TRACEGEN: SYNTHESIZING DIFFICULTY-CONTROLLABLE BUGS FOR TRAINING AND EVALUATING CODE REPAIR AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Training and evaluating code repair agents both demand large volumes of quality bug data, but existing synthesis methods depend on known patch locations and produce bugs without difficulty labels, leaving training data flat and evaluation unable to distinguish agents at different capability levels. We present TRACEGEN, a framework that mines structured causal paths, termed *DefectChains*, from repair traces and uses them to synthesize bugs at independently discovered code locations without access to the ground-truth patch. The causal depth of each DefectChain yields three difficulty tiers that serve a dual purpose: ordering trajectories for curriculum fine-tuning and producing calibrated evaluation gradients. On SWE-bench Verified seeds, TRACEGEN validates 194 of 318 attempts at a 61.0% rate, while baselines deprived of the original patch location suffer 63–92% validation drops. An agent trained via curriculum SFT on TRACEGEN trajectories achieves **24.7%** on SWE-bench Verified at the 7B scale, surpassing SWE-Dev at 23.4%, R2E-Gym at 19.0%, and Lingma SWE-GPT at 18.2%; curriculum ordering alone contributes 3.3 percentage points over random-order SFT on identical data. All six evaluated frontier models exhibit monotonic resolve-rate increases across the three tiers, confirming the stability of the difficulty gradient.

1 INTRODUCTION

Code repair agents powered by large language models have advanced rapidly: frontier systems now resolve over 70% of SWE-bench Verified instances OpenAI (2026), and open-source frameworks such as SWE-agent Yang et al. (2024), AutoCodeRover Zhang et al. (2024), and Agentless Xia et al. (2024) have made agent-based repair widely accessible. Both *training* and *evaluation* of these agents, however, face a growing data bottleneck. On the evaluation side, SWE-bench-specific analyses identify solution leakage in issue reports and weaknesses in static evaluation Aleithan et al. (2024); OpenAI (2026), raising concerns that performance on a fixed benchmark can overstate generalizable repair ability. On the training side, curating real bugs with executable test suites remains prohibitively expensive; even SWE-Gym Pan et al. (2025) converts only 491 of 2,438 candidate pull requests into usable trajectories, a rate below 21%. Synthetic bug generation can address both needs at once, supplying fresh instances that resist contamination while scaling the volume of training signal.

Existing synthesis methods, however, share two fundamental limitations. **First, they depend on the known patch location.** SWE-Smith Yang et al. (2025) and BugPilot Sonwane et al. (2025) inject or replay defects at or near the site the developer ultimately modified, so the synthesizer never demonstrates that it can identify *where* bugs plausibly reside; it only perturbs code at a known location. When forced to operate at independently discovered locations, SWE-Smith's validation rate drops by 92% and BugInstruct's by 63%, as shown in Figure 2, revealing that their apparent synthesis capability is largely an artifact of location privilege. **Second, no current method produces bugs with structured difficulty labels.** Without such labels, training data is flat: instances can only be consumed in random order, forgoing curriculum strategies that have proven effective in other domains Bengio et al. (2009). Evaluation likewise cannot grade agents along a capability spectrum; every instance is equally "hard" or "easy" depending solely on whether the agent solves it.

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.

These two limitations are not independent. A synthesis method that discovers injection locations autonomously must model the *causal structure* of the bug: which code entities are implicated, how deeply nested the root cause is, and what repair operations are required. Such a structured representation naturally encodes difficulty, because deeper causal chains demand more complex reasoning from the solver. This observation motivates our core insight: *when the synthesis process itself produces structured difficulty information, the same data can serve both curriculum training and graded evaluation*.

The *DefectChain* at the heart of TRACEGEN encodes exactly this structure. Given a seed bug and its repair trace, TRACEGEN extracts a localization chain $\mathcal{L}$ recording the causal path from the observable symptom to the root cause, and a repair chain $\mathcal{R}$ recording the AST-level operations the developer applied. Together, $\mathcal{D} = (\mathcal{L}, \mathcal{R})$ captures what went wrong and how it was fixed at an abstraction level that transfers across code locations. Selectively exposing or withholding nodes from $\mathcal{L}$ in the problem statement produces a difficulty gradient with three tiers. This gradient is unavailable to SWE-Smith Yang et al. (2025) and R2E-Gym Jain et al. (2025), whose synthesized bugs carry no difficulty dimension.

We present TRACEGEN, a framework that closes the loop from synthesis through training to evaluation. The pipeline comprises four stages: (i) *DefectChain Extraction* mines structured causal patterns from seed repair traces; (ii) *Difficulty-Aware Synthesis* discovers candidate injection locations via chain-guided retrieval and synthesizes analogous defects at three difficulty levels without access to the ground-truth patch; (iii) *Trajectory Distillation with Curriculum Training* converts the synthesized bugs into quality-filtered training trajectories ordered from easy to hard; and (iv) *Quality Control* validates every instance through differential testing.

Our contributions are threefold:

- **Location-independent synthesis via DefectChains:** We propose a synthesis framework that extracts causal patterns from seed bugs and discovers injection points at new code locations via chain-guided retrieval, without relying on the original patch location during synthesis. The framework validates bugs at a 61.0% rate on SWE-bench Verified seeds. Baselines deprived of the original patch location see validation drops of 63–92%.
- **Curriculum training for code repair agents:** We introduce a training pipeline that exploits synthesized difficulty labels to order trajectories from easy to hard. The resulting agent achieves 24.7% on SWE-bench Verified at the 7B scale, surpassing SWE-Dev at 23.4% and R2E-Gym at 19.0%; curriculum ordering alone contributes 3.3 percentage points over random-order SFT on identical data.
- **Stable difficulty gradient for evaluation:** We show that DefectChain-guided difficulty control yields consistent evaluation gradients: all six evaluated frontier models exhibit monotonic resolve-rate increases across the three tiers (Figure 3).

2 RELATED WORK

Bug Synthesis for Code LLMs. Dynamic bug generation has emerged as a response to data leakage in static benchmarks Aleithan et al. (2024); OpenAI (2026). SWE-Smith Yang et al. (2025) applies four injection strategies, namely LM-based function rewriting, AST mutations Jia & Harman (2011), random statement deletion, and feature addition, producing 50,137 instances across 128 repositories. SWE-Synth Pham et al. (2025) selects repository-level functions or classes and synthesizes verifiable bug-fix instances, while BugPilot Sonwane et al. (2025) uses agents for intentional bug insertion or feature additions that unintentionally break existing behavior. R2E-Gym Jain et al. (2025) synthesizes problem descriptions via reverse translation from GitHub commits, but the underlying code changes are still drawn from the original commit diff. Garg et al. (2026) propose prompt-level rewrites to disrupt memorization without generating new bugs. These approaches provide scalable bug or task generation, but do not jointly transfer an extracted causal defect structure to retrieved locations and construct matched difficulty variants of the same underlying bug. TRACEGEN targets this combination through DefectChain-guided synthesis and problem statement design that yields three localization-evidence tiers.

Training Code Repair Agents. Fine-tuning on repair trajectories substantially improves agent performance over prompting alone. SWE-Gym Pan et al. (2025) trains on 491 real-PR trajectories via rejection-sampling SFT and achieves 10.6% on SWE-bench Verified at 7B scale; SWE-Smith reaches 15.2% with 5,016 synthetic trajectories; Lingma SWE-GPT Ma et al. (2024) reaches 18.2%; R2E-Gym reaches 19.0% with 3,321; and SWE-Dev Wang et al. (2025) reaches 23.4%. Complementary approaches train specialized retrieval and editing modules Xie et al. (2025) or apply reinforcement learning to software-evolution data Wei et al. (2025). Lingma SWE-GPT already uses an adaptive curriculum that accumulates tasks the current model fails to solve; TRACEGEN instead uses synthesis-time localization-evidence tiers to impose an explicit intra-epoch easy-to-hard ordering on matched task variants.

Curriculum Learning for Code. Curriculum learning Bengio et al. (2009) orders training examples by difficulty to accelerate convergence and improve generalization. The strategy has been widely adopted in NLU and computer vision Wang et al. (2022), and has recently shown promise in code summarization He et al. (2025) and program repair Hao et al. (2024). SWE-Smith assigns post-hoc easy, medium, and hard ratings to generated instances Yang et al. (2025), whereas TRACEGEN constructs matched variants by controlling the localization evidence exposed for the same bug. DefectChain depth and the problem statement design in TRACEGEN supply exactly such a measure, enabling curriculum learning for agent training. Table 1 summarizes how these capabilities distinguish TRACEGEN from prior frameworks.

Table 1: Capability comparison across frameworks. TRACEGEN is the only framework that supports synthesis with difficulty control, curriculum training, and graded evaluation.

Capability	SWE-Gym	SWE-Smith	R2E-Gym	TRACEGEN
Bug synthesis	—	✓	✓	✓
Loc.-indep. synthesis	—	—	—	✓
Difficulty control	—	—	—	✓
Agent training	✓	✓	✓	✓
Curriculum training	—	—	—	✓
Graded evaluation	—	—	—	✓

3 THE TRACEGEN FRAMEWORK

TRACEGEN operates in four stages, as illustrated in Figure 1. Given seed bugs from SWE-bench Verified Jimenez et al. (2024), the framework first extracts a structured *DefectChain* capturing the causal path from symptom to root cause and the developer's repair intent (§3.1). It then discovers candidate injection locations via chain-guided retrieval and deploys an LLM agent to synthesize analogous defects at three difficulty levels (§3.2). A frontier model collects repair trajectories on the synthesized bugs; these trajectories are filtered by quality tier and arranged into a curriculum from easy to hard (§3.3). Quality control validates every instance through differential testing (§3.4).

3.1 DEFECTCHAIN EXTRACTION

A DefectChain $\mathcal{D} = (\mathcal{L}, \mathcal{R})$ pairs a *localization chain* $\mathcal{L}$ with a *repair chain* $\mathcal{R}$. The localization chain traces the causal path from symptom to root cause; the repair chain records the AST-level operations the developer applied. Together, these complementary views capture *where* the defect manifests and *how* it was fixed at an abstraction level that transfers across code locations and repositories.

Localization Chain. Given an issue report and the buggy code, TRACEGEN constructs a directed code graph $G = (V, E)$ from the repository's abstract syntax tree, following the methodology of LocAgent Chen et al. (2025). Nodes represent code entities such as functions, classes, and statements; edges encode structural relationships including call, data-flow, and containment. An LLM analyzes the issue report and traces the causal path through G, producing a *localization chain* $\mathcal{L} = \langle v_{\text{symptom}}, v_1, \ldots, v_{\text{root}} \rangle$. Each $v_i \in V$ lies on the path from the observable symptom to the root

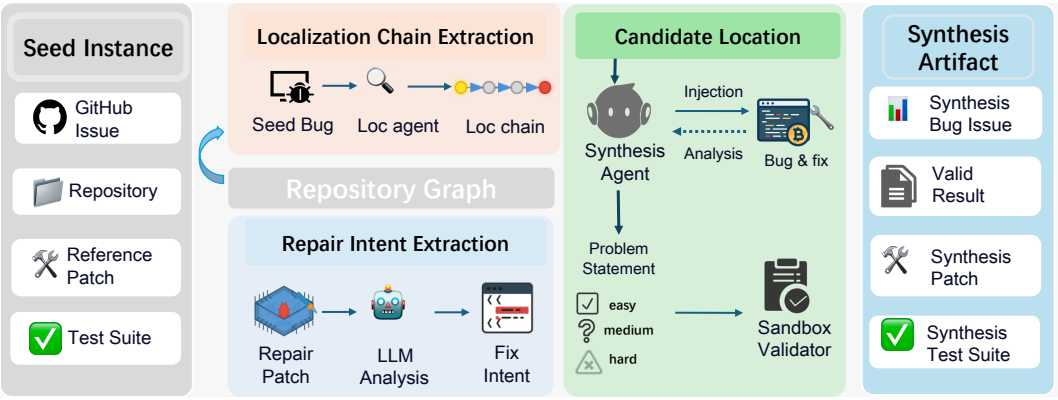

Figure 1: Overview of TRACEGEN. The four stages, DefectChain Extraction, Difficulty-Aware Synthesis, Trajectory Distillation with Curriculum Training, and Quality Control, form a closed loop from synthesis through training to evaluation.

cause v_{root} that the developer ultimately modified. The chain length $|\mathcal{L}|$ serves as a natural difficulty proxy: a length of 2 indicates a shallow single-function defect, whereas a length of 5 or more implies cross-module reasoning requiring the solver to navigate multiple intermediate code entities.

Repair Intent. In parallel, TRACEGEN abstracts the commit diff into a sequence of *repair intents*, atomic AST-level operations that characterize the developer's change. We parse the pre- and post-fix ASTs, compute a tree diff via ChangeDistiller Fluri et al. (2007) and GumTree Falleri et al. (2014), and classify each atomic change into one of ten categories in four groups:

Logic	Condition Refinement, Guard Clause Addition, Exception Fix
Interface	Argument Update, API Replacement
Data Flow	Variable Replacement, Constant Update, Type Cast Fix, Data Initialization
Structure	Statement Insertion

This taxonomy was developed through iterative refinement on the SWE-bench Verified seed set: we manually inspected 100 commit diffs, merged overlapping categories, and split coarse ones until each intent was unambiguously classifiable. The ten categories cover 96.4% of the seed diffs. A seed bug may involve multiple repair intents; their ordered sequence forms the *repair chain* $\mathcal{R} = \langle r_1, \ldots, r_m \rangle$, yielding a complete DefectChain $\mathcal{D} = (\mathcal{L},\ \mathcal{R})$.

3.2 DIFFICULTY-AWARE BUG SYNTHESIS

Candidate Location Discovery. To find locations where the seed's defect pattern can be plausibly reinstantiated, we retrieve candidates using multiple signals from $\mathcal{D}$. Each code node is encoded via a code embedding model. For each root cause $v_{\text{root}} \in \mathcal{L}$, we retrieve the top-$k$ most similar candidates by cosine similarity and then apply two filters derived from the chain:

1. **Depth matching** ($\mathcal{L}$-guided): the candidate's graph depth must lie within ± 1 hop of the seed's root cause depth, ensuring comparable structural depth.
2. **Intent compatibility**, guided by $\mathcal{R}$: the candidate must be structurally capable of hosting at least one repair intent from $\mathcal{R}$. For instance, a Condition Refinement intent requires a conditional statement at the target node.

Candidates passing both filters are ranked by similarity to form the candidate set $\mathcal{C}$. This process operates entirely without knowledge of the original patch location. In practice, we set k=50 and typically retain 5–15 candidates per seed after filtering, balancing coverage against computational cost.

ReAct Agent Synthesis. For each candidate $c_i \in \mathcal{C}$, an LLM agent explores the target codebase before injecting the defect. The agent operates in a ReAct loop Yao et al. (2023) and wields two categories of tools: *graph tools*, namely QueryGraph and ComputeChainMetrics, for navigating the

code graph and computing structural metrics; and *context tools*, namely ReadCode and SearchSimilarNodes, for reading source files and finding structurally similar code patterns. The agent follows a *Mimic Seed Pattern* strategy. It first studies the seed's DefectChain to understand the defect pattern, then inspects the local control-flow and data-flow context at the candidate location, and finally crafts an analogous defect that respects the candidate's specific code structure. The output is a set of artifacts in SWE-bench format: an injection patch that introduces the bug, a fix patch that reverses it, and a test suite that distinguishes the buggy and fixed states.

Chain-Guided Problem Statement Design. Difficulty labels in TRACEGEN are a *byproduct* of synthesis, not a post hoc annotation. By selectively exposing or withholding nodes from $\mathcal{L}$, an LLM generates problem statements at three levels that vary in how much localization evidence the solver receives:

- **Hard**: only the symptom node is exposed, providing a vague traceback or error message with no localization hints; the solver must trace the full causal chain independently.
- **Medium**: intermediate nodes are revealed, providing module-level and API context without naming the faulty function. For example, the problem statement might state that "the error occurs in the `django.db.models` module when processing QuerySet annotations."
- **Easy**: most chain nodes are exposed except the root cause itself, offering function-level cues that narrow the search space to a small code region.

This yields a monotonic information ordering $\text{info}(\text{hard}) < \text{info}(\text{medium}) < \text{info}(\text{easy})$ while preserving the same underlying bug. Unlike approaches that rewrite prompts at the surface level Garg et al. (2026), this mechanism controls *what localization evidence the solver sees*. The resulting gradient is grounded in the DefectChain's causal structure and stable because removing a node deterministically reduces the information available to the solver.

3.3 TRAJECTORY DISTILLATION AND CURRICULUM TRAINING

Trajectory Collection. A frontier model, serving as the backbone of SWE-agent Yang et al. (2024), attempts each bug's easy, medium, and hard problem statements independently, producing up to three trajectories per bug so that a successful easy or medium attempt still yields training signal when the hard variant fails.

Quality-Tier Filtering. We assign each trajectory to one of three quality tiers by submission pathway:

- **Gold**: the agent actively submitted a patch that resolved the bug, as verified by the full test suite.
- **Silver**: the agent submitted via a format-exit pathway, such as an explicit submission command, but did not fully resolve the bug.
- **Bronze**: the agent hit the cost limit or was auto-submitted by the harness.

Each trajectory receives a composite weight equal to the product of a tier weight and a repository weight. Tier weights of 3.0, 1.5, and 0.5 for gold, silver, and bronze emphasize successful repair patterns while retaining signal from partial attempts. The repository weight $w_{\text{repo}} = \sqrt{\bar{n}/n_r}$, clamped to $[0.5, 1.5]$, prevents overrepresented repositories from dominating the gradient signal; for instance, Django contributes roughly half of all validated bugs. Here $\bar{n}$ is the mean trajectory count across repositories and n_r the count for repository r.

Curriculum Learning. The difficulty labels generated during problem statement design provide a natural ordering. TRACEGEN therefore arranges training in a curriculum from easy to hard Bengio et al. (2009):

1. **Phase 1**, easy: the agent learns basic repair patterns, including tool usage, localization strategies, and simple edits, on bugs where most of the localization chain is exposed.

2. **Phase 2**, medium: the agent progresses to cross-module reasoning with only coarse context, such as module name and API surface.
3. **Phase 3**, hard: the agent tackles deep causal chains spanning multiple files with minimal hints, building on capabilities acquired in earlier phases.

Existing pipelines Pan et al. (2025); Yang et al. (2025); Jain et al. (2025) lack difficulty labels and can only present data in random order.

Curriculum Implementation. Each synthesized bug yields up to three trajectories, one per problem statement level. A trajectory's difficulty label is determined by the problem statement that generated it, not by the underlying bug's chain depth alone. For example, a bug with a deep DefectChain still produces an easy trajectory when the solver receives a problem statement exposing most of the chain, so the three phases draw from the same pool of bugs with localization evidence as the only variable.

Concretely, the 2,847 trajectories divide into 936 easy, 971 medium, and 940 hard instances. Before each epoch, the training set is sorted by difficulty level so that easy trajectories appear first, followed by medium and then hard. Within each level, trajectory order is shuffled randomly. This *intra-epoch* curriculum means the model sees the full easy-to-hard progression in every epoch rather than dedicating separate epochs to separate levels.

In the random-order baseline used for ablation, the same 2,847 trajectories are shuffled uniformly without regard to difficulty; both variants share identical hyperparameters and data, isolating the effect of ordering. We neither reset optimizer state between phases nor replay earlier-phase data, as each epoch already revisits the full sequence.

Supervised Fine-Tuning. We fine-tune Qwen2.5-Coder-7B-Instruct with LoRA Hu et al. (2022) at rank 64 and $\alpha = 128$, using DeepSpeed ZeRO-3 Rajbhandari et al. (2020). Each trajectory is formatted as a multi-turn conversation matching the agent scaffold, with tool observations truncated to 16,384 characters to maintain consistency between training and inference. Training runs for 3 epochs over the 2,847 trajectories with a cosine schedule, a warmup ratio of 0.03, and a 32,768-token context window on 8×A800-80GB GPUs.

3.4 Quality Control

Quality control spans four layers, each targeting a distinct failure mode: (i) *Seed filtering* discards chains with fewer than two nodes, roughly 18% of candidates, which lack the causal depth needed for transferable synthesis. (ii) *Intent compatibility checking* ensures that candidate locations can structurally host the seed's repair intent; for example, a candidate lacking conditional statements is rejected when the intent is Condition Refinement. (iii) *Differential testing* validates each bug in an isolated Podman container: the injection patch must trigger targeted test failures while leaving the remaining test suite intact; instances causing collateral failures are discarded. (iv) *Patch sanitization* verifies that no explanatory comments are injected, docstrings remain unchanged, whitespace-only changes are rejected, and the problem statement describes symptoms without revealing the fix location.

4 Experiments

4.1 Experimental Setup

Dataset. We use SWE-bench Verified Jimenez et al. (2024) as the evaluation benchmark, comprising 500 validated instances across 12 Python repositories. From this benchmark we construct a balanced seed set of 200 instances spanning 9 repositories, each paired with 2 injection points selected by the semantic matching module, stratified so that no single repository contributes more than 25% of seeds.

Synthesis Baselines. We compare against three representative synthesis methods: BugInstruct Sonwane et al. (2025), which generates bugs through instruction following; SWE-Smith LM Rewrite Yang et al. (2025), which rewrites functions via LLM prompting; and FeatAdd Sonwane et al. (2025), which treats feature additions as bug proxies. To isolate the effect of injection location from synthesis

strategy, we evaluate each baseline at both its original patch location and at DefectChain locations discovered by TRACEGEN.

Training Baselines. We compare against five published agent training pipelines at the 7B scale: SWE-Gym Pan et al. (2025) with 491 trajectories and 10.6% on Verified, SWE-Smith Yang et al. (2025) with 5,016 trajectories and 15.2%, Lingma SWE-GPT Ma et al. (2024) at 18.2%, R2E-Gym Jain et al. (2025) with 3,321 trajectories and 19.0%, and SWE-Dev Wang et al. (2025) with 2,300 trajectories and 23.4%. All use Qwen2.5-Coder-7B-Instruct Hui et al. (2024) as the base model with standard SFT on randomly ordered data.

Evaluation Models. For the difficulty gradient evaluation, we test six frontier models: Qwen3.5-Plus and Qwen3.5-Flash Qwen Team (2026), Qwen3-Coder-Next Cao et al. (2026), DeepSeek-3.2 DeepSeek-AI (2025), Qwen3-Coder-Plus Qwen Team (2025a), and Kimi-K2 Kimi Team et al. (2026). Each model runs inside SWE-agent with identical scaffolding and tool configurations to ensure fair comparison.

Implementation Details. The synthesis model is Qwen3.6-Max; code embeddings are produced by Qwen3-Embedding-0.6B Qwen Team (2025b). We set the candidate pool to $k{=}50$, allow up to 5 synthesis attempts per seed, and enforce a minimum chain length of 2. For agent training, we use Qwen2.5-Coder-7B-Instruct as the base model and generate 2,847 trajectories with a learning rate of 2×10^{-5} and a batch size of 4.

4.2 MAIN RESULTS

Synthesis Effectiveness. Figure 2 compares validation rates across synthesis methods. TRACEGEN validates 194 of 318 attempts at a 61.0% rate, substantially outperforming all alternatives. When the two strongest baselines are relocated from their original patch sites to DefectChain locations, their validation rates collapse: SWE-Smith LM Rewrite drops from 15.6% to 1.3%, a 92% relative decrease, and BugInstruct from 22.9% to 8.4%, a 63% relative decrease. FeatAdd performs poorly regardless of location, at 1.5% and 1.9% respectively, confirming that feature additions are a fundamentally weak proxy for realistic bugs. The sharp degradation demonstrates that their synthesis capability depends on the known patch location rather than on a genuine ability to discover where bugs plausibly reside; TRACEGEN operates exclusively at DefectChain locations, so no such dependence can arise.

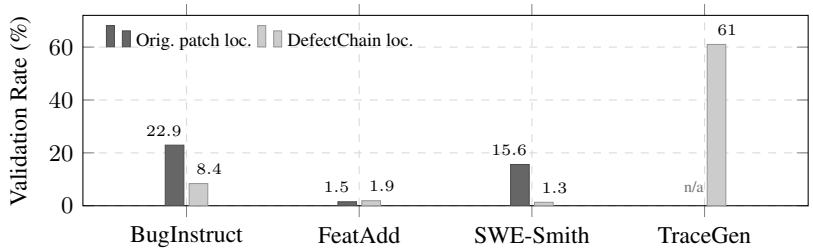

Figure 2: Synthesis validation rates at original and DefectChain locations. Baselines collapse when deprived of the known patch location; TRACEGEN operates exclusively at DefectChain locations, so its original-patch-location slot is marked n/a by design.

Agent Training. Table 2 compares agent performance on SWE-bench Verified. TRACEGEN with curriculum SFT achieves 24.7%, surpassing the best published baseline SWE-Dev at 23.4% by 1.3 percentage points. The curriculum variant outperforms random-order SFT on the same data by 3.3 points, confirming that structured difficulty progression yields more capable agents than uniform random ordering. Notably, TRACEGEN achieves these gains with 2,847 trajectories, comparable in scale to the baselines, rather than through brute-force data expansion.

4.3 ABLATION STUDY

Synthesis Component Ablation. Appendix A ablates each synthesis component: semantic matching is the most critical, and removing it reduces valid instances from 194 to 52, a 73% drop, while removing DefectChain guidance costs over half of the output.

Table 2: Agent training comparison on SWE-bench Verified. All methods use Qwen2.5-Coder-7B-Instruct as the base model.

Method	Source	Traj.	Training	Verified
Base (untrained)	—	—	—	3.2%
SWE-Gym	Real PR	491	SFT	10.6%
SWE-Smith	Synthetic	5,016	SFT	15.2%
Lingma SWE-GPT	Synthetic	—	SFT	18.2%
R2E-Gym	Synthetic	3,321	SFT	19.0%
SWE-Dev	Synthetic	2,300	SFT	23.4%
TRACEGEN (SFT)	Synthetic	2,847	SFT	21.4%
TRACEGEN (Curr.)	Synthetic	2,847	Curr. SFT	**24.7%**

Training Ablation. Table 3 (Appendix B) isolates the effect of curriculum ordering. Both the curriculum and random-order variants train on exactly the same trajectories with identical hyperparameters; the only difference is whether the data is sorted by difficulty within each epoch or shuffled uniformly. Random-order SFT achieves 21.4% while curriculum SFT reaches 24.7%, a gap of 3.3 points attributable solely to the easy-to-hard ordering. Scaling the curriculum from 1,000 to 2,847 trajectories yields an additional 4.3 points. The consistent advantage of curriculum over random ordering at every data scale confirms that the benefit stems from structured ordering rather than data selection or weighting.

4.4 DIFFICULTY ANALYSIS

Information Gradient. Appendix C characterizes information content and agent behavior across difficulty levels: identifier density increases nearly threefold from the hard tier to the easy tier, while git actions decrease by 62%, indicating that harder problems redirect agent effort toward code exploration.

Resolve Rate Gradient. Figure 3 plots resolve rates for six frontier models across the three difficulty levels. All six exhibit monotonic increases from hard to easy, with gains of 8.9–9.9 percentage points. The consistency across diverse architectures and training regimes provides strong evidence that the gradient captures a genuine property of the bugs rather than an artifact of any particular model. The Spearman correlation between seed and synthesized difficulty rankings is weak, with $\rho{=}-0.126$ and $p{=}0.117$, indicating that synthesized difficulty arises from the DefectChain's causal structure at the new location rather than being inherited from the seed.

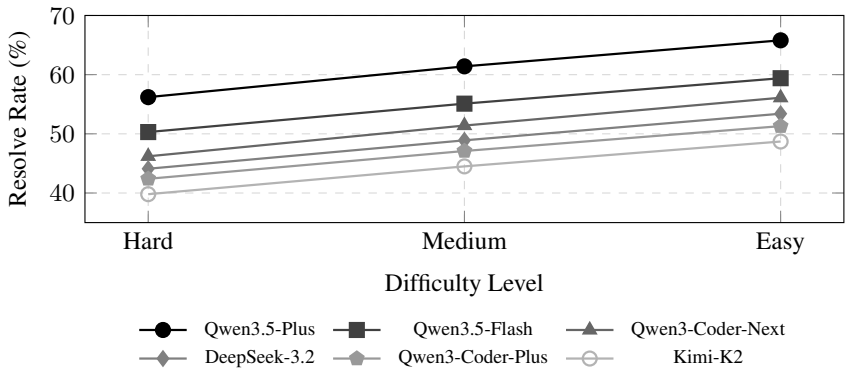

Figure 3: Resolve rates of six frontier models across three difficulty levels. All models show monotonic increases from hard to easy, validating the difficulty gradient.

4.5 ERROR ANALYSIS

To understand when TRACEGEN fails, we manually inspected all 124 invalid synthesis attempts and categorized them into four failure modes:

Collateral test failures, accounting for 38.7% of failures. The injected defect triggers failures in unrelated tests beyond the targeted test suite. This occurs most often when the candidate location lies on a shared code path invoked by multiple modules, causing the defect to propagate beyond the intended scope.

Structural mismatch at the candidate site, at 25.0%. Despite passing the depth and intent filters, the candidate's local code structure differs sufficiently from the seed's that the agent cannot produce a semantically valid injection. For example, a Condition Refinement intent may target a branch that, while syntactically present, guards a fundamentally different computation.

Test inadequacy, at 21.0%. The injected defect is real, but the generated test suite fails to distinguish the buggy and fixed states. The agent-written tests either do not exercise the affected code path or assert on an output that is invariant to the injected change.

Agent exploration failures, at 15.3%. The synthesis agent exhausts its step budget without producing a patch, typically because the candidate's codebase context is too large for the agent to navigate within the allotted steps.

Collateral test failures and structural mismatch together account for nearly two-thirds of all failures. Both are amenable to mitigation: finer-grained test-impact analysis could filter candidates on shared code paths, and stronger structural similarity metrics could reduce mismatch rates.

4.6 Discussion

Why Does Curriculum Ordering Help? The curriculum's benefit likely stems from two complementary mechanisms. First, easy instances establish foundational repair skills, including tool usage, localization heuristics, and edit strategies, that serve as building blocks for harder tasks. An agent that first learns to fix a bug when the faulty function is nearly identified is better positioned to transfer those patterns to scenarios where only a module name is given. Second, the progressive withdrawal of localization evidence mirrors the increasing difficulty of real-world deployment, where issue reports vary widely in specificity, encouraging the agent to develop robust fallback strategies rather than relying on rich cues that may not always be available.

Difficulty as a Training Signal vs. an Evaluation Metric. An important property of TraceGen's difficulty labels is their generality: the same labeling mechanism applies to both training and held-out data. In training, the labels order trajectories for curriculum learning; in evaluation, a separate set of synthesized bugs is partitioned by the same difficulty tiers to reveal capability gradients. This dual use is possible because difficulty is a property of the bug instance, specifically the information content of its problem statement, rather than a property of the model being trained or evaluated. The resulting ranking is stable across models, as shown in Figure 3, and remains valid as models improve, eliminating the need to re-annotate the dataset for each new model generation.

Limitations. Several limitations merit discussion. First, the pipeline is currently limited to Python repositories in SWE-bench; extending DefectChain extraction to statically typed languages such as Java and C++ requires adapting the AST differencing and code graph modules. Second, the curriculum uses three discrete tiers; a continuous difficulty measure could enable smoother training schedules. Third, quality-tier filtering depends on the frontier model's trajectory quality; shifts in model capability may require recalibrating tier weights. Finally, while synthesis operates without the original patch location, it still requires seed bugs to bootstrap DefectChain extraction; fully unsupervised bug discovery remains an open challenge.

5 Conclusion

TraceGen mines DefectChains from repair traces to synthesize difficulty-labeled bugs at independently discovered locations, achieving a 61.0% validation rate, and arranges trajectories into a curriculum that trains a 7B agent to 24.7% on SWE-bench Verified while producing stable evaluation gradients across six frontier models. Future work includes extending DefectChain extraction to statically typed languages and scaling trajectory collection for reinforcement learning.

AI USE STATEMENT

In this work, generative AI models served three roles in the research methodology: generating synthetic bug instances within TRACEGEN, acting as repair agents to produce training trajectories, and serving as the models trained and evaluated in our experiments. These uses are described in the method and experiments sections. All AI-generated artifacts used in the experiments were subjected to the validation and quality-control procedures described in the paper. Generative AI tools were also used for limited manuscript formatting and readability editing. The authors reviewed all AI-assisted work and take responsibility for the final content of this paper, including its text, claims, code, data, and artifacts.

REPRODUCIBILITY STATEMENT

To support reproducibility, the implementation and evaluation code are available at `https://anonymous.4open.science/r/TraceGen-ICLR-0526`.

REFERENCES

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-Bench+: Enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pp. 41–48, 2009.

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. *arXiv preprint arXiv:2603.00729*, 2026.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.426.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2014.

Beat Fluri, Michael Würsch, Martin PInzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33 (11):725–743, 2007.

Spandan Garg, Benjamin Steenhoek, and Yufan Huang. Saving SWE-bench: A benchmark mutation approach for realistic agent evaluation. In *Proceedings of the 4th International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2026. arXiv:2510.08996.

Sichong Hao, Xianjun Shi, and Hongwei Liu. Exploring the potential of pre-trained language models of code for automated program repair. *Electronics*, 13(7):1200, 2024. doi: 10.3390/electronics13071200.

Hongkui He, Jiexin Wang, Liuwen Cao, and Yi Cai. CLCoSum: Curriculum learning-based code summarization for code language models. In *Proceedings of the 33rd IEEE/ACM International Conference on Program Comprehension (ICPC)*, pp. 1–13, 2025. doi: 10.1109/ICPC66645.2025.00035.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

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 *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

Kimi Team et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2026.

Yingwei Ma, Rongyu Liu, Yue Yang, Chao Cao, Binhua Chen, Yongji Cai, et al. Lingma SWE-GPT: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`, February 2026. Accessed: 2026-03-18.

Jiayi Pan, Xingyao Zheng, Hao Chen, Yue Zhang, Jian Zhang, Jianfeng Shang, and Jiawei Han. SWE-Gym: Training software engineering agents and verifiers with SWE-bench. *arXiv preprint arXiv:2412.21139*, 2025.

Minh V. T. Pham, Huy Nhat Phan, Hoang Nhat Phan, Cuong Chi Le, Tien N. Nguyen, and Nghi D. Q. Bui. SWE-Synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Qwen Team. Qwen3-Coder: Agentic coding in the world, 2025a. URL `https://qwenlm.github.io/blog/qwen3-coder/`. Accessed 2026-09-26.

Qwen Team. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025b.

Qwen Team. Qwen3.5-Plus and Qwen3.5-Flash. Qwen API Platform, 2026. URL `https://qwen.ai/apiplatform`. Accessed 2026-09-26.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. BugPilot: Complex bug generation for efficient learning of SWE skills. *arXiv preprint arXiv:2510.19898*, 2025.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-Dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3742–3761. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-acl.193.

Xin Wang, Yudong Chen, and Wenwu Zhu. A survey on curriculum learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):4555–4576, 2022. doi: 10.1109/TPAMI.2021.3069908.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2629.

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

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. SWE-Fixer: Training open-source LLMs for effective and efficient GitHub issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1123–1139. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.findings-acl.62/.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. Spotlight Poster.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

A Synthesis Component Ablation

Figure 4 quantifies each component's contribution. Semantic matching is the most critical: removing it reduces valid instances from 194 to 52, a 73% drop. Without DefectChain guidance the framework still outperforms all baselines with 94 valid instances but loses over half its output, suggesting that DefectChain provides a structured prior over the code graph that substantially narrows the search space. Removing intent alignment or depth filtering each causes a 71% drop; the similar magnitude indicates complementary failure modes: intent alignment rejects structurally incompatible candidates, while depth filtering rejects candidates at inappropriate graph positions.

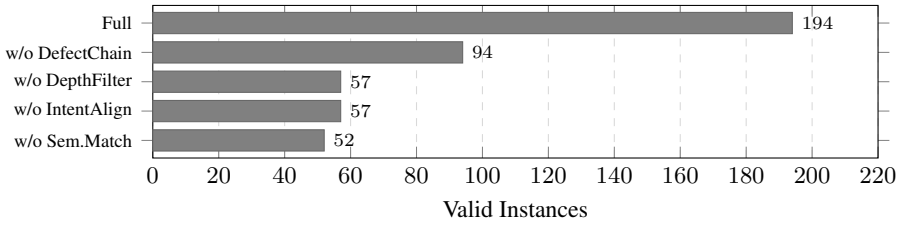

Figure 4: Synthesis ablation. Each bar removes one component from the full pipeline. Semantic matching contributes the largest single effect (-73%).

B Training Ablation

Table 3 isolates the effect of curriculum ordering at each data scale.

Table 3: Training ablation on SWE-bench Verified. Curriculum ordering outperforms random ordering at each data scale.

Trajectories	Random SFT	Curriculum SFT
1,000	17.8%	20.4%
2,000	20.2%	22.8%
2,847	21.4%	**24.7%**

C Problem Statement Information Gradient

Table 4 characterizes information content and agent behavior across difficulty levels. Identifier density increases nearly threefold from 0.066 at the hard tier to 0.193 at the easy tier, confirming that easier problem statements provide more explicit localization pointers. Git actions decrease by 62% from hard to easy, dropping from 24.6 to 9.3, while test actions remain stable between 24.8 and 30.9, indicating that agents facing harder problems redirect substantially more effort toward code exploration. Total step count also decreases from 97.6 to 92.4, suggesting that richer localization cues not only redirect effort but also reduce the overall work required.

Table 4: Problem statement characteristics and agent behavior across difficulty levels.

	Problem Statement		Agent Behavior			
Level	Tokens	ID Den.	Steps	Search	Git	Test
Hard	14.1	0.066	97.6	30.8	24.6	24.8
Medium	19.6	0.116	93.2	33.2	14.0	31.1
Easy	28.3	0.193	92.4	33.4	9.3	30.9