can provide reusable repair knowledge. Getafix (Bader et al., 2019) and FixMiner (Koyuncu et al., 2020) mine reusable code edit

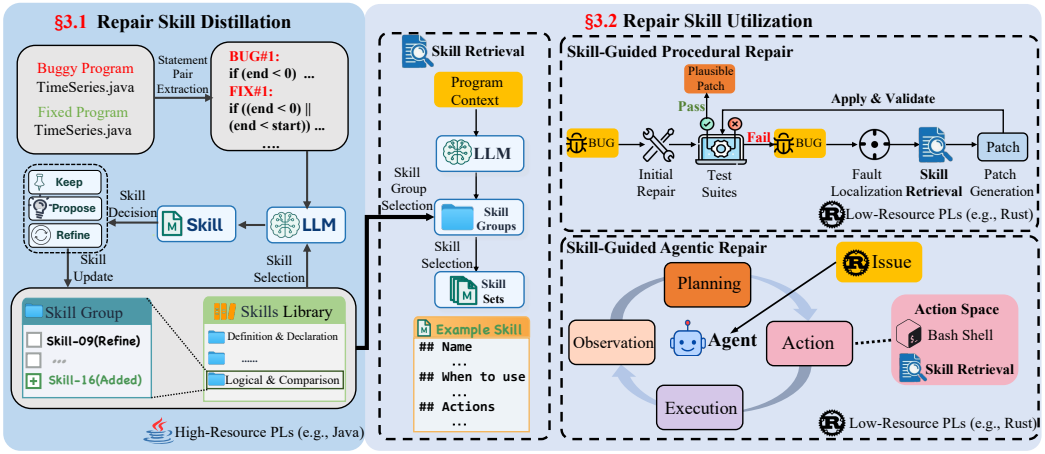

Figure 1: Overview of *RepairChemist*.

patterns from historical patches to guide candidate patch generation. ThinkRepair (Yin et al., 2024) collects successful repair examples and their reasoning processes, then retrieves similar repair cases to guide program repair. ExpeRepair (Mu et al., 2026) stores repair examples and distills general repair insights from repair trajectories to guide program repair. However, these insights only abstract general repair knowledge without clarifying when to reuse it. These methods primarily focus on reusing historical repair knowledge within the same PL, without explicitly addressing knowledge abstraction and retrieval for cross-language transfer. In contrast to merely abstracting repair insights, *RepairChemist* explicitly decomposes repair knowledge from historical patches into defect patterns (when to apply) and repair actions (how to repair), enabling precise reuse of repair knowledge.

3 METHODOLOGY

As shown in Figure 1, *RepairChemist* comprises repair skill distillation, which constructs a structured skill library from high-resource PL, and repair skill utilization, which retrieves relevant skills to guide patch generation in low-resource PLs.

3.1 DISTILLING CROSS-LANGUAGE REPAIR SKILLS

Given a source dataset $\mathcal{D}_{src}$ of bug-fix pairs from a high-resource PL (Java in our experiments), where B_i and F_i denote the buggy and fixed programs, respectively, repair skill distillation aims to abstract reusable cross-language repair skills from these pairs. Compared with concrete code edits, the underlying defect patterns and repair actions exhibit stronger commonality across PLs. Therefore, a repair skill should capture defect patterns and repair actions while also abstracting away language-specific implementation details. Following the skill representation in (AgentSkills, 2025), we represent each repair skill as $s = (T_s, A_s)$, where T_s denotes the defect pattern specifying when the skill should be applied and A_s denotes a set of repair actions specifying how to repair the defect. A complete repair skill example is provided in Appendix B.2.

However, distilling reusable cross-language repair skills from raw patches presents two challenges. ❶ Concrete code patches contain language-specific implementation details and often mix multiple statement-level modifications associated with different defect patterns and repair actions, making it difficult to isolate repair knowledge that generalizes across PLs. ❷ Different bug-fix pairs may share the same defect pattern, but treating them independently can result in redundant skills. Even after consolidation, a growing skill library makes it challenging to retrieve relevant skills. To address these challenges, *RepairChemist* extracts statement-level bug-fix pairs and distills language-agnostic defect patterns and repair actions. It then consolidates statement-level bug-fix pairs sharing the same defect pattern into a single skill and organizes the resulting skills by statement type to facilitate retrieval.

Step 1: Statement-Level bug-fix Pair Extraction. To mitigate the ambiguity introduced by patch-level modifications, extracting repair knowledge requires an appropriate code granularity that balances locality and semantic completeness. Compared with lines, which lack stable syntactic boundaries, and functions, which may contain multiple distinct modifications, statements provide clear syntactic boundaries and a more localized scope, making them suitable units for extracting repair knowledge. Moreover, different PLs share fundamental statement types, providing a unified basis for facilitating cross-language retrieval.

Specifically, for each (B_i, F_i), *RepairChemist* first uses GumTree (Falleri et al., 2014) to compare the abstract syntax trees before and after repair, identify code edit operations, and merge edits belonging to the same statement into a statement-level bug-fix pair. *RepairChemist* then provides each statement-level bug-fix pair, together with the complete buggy and fixed programs, to an LLM, which assigns it to one of 15 predefined statement types based on the corresponding statement type. *RepairChemist* organizes statement-level bug-fix pairs by statement type to support subsequent skill distillation and cross-language retrieval. These statement types are defined with reference to classical classifications of program structures (Scott, 2000; Aho et al., 2007), with the complete list provided in Appendix B.1.

Step 2: Skill Library Consolidation and Evolution. After extracting statement-level bug-fix pairs, *RepairChemist* distills language-agnostic defect patterns and repair actions from each pair. To avoid redundant skills, *RepairChemist* consolidates bug-fix pairs that share a defect pattern into a single skill, allowing multiple pairs to contribute different repair actions. For each bug-fix pair, *RepairChemist* first identifies its defect pattern and updates the skill library based on its match with existing skills. It then applies the same mechanism to distill the repair action and update the corresponding skill's action set based on its consistency with existing actions.

Specifically, for each statement-level bug-fix pair, *RepairChemist* provides the pair, the complete buggy program, and the defect patterns of existing skills in the corresponding skill group to an LLM. The LLM identifies the underlying defect mechanism and applicable conditions while removing language-specific implementation details and performs one of three operations: ❶ **Keep**: if the defect pattern of an existing skill accurately captures the defect mechanism of the current pair, its T_s is retained; ❷ **Propose**: if no existing skill describes the defect mechanism of the current pair, the LLM generates a new defect pattern T_s to create a new skill; ❸ **Refine**: if the current pair matches an existing skill but its T_s is inaccurate, the LLM generates a candidate description and adds it to the corresponding refinement buffer. Once the buffer accumulates K candidate descriptions, the LLM updates the defect pattern based on the existing T_s and these candidates, enabling refinement based on multiple observations rather than isolated pairs.

After determining the corresponding skill, *RepairChemist* abstracts the repair action from the current statement-level bug-fix pair. Specifically, *RepairChemist* provides the pair, the complete buggy program, and the skill's existing repair actions to the LLM, which identifies the underlying code transformation required for the repair while removing language-specific implementation details. The LLM then applies the same **Keep**, **Propose**, and **Refine** mechanism to reuse an existing action, add a new repair action, or refine an existing action after K candidate descriptions have accumulated. In this way, multiple bug-fix pairs sharing the same defect pattern can contribute different repair actions to the same skill.

Through this process, *RepairChemist* constructs a repair skill library organized by statement types, where each skill group contains specialized repair skills distilled and evolved from statement-level bug-fix pairs. In addition, each skill group contains a predefined fallback skill that directly generates patches from the target code and context when no specialized skill applies.

3.2 Repair Skill Utilization

RepairChemist retrieves relevant skills from the distilled skill library and integrates them into procedural repair workflows and repair agents for low-resource program repair.

Skill Retrieval. The Skill Retrieval process consists of two steps: Skill Group Selection and Skill Selection. Given the localized code region (which may contain multiple faulty statements) and its context, the LLM first identifies the statement types associated with the defect and retrieves the

corresponding skill groups. It then selects an applicable skill from each retrieved group by matching its defect pattern T_s to the current defect. The selected skills collectively form a skill set, which guides the generation of a candidate patch.

Skill-Guided Procedural Repair. The LLM first directly generates a candidate patch and performs validation. Following LANTERN (Luo et al., 2025), candidate patches are validated against the benchmark test suites. If validation fails, the LLM performs fault localization on the original buggy program to identify the faulty code region and retrieves a skill set based on the localized region and its context. The retrieved skill set then guides the generation of a single candidate patch, followed by validation. If validation fails again, the workflow repeats fault localization, skill retrieval, and patch generation on the original buggy program until a candidate passes validation or the maximum of M repair rounds, including the initial direct repair, is reached. To stabilize skill-guided repair, fault localization, skill group selection, and skill selection each employ v parallel samples, with identical results counted as votes to determine the final localized region and skill set by majority voting.

Skill-Guided Agentic Repair. *RepairChemist* exposes a callable skill retrieval interface to the repair agent. Based on the issue description, repository state, and execution feedback, the agent autonomously decides when to inspect code, make edits, and run tests. Unlike procedural repair, the agent can evaluate and select among multiple candidate patches; therefore, *RepairChemist* retains multiple skill retrieval results without majority voting. Specifically, when generating candidate patches, the agent samples m skill-group sets and n skill sets per group set through the retrieval interface, yielding $k = m \times n$ skill sets and k corresponding candidate patches. The agent then applies and validates candidate patches, using execution feedback to decide whether to continue repair or submit a final patch, subject to a maximum of L agent steps.

4 Experiments

4.1 Experiment Setup

Data for Repair Skill Distillation. XCodeEval (Khan et al., 2024) is a large-scale multilingual benchmark containing multiple bug-fix pairs per programming problem. We use Java, a widely studied PL in APR, as the source PL for skill distillation and select a subset of problems from the Java training split. For each selected problem, we retain at most four bug-fix pairs, yielding 1,039 bug-fix pairs for skill distillation. We further verify that the source data has no problem-level overlap with the evaluation sets (see Appendix A.3).

Evaluation Benchmarks. We evaluate procedural repair and agentic repair using xCodeEval and SWE-bench Multilingual (Yang et al., 2025b), respectively. For procedural repair, we evaluate on the xCodeEval validation set, using its Kotlin, Ruby, and Rust subsets with 313, 343, and 205 repair tasks, respectively. SWE-bench Multilingual contains real-world repository-level repair tasks across multiple PLs. We use its Rust, Ruby, and Go subsets, comprising 43, 44, and 42 repair tasks, respectively.

Baselines. For procedural repair, we construct *Direct Repair* as the basic baseline and compare it with three categories of methods. *Repair workflow enhancement* methods include ChatRepair (Xia & Zhang, 2024), Self-Planning (Jiang et al., 2024), and Self-Collaboration (Dong et al., 2024), which improve repair through test feedback, explicit planning, and multi-role collaboration, respectively. *Repair knowledge reuse* methods include ThinkRepair (Yin et al., 2024) and ExpeRepair (Mu et al., 2026). The former primarily relies on historical repair examples, while the latter additionally incorporates general repair insights to guide patch generation. *Cross-language repair* is represented by LANTERN (Luo et al., 2025), which generates patches through translation, repair, and back-translation. To ensure a fair comparison, all procedural repair methods, including Direct Repair, use the same benchmark-test-suite validation protocol and are allowed up to three repair rounds. For agentic repair, we use *mini-SWE-agent* (Yang et al., 2024) as the baseline and limit both mini-SWE-agent and *RepairChemist* to $L = 250$ steps.

Models. We use GPT-5.5 to distill repair skills with medium reasoning effort. For evaluation, we consider open-source models of different sizes, including Qwen3-Coder-30B-A3B-Instruct (referred

Table 1: Comparison of repair methods using Qwen3-30B on xCodeEval, reporting resolved rates (%) and average token consumption. The best and second-best results are bolded and underlined, respectively. Green arrows indicate absolute improvements over Direct Repair.

Method	Rust	Ruby	Kotlin	Avg.	Avg. Tokens
Direct Repair	65.37	58.31	61.66	61.78	1,525.69
ChatRepair	70.73 ↑ 5.36	65.01 ↑ 6.70	70.93 ↑ 9.27	68.89 ↑ 7.11	7,939.06
Self-Planning	70.24 ↑ 4.87	65.31 ↑ 7.00	71.57 ↑ 9.91	69.04 ↑ 7.26	2,230.25
Self-Collaboration	69.76 ↑ 4.39	66.18 ↑ 7.87	68.37 ↑ 6.71	68.10 ↑ 6.32	5,332.29
ThinkRepair	73.17 ↑ 7.80	67.64 ↑ 9.33	70.29 ↑ 8.63	70.37 ↑ 8.59	9,655.21
ExpeRepair	72.68 ↑ 7.31	69.97 ↑ 11.66	72.20 ↑ 10.54	71.62 ↑ 9.84	4,397.19
LANTERN	73.17 ↑ 7.80	71.14 ↑ 12.83	71.25 ↑ 9.59	71.85 ↑ 10.07	4,593.74
RepairChemist	**76.10** ↑ **10.73**	**75.80** ↑ **17.49**	**77.32** ↑ **15.66**	**76.41** ↑ **14.63**	5,253.29

to as Qwen3-30B) (Team, 2025), Qwen3-Coder-Next (80B) (Cao et al., 2026), and DeepSeek-V4-Flash (Preview, 284B) (Xu et al., 2026), as well as the proprietary models Claude-Sonnet-4.5 and GPT-5.6-Luna. GPT-5.6-Luna uses medium reasoning effort; all other evaluation models operate in non-thinking mode. We serve Qwen3-30B on a single NVIDIA A100 GPU using SGLang (Zheng et al., 2024) and access all remaining models through their official APIs.

Metrics. We evaluate all methods in terms of repair effectiveness and token consumption. For repair effectiveness, we use the resolved rate (Jimenez et al., 2024), defined as the percentage of tasks for which a generated patch passes all benchmark-required tests. For token consumption, we report the average total token consumption per task, calculated by summing input and output tokens across all LLM calls during the repair process.

Implementation Details. We run evaluations in benchmark-specific Docker environments and set top-p to 0.95 throughout. During skill distillation, we set $K = 6$, such that a skill is refined after six candidate descriptions are accumulated in its refinement buffer. For xCodeEval, we set the maximum number of repair rounds to $M = 3$ and generate one candidate patch per round. We set $v = 5$ for fault localization, skill-group selection, and skill selection, applying parallel sampling and majority voting at each stage with temperature 0.2. For SWE-bench Multilingual, we set $n = m = 2$ to retrieve $k = 4$ skill sets without voting, generating one candidate patch for each skill set. We use temperature 0 for patch generation on both benchmarks. Full prompts and additional implementation details appear in Appendix C.

4.2 Comparison with Existing Repair Methods

Table 1 compares *RepairChemist* with different repair methods on xCodeEval. *RepairChemist* achieves the highest resolved rates across all three low-resource PLs, averaging 76.41. Compared with ChatRepair, Self-Planning, and Self-Collaboration, *RepairChemist* achieves absolute gains of 7.37–8.31 (10.7–12.2% relative), highlighting the benefits of cross-language repair knowledge transfer. It outperforms ThinkRepair and ExpeRepair by 6.04 and 4.79 (8.6% and 6.7% relative), respectively, suggesting that organizing repair knowledge into defect-specific skills and retrieving relevant skills enables more effective knowledge reuse than relying on historical repair examples or general repair insights. Compared with LANTERN, *RepairChemist* achieves an absolute gain of 4.56 (6.3% relative), suggesting that abstracting repair knowledge into reusable skills provides an effective alternative to program translation for cross-language transfer.

Across different PLs, *RepairChemist* achieves the largest improvement on Ruby, with an absolute gain of 17.49 (30.0% relative). Notably, the resolved rate gap (highest minus lowest) across the three PLs narrows from 7.06 under Direct Repair to 1.52 with *RepairChemist*, suggesting that transferable repair skills can help reduce performance disparities across the evaluated PLs. Moreover, although Kotlin performs worse than Rust under Direct Repair, it surpasses Rust with *RepairChemist*. One possible explanation is that Kotlin's similarity to Java, the source language for skill distillation, facilitates repair knowledge transfer.

We further compare the token consumption of different methods. *RepairChemist* uses an average of 5,253.29 tokens per task, fewer than ChatRepair (7,939.06) and ThinkRepair (9,655.21), and

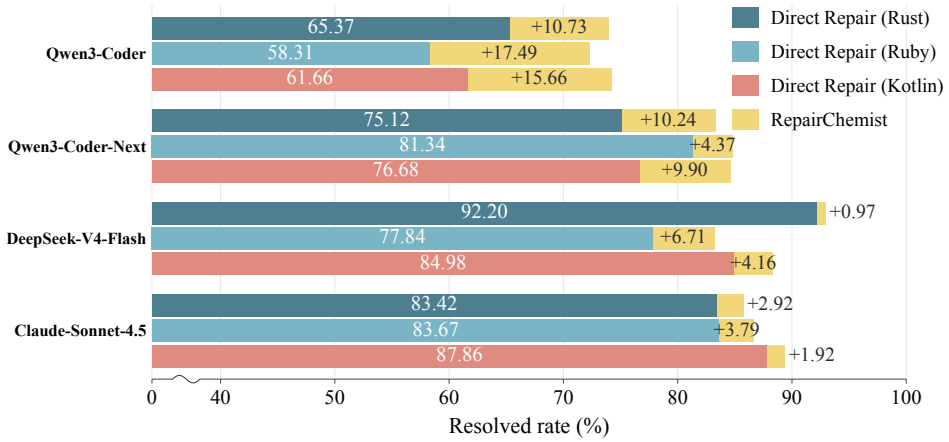

Figure 2: Resolved rates (%) across LLMs on xCodeEval.

comparable to Self-Collaboration (5,332.29). Compared with the strongest baseline, LANTERN, *RepairChemist* requires only 659.55 additional tokens (14.4% relative) while achieving an absolute gain of 4.56 (6.3% relative). Overall, *RepairChemist* achieves higher repair performance than all evaluated baselines while maintaining a moderate level of token consumption.

We assess statistical significance using the McNemar test (McNemar, 1947). *RepairChemist* significantly outperforms all baselines on xCodeEval using Qwen3-30B ($p < 0.05$; see Appendix A.1).

4.3 Effectiveness Across Different LLMs

As shown in Figure 2, *RepairChemist* consistently improves repair performance across all LLMs and low-resource PLs. For Qwen3-30B, whose direct repair capability is relatively limited, *RepairChemist* achieves absolute gains of 10.73, 17.49, and 15.66 on Rust, Ruby, and Kotlin, respectively, with relative gains of 16.4%, 30.0%, and 25.4%. On Qwen3-Coder-Next, *RepairChemist* also achieves absolute gains of 10.24, 4.37, and 9.90, respectively, with relative gains of 13.6%, 5.4%, and 12.9%. As the intrinsic repair capability of the LLMs increases, the overall gains from *RepairChemist* tend to decrease. Nevertheless, DeepSeek-V4-Flash and Claude-Sonnet-4.5 still achieve average absolute gains of 3.95 and 2.88, with relative gains of 4.6% and 3.4%, respectively. Notably, even for stronger LLMs, the distilled repair skills can help mitigate weaknesses on specific low-resource PLs. For example, *RepairChemist* achieves an absolute gain of 6.71 (8.6% relative) on Ruby with DeepSeek-V4-Flash, where the baseline performance is relatively low.

4.4 Effectiveness in Repository-Level Agentic Repair

As shown in Figure 3, *RepairChemist* consistently improves repository-level agentic repair performance across all LLMs and low-resource PLs. The gains are particularly pronounced for Qwen3-30B: *RepairChemist* achieves absolute gains of 16.28, 11.36, and 9.52 on Rust, Ruby, and Go, respectively, with relative gains of 87.5%, 71.4%, and 57.1%, suggesting that the distilled repair skills effectively complement its limited repository-level repair knowledge in low-resource PLs. On stronger LLMs, the gains tend to decrease, with average absolute improvements of 5.46, 6.22, and 4.67 on Qwen3-Coder-Next, DeepSeek-V4-Flash, and GPT-5.6-Luna, respectively, with relative gains of 8.5%, 9.4%, and 7.2%. Despite these smaller gains, distilled repair skills can still complement the capabilities of stronger LLMs, particularly on low-resource PLs where their baseline repair performance is relatively weak. For example, *RepairChemist* achieves an absolute gain of 7.14 (14.3% relative) on Go with DeepSeek-V4-Flash. Overall, these results suggest that repository-level agentic repair can benefit from reusable repair knowledge, even when distilled from Java bug-fix pairs in competitive programming.

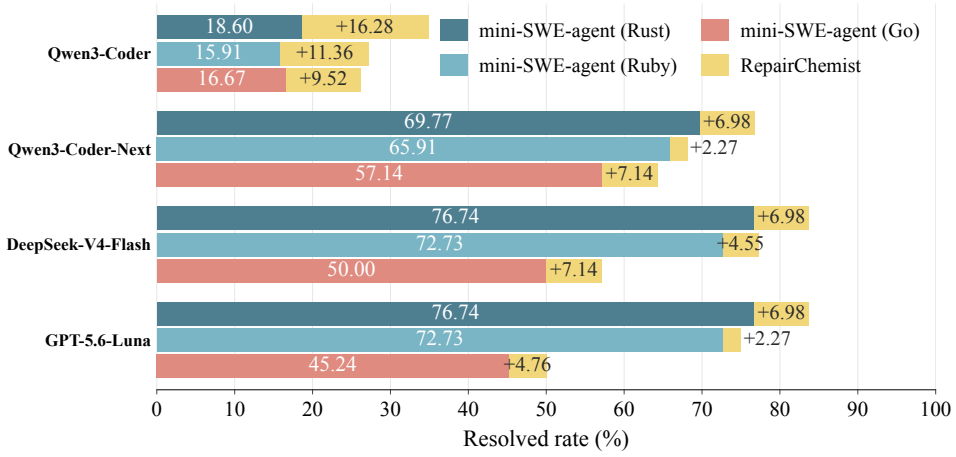

Figure 3: Resolved rates (%) across LLMs on SWE-bench Multilingual.

Table 2: Ablation results on xCodeEval using Qwen3-30B (%).

Method	Rust	Ruby	Kotlin	Avg.
RepairChemist (Full)	**76.10**	**75.80**	**77.32**	**76.41**
w/o Skill Library	68.78	71.43	70.28	70.16
w/o Statement-Type Grouping	73.66	72.01	72.52	72.73
w/o Majority Voting	74.15	74.34	76.04	74.84

Table 3: Ablation results on SWE-bench Multilingual using DeepSeek-V4-Flash (%).

Method	Rust	Ruby	Go	Avg.
RepairChemist (Full)	**83.72**	**77.27**	**57.14**	**72.71**
w/o Skill Library	76.74	70.45	47.62	64.94
w/o Statement-Type Grouping	79.07	70.45	50.00	66.51
w/o Multiple Candidates	81.40	72.73	54.76	69.63

4.5 Ablation Study

We conduct ablation studies using Qwen3-30B on xCodeEval and DeepSeek-V4-Flash on SWE-bench Multilingual, as Qwen3-30B has a relatively low baseline on the latter. For **w/o Skill Library**, we remove the skill library and skill retrieval. On xCodeEval, we retain fault localization and its majority voting, but generate patches directly after localization; on SWE-bench Multilingual, we keep the original agent workflow and let the model directly generate four candidate patches. For **w/o Statement-Type Grouping**, we remove Statement-Type Grouping and retrieve applicable skills directly from the entire skill library, while keeping all other settings unchanged. Since xCodeEval uses majority voting, whereas SWE-bench Multilingual uses multiple candidate patches, we design benchmark-specific ablations. On xCodeEval, **w/o Majority Voting** replaces majority voting in fault localization and skill retrieval with a single selection. On SWE-bench Multilingual, **w/o Multiple Candidates** retrieves only one skill set and generates a single candidate patch.

As shown in Table 2, for procedural repair on xCodeEval, removing the skill library causes the largest performance drop (-6.25), indicating that distilled repair skills provide effective external repair knowledge for improving LLM-based repair in low-resource PLs. Removing Statement-Type Grouping also reduces the resolved rate by 3.68 on average, indicating that the gains depend not only on the skill content itself, but also on how effectively relevant skills are retrieved. Removing majority voting consistently reduces performance, resulting in an average decrease of 1.57, indicating that majority voting reduces uncertainty in multi-stage decision-making and mitigates the impact of individual incorrect selections.

As shown in Table 3, for repository-level agentic repair on SWE-bench Multilingual, removing the skill library causes the largest performance drop, reducing the resolved rate by 7.77, which confirms that distilled repair skills provide effective support for repository-level repair. After removing Statement-Type Grouping, the average resolved rate drops by 6.20, a decrease comparable to that caused by removing the skill library. This suggests that, in repository-level repair, effective skill retrieval is nearly as important as repair knowledge itself. This may be because inferring underlying defect patterns from repository-level issues is challenging, making effective skill organization important for reducing ambiguity in skill retrieval. In addition, w/o Multiple Candidates reduces the average resolved rate by 3.08. This suggests that multiple candidates provide the agent with diverse repair options for handling repository-level defects.

Table 4: Fault localization and repair performance (%) of *RepairChemist* and w/ Execution Feedback using Qwen3-30B on xCodeEval.

(a) *RepairChemist*

Language	Localization	Resolved
Rust	78.05	76.10
Ruby	88.63	75.80
Kotlin	78.59	77.32

(b) *RepairChemist* w/ Feedback

Language	Localization	Resolved
Rust	83.90	77.56
Ruby	89.21	78.72
Kotlin	90.74	83.07

4.6 Discussion

Here, we analyze potential repair bottlenecks and discuss how execution feedback and skill library expansion can further improve *RepairChemist*.

Analysis of Repair Bottlenecks. Accurate fault localization is a key prerequisite for generating correct patches (Xu et al., 2020; Meng et al., 2022). To analyze the repair bottlenecks of *RepairChemist*, we examine how fault-localization accuracy relates to the final resolved rate. Since xCodeEval does not provide ground-truth patches, We construct reference patches from fixes in other subsets of xCodeEval (see Appendix A.2) and evaluate localization at the line level following prior work (Ouyang et al., 2025; Xia et al., 2024). An instance is considered successfully localized if the identified code region covers all modified lines in the reference patch. As shown in Table 4(a), under the original setting without execution feedback, the localization accuracy exceeds the resolved rate by only 1.95 and 1.27 for Rust and Kotlin, respectively, suggesting that fault localization may be an important limiting factor for these two PLs. In contrast, Ruby achieves 88.63 localization accuracy but only a 75.80 resolved rate, suggesting that patch generation may be a bottleneck for Ruby.

Incorporating Execution Feedback. To alleviate these bottlenecks, we incorporate execution feedback into the *RepairChemist* repair workflow. The generated patches and validation feedback from the previous round are incorporated into subsequent fault localization, skill retrieval, and patch generation. As shown in Table 4(b), execution feedback improves fault localization accuracy across all three PLs. Kotlin achieves the largest absolute gain in localization accuracy (+12.15), along with the largest improvement in resolved rate (+5.75). For Rust, although localization accuracy improves by 5.85, the resolved rate increases by only 1.46, suggesting that improvements in fault localization alone are insufficient to substantially improve repair performance. In contrast, Ruby achieves a localization gain of only 0.58 but an improvement of 2.92 in resolved rate, suggesting that execution feedback provides greater benefits in later repair stages.

Further Expanding the Skill Library. Future work can further expand the *RepairChemist* skill library along two dimensions: target PLs and repository-level data sources. First, the current skills mainly capture repair knowledge shared across PLs. Future work can further distill language-specific skills from bug-fix pairs in target PLs, such as Rust-specific skills from Rust bug-fix pairs for ownership-related defects. Second, the current skills are primarily distilled from competitive programming datasets. Although our repository-level experiments demonstrate their transferability to real-world repositories, future work can incorporate patches from such repositories to capture richer repository-level repair knowledge.

5 Conclusion

In this work, we propose *RepairChemist*, a cross-language repair knowledge transfer framework for low-resource program repair. *RepairChemist* distills defect patterns (when to apply) and repair actions (how to repair) from high-resource PLs into reusable repair skills for cross-language transfer, and organizes them into skill groups to facilitate the retrieval of applicable skills. Extensive experiments demonstrate that *RepairChemist* consistently improves repair performance across different low-resource PLs and LLMs in both procedural and repository-level agentic repair settings. These results demonstrate that repair knowledge can be explicitly abstracted from language-specific implementations and reused across PLs, providing a knowledge transfer pathway for low-resource program repair without model adaptation or program translation.

AI USE STATEMENT

Large language models were used to assist with language polishing and code implementation. All technical content, code, experimental results, and citations were independently reviewed and verified by the authors.

ETHICS STATEMENT

This work uses only publicly available program repair benchmarks and does not involve human subjects or personal data. Automatically generated patches may contain errors or be misused; therefore, they should be validated with project-specific tests and further reviewed by developers for correctness and safety before deployment.

REPRODUCIBILITY STATEMENT

We provide an anonymous code repository (`https://anonymous.4open.science/r/repairchemist-47E5`) containing the benchmark configurations, model and prompt settings, a snapshot of the skill library, and per-instance results to support reproducibility.

REFERENCES

AgentSkills. Agentskills. GitHub repository, 2025. URL `https://github.com/agentskills/agentskills`.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Addison-Wesley, 2007.

Anthropic. Claude code, 2026. URL `https://www.anthropic.com/claude-code`. Accessed: 2026-09-02.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

Razan Baltaji, Saurabh Pujar, Martin Hirzel, Louis Mandel, Luca Buratti, and Lav R Varshney. Cross-lingual transfer in programming languages: An extensive empirical study. *Transactions on Machine Learning Research*, 2025.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1186–1197. IEEE, 2021.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report, February 2026.

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.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 313–324, 2014. doi: 10.1145/2642937.2642982. URL `http://doi.acm.org/10.1145/2642937.2642982`.

Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. T5apr: Empowering automated program repair across languages through checkpoint ensemble. *arXiv preprint arXiv:2309.15742*, 2023.

Reza Gharibi, Mohammad Hadi Sadreddini, and Seyed Mostafa Fakhrahmad. Multimend: multilingual program repair with context augmentation and multi-hunk patch generation. *Automated Software Engineering*, 33(2):69, 2026.

GitHub. Github copilot · your ai pair programmer. `https://github.com/features/copilot/.`, 2023.

Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering*, 1(FSE):1471–1493, 2024.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

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, pp. 54107–54157, 2024.

Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 5131–5140, 2023.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6766–6805, 2024.

Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, 2020.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Wenqiang Luo, Jacky Wai Keung, Boyang Yang, Jacques Klein, Tegawende F Bissyande, Haoye Tian, and Bach Le. Unlocking llm repair capabilities through cross-language translation and multi-agent refinement. *arXiv preprint arXiv:2503.22512*, 2025.

Wenqiang Luo, Jacky Wai Keung, Boyang Yang, He Ye, Claire Le Goues, Tegawende F Bissyande, Haoye Tian, and Bach Le. When fine-tuning llms meets data privacy: An empirical study of federated learning in llm-based program repair. *ACM Transactions on Software Engineering and Methodology*, 35(3):1–46, 2026.

Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pp. 1169–1180, 2022.

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. Experepair: Dual-memory enhanced llm-based repository-level program repair. *Proceedings of the ACM on Software Engineering*, 3(FSE):3954–3976, 2026.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. In *International Conference on Learning Representations*, volume 2025, pp. 30098–30121, 2025.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33: 20601–20611, 2020.

Michael Lee Scott. *Programming language pragmatics*. Morgan Kaufmann, 2000.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Weishi Wang, Yue Wang, Steven Hoi, and Shafiq Joty. Towards low-resource automatic program repair with meta-learning and pretrained language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 6954–6968, 2023.

Zhipeng Wang, Boyang Yang, Yidong Wan, Liuye Guo, You Lv, Tao Zheng, Zhuowei Wang, and Tieke He. Helo-apr: Enhancing low-resource program repair through cross-lingual knowledge transfer. *arXiv preprint arXiv:2604.17016*, 2026.

Kyle Wong, Alfonso Amayuelas, Liangming Pan, and William Yang Wang. Investigating the transferability of code repair for low-resource programming languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3410–3432, 2025.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, 2022.

Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 819–831, 2024.

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

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo A Furia. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering*, 48(1):309–326, 2020.

Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F Bissyandé, Yang Liu, and Haoye Tian. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*, 2025a.

Boyang Yang, Haoye Tian, Jiadong Ren, Hongyu Zhang, Jacques Klein, Tegawendé F Bissyandé, Claire Le Goues, and Shunfu Jin. Morepair: Teaching llms to repair code via multi-objective fine-tuning. *ACM Transactions on Software Engineering and Methodology*, 35(2):1–38, 2026.

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

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

Xin Yin, Chao Ni, Shaohua Wang, Zhenhao Li, Limin Zeng, and Xiaohu Yang. Thinkrepair: Self-directed automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1274–1286, 2024.

Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. Circle: Continual repair across programming languages. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 678–690, 2022.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37: 62557–62583, 2024.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. *arXiv preprint arXiv:2103.11318*, 2021.

A Additional Experimental Analysis

A.1 Statistical Significance Analysis

We use the McNemar test to assess the statistical significance of *RepairChemist*'s performance improvements. On xCodeEval, *RepairChemist* significantly outperforms all baselines using Qwen3-30B ($p < 0.05$). On SWE-bench Multilingual, it also significantly outperforms mini-SWE-agent ($p < 0.05$).

A.2 Reference Patch Construction

Since the official xCodeEval validation set does not provide ground-truth patches, we construct reference patches for localization analysis. We collect 820 corrected versions from other xCodeEval tasks (e.g., code translation and tag classification) and generate the remaining 41 using GPT-5.6-Sol. All 861 reference patches pass the hidden tests. We compute code differences between buggy and corrected versions and treat the modified lines as reference repair regions. A localization is successful only when the identified code region covers all modified lines. These reference patches are used only for localization analysis and not for evaluating repair outcomes.

A.3 Problem Overlap Analysis

Since xCodeEval is constructed from Codeforces problems, the same problem may have implementations in different PLs. Although XCodeEval performs problem-level splitting to avoid overlap between its training and evaluation sets, cross-language skill distillation may still introduce potential problem overlap between the source language and target evaluation languages. Therefore, we further verify the problem-level separation between the Java data used for skill distillation and the Kotlin, Ruby, and Rust evaluation sets.

Specifically, we construct the skill distillation data from the Java training shard `train_000`, which contains 4,998 Java bug-fix pairs from 289 unique problems identified by `src_uid`. After retaining at most four bug-fix pairs per problem, we obtain 1,039 bug-fix pairs. We compare the `src_uid` values of these 289 Java problems with those in the Kotlin, Ruby, and Rust evaluation sets. The comparison shows that no overlapping problems exist between the source data and the evaluation sets.

B Skill Library Details

B.1 statement types and Skill Distribution

The skill library organizes repair skills into 15 statement types according to the semantic functions of their associated statements. Skills within the same category are grouped together to support cross-language skill retrieval. As shown in Table 5, *RepairChemist* ultimately distills 191 repair skills, which are distributed across 15 statement types.

Table 5: Distribution of repair skills distilled by GPT-5.5 from Java bug-fix pairs in xCodeEval.

Statement Type	**Count**	**Statement Type**	**Count**
Definition and Declaration Statements	10	Assignment Statements	19
Selection and Conditional Statements	15	Iterative Statements	17
Return and Jump Control Statements	9	Exception-Related Statements	4
Context Control Statements	5	Assertion Statements	4
Import, Export, and Name Visibility Statements	2	Statements with Access or Reference Expressions	9
Statements with Arithmetic or Bitwise Expressions	31	Statements with Logical or Comparison Expressions	19
Statements with Constant Values	4	Statements with Cast or Type Expressions	8
Statements with Call Expressions	35	**Total**	**191**

B.2 Repair Skill Example

The following presents an example of a distilled repair skill. During patch generation, the skill's repair actions are provided to the LLM, which generates a patch based on the provided actions and program context.

Distilled Repair Skill

```
## name
predicate_basis_error
## When to use
The boolean condition contains incorrect or incomplete predicates,
    causing objects to be incorrectly filtered, retained, or
    classified.
## Actions
1. Replace the incorrect predicate
2. Add the missing predicate
3. Remove the irrelevant predicate
```

C Prompts

We provide the complete prompts used in *RepairChemist*, covering repair skill distillation, skill retrieval, and skill utilization in both procedural and agentic repair settings.

C.1 Skill Distillation

Skill Group Selection

```
For each statement diff, select exactly one existing statement type (primary_intent).
     Use the buggy source only as context, classify each diff independently, choose the
      smallest changed statement, do not invent category names, return one result per
     statement_index, and output JSON only.
Language: {language};
Category catalog: {catalog}
Buggy source: {buggy_source}
Fixed source: {fixed_source}
Statement diffs: {statement_diffs}
Output:
{
  "results": [
    {
      "statement_index": 0,
      "primary_intent": "exact_category_name"
    }
  ]
}
```

Skill Defect-Pattern Decision

```
Analyze the given statement diff and determine its repair skill.


First, summarize the statement's defect pattern from the buggy statement and diff.
     Describe the underlying defect mechanism in language-agnostic terms, avoiding
     source-language-specific syntax and implementation details while preserving the
     conditions under which the defect occurs. Apply this requirement to both newly
     proposed and refined skill descriptions. Then compare this defect pattern with the
      defect patterns of the candidate repair skills.


propose = no candidate repair skill applies; create a new repair skill
```

```
refine = a candidate generally applies, but its name, description, or example needs
     refinement
keep = a candidate fully applies; reuse it unchanged

For refine or keep, return the exact candidate skill name. For propose, create a
     lowercase snake_case skill name with a concise description and Return JSON only.

Primary intent: {primary_intent}
Intent description: {intent_description}
Statement index: {statement_index}
Buggy statement: {buggy_statement}
Diff: {diff}
Candidate repair skills and applicable conditions: {candidates}

Output:
{
  "defect_pattern": "concise description of the statement's defect pattern",
  "decision": "propose|refine|keep",
  "skill": "exact_candidate_name_or_new_skill",
  "description": "concise description_or_empty",
  "reason": "concise justification"
}
```

Skill Action Decision

```
Analyze the given statement diff and determine the repair action under the selected
     repair skill.

First, summarize the required code edit from the buggy statement and diff. Then compare
      this edit with the candidate repair actions. Select exactly one option:

propose = no candidate action matches the required edit; create a new action
refine = a candidate describes the same repair operation but needs clearer wording
keep = a candidate fully describes the required edit; reuse it unchanged

For refine or keep, return the exact candidate action name. For propose, create a
     concise, concrete, language-agnostic action phrase. Refinement must preserve the
     same repair operation. Return JSON only.

Primary intent: {primary_intent}
Sample ID: {sample_id}
Statement index: {statement_index}
Selected repair skill: {selected_skill}
Buggy statement: {buggy_statement}
Diff: {diff}
Candidate repair actions: {candidates}

Output:
{
  "repair_action": "concise description of the required code edit",
  "decision": "propose|refine|keep",
  "action": "exact_candidate_name_or_new_action",
  "reason": "concise justification"
}
```

C.2 SKILL RETRIEVAL

Skill Group Selection

```
Select one or more distinct statement types for every supplied location. Use each
     category's intent and the statement's role. Do not rely on superficial keyword
     matching. For a multi-statement region, choose categories relevant to the defect
     within the region. Use the supplied reason as context and preserve it verbatim. Do
      not locate other statements, change SEARCH, diagnose a bug, choose skills/actions
     , or generate repaired code.
Return JSON only with a "selections" array. Include exactly one item per supplied
     region and preserve the input order. Each item must contain "STATEMENT_SEMANTICS",
      an array of one or more distinct catalog filenames. Do not return indices or
     other fields.
```

```
Problem and public examples:
{sample_summary(sample)}

Locations:
{json.dumps(locations, ensure_ascii=False)}

Semantic catalog (filename to intent):
{json.dumps(catalog, ensure_ascii=False)}
```

Skill Selection

```
Choose exactly one repair skill for each of the {block_count} blocks.
Use only each block's selected statement type, localization reason, and candidate skill
     names and descriptions. Use the complete source, problem, and original error
     information below as supporting context. Choose the skill whose defect pattern
     best matches the reason. If the specialized skill does not adequately match,
     include the fallback skill when no specialized skill sufficiently matches the
     current defect. Do not invent missing evidence or generate code.
Return JSON only with a "selections" array that contains each block exactly once. Each
     item must contain the integer field "block" and "skill_indices", an array
     containing exactly one 1-based index. The index must address skill_sets[block.
     skill_set] and fall within that block's skill_index_range.

Problem and public examples:
{sample_summary_text}

Complete original source:
{bug_code}

Blocks and skill_sets:
{block_skill_candidates_jsonl}
```

C.3 Repair Skill Utilization for xCodeEval

Fault Localization

```
Return {"locations":[{"SEARCH":"exact original source region","reason":"direct cause of
     the defect"}]}, with 1 to 5 regions.
Each SEARCH must be a nonempty, unique, nonoverlapping exact source substring; preserve
     whitespace using JSON escapes. Only selected regions will be replaced. For
     missing code, include its insertion context; select the whole program if needed.
     Give a brief reason grounded in the problem and available evidence. Do not output
     replacement code.

Problem and public examples:
{sample_summary(sample)}

Complete original source:
{bug_code}
```

Patch Generation

```
Fix a buggy program written in {lang_cluster} language to solve the following
     programming problem:
Description: {prob_desc_description}
Input Specification: {prob_desc_input_spec}
Output Specification: {prob_desc_output_spec}
Sample Input: {sample_input}
Sample Output: {sample_output}
Here is the bug code: {bug_source_code}
Retrieved repair actions: {repair_actions}
Take input from {prob_desc_input_from} and output to {prob_desc_output}

Example response:
```{language}
<<<<<<< SEARCH
{language_specific_example_search}
```
```

```
=======
{language_specific_example_replace}
>>>>>>> REPLACE
```

C.4 Repair Skill Utilization for SWE-bench Multilingual

Recommended Workflow

```
## Recommended Workflow
1. Analyze the codebase by finding and reading relevant files.
2. Create a script to reproduce the issue.
3. Form a concrete hypothesis about the root cause, then edit the source code to test
     that hypothesis.

- Use `edit_code` for source edits. Treat the `file` and `oldcode` passed to `edit_code
     ` as the current hypothesis about the repair location. Call `apply_code_candidate(
     file, candidate_batch_id, candidate)` with a one-based integer to apply an
     existing candidate, or with your own replacement code string to apply a candidate
     you generated yourself. Test every applied candidate. Use `revert_code` only to
     restore oldcode. Use `apply_code_candidate` to switch between candidates.
- Treat source files as read-only when using bash; all source changes must go through `
     edit_code`, `apply_code_candidate`, or `revert_code`.

4. Validate the hypothesis by running the reproduction and relevant regression tests.
     Check whether the observed behavior changed as predicted by the hypothesis.
5. If validation contradicts the hypothesis, reject it. Inspect different callers,
     callees, state owners, or analogous implementations, form a new hypothesis, and
     return to step 3 with a new repair location.
6. Test relevant edge cases and review the patch before submitting.
```