Rapid Fixes, Gradual Failures: Iterative Self-Correction Dynamics in Large Language Models for Program Synthesis

Anonymous Authors[1]

Abstract

Achieving one-shot success in code generation remains challenging for large language models (LLMs). Iterative self-repair, in which models refine their own programs in response to feedback, has been proposed as a potential remedy, but its effectiveness for programming tasks is not well understood. We study the *dynamics* of this loop across three frontier LLMs (GPT-4, GPT-5.1, Claude Sonnet 4.5), four execution-based benchmarks (HumanEval, MBPP+, APPS-Intro, APPS-Comp), two languages (Python and Java), and two history conditions (h=0: critique only; h=1: full refinement history). Three findings emerge. (i) Recovery is overwhelmingly a *task-level* property: whether at least one of the model's five initial samples already passes the task (its pass@1) predicts seed-level recovery far more strongly (ρ=0.71) than the failing seed's own initial pass fraction (ρ=0.31). Failing seeds on tasks where at least one other initial sample already passed are 36$\times$ more likely to recover than failing seeds on tasks where every initial sample failed. (ii) Refinement is heavily front-loaded: 53% of the total pass-fraction gain occurs at attempt 1, recovered and non-recovered seeds diverge *immediately* (a 46 pp gap in pass fraction by attempt 1), and a substantial fraction of non-recovered seeds ($\sim$38$\%$ at h=0, $\sim$49$\%$ at h=1) reproduce the same wrong program for ten attempts. (iii) Adding the full refinement history does not improve recovery rate (p>0.35) and on harder benchmarks actively degrades it ($-$7.5 pp on APPS-Comp).

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Program synthesis is the task of automatically generating computer programs that satisfy a given specification, such as input-output examples or a test suite. This task has been a long-standing goal in the fields of Artificial Intelligence (AI), software engineering, and programming languages (PL) research. Early approaches primarily rely on search-based techniques, including bottom-up enumerative search, which constructs small program fragments and systematically combines them into larger programs until a correct solution is generated (Alur et al., 2017; Odena et al., 2020; Lee, 2021). Additional methods include counterexample-guided inductive synthesis, where a verifier generates and incorporates counterexamples into the specification, thereafter prompting the system to start a new search for candidate programs (Solar-Lezama et al., 2006); genetic programming, which evolves a population of programs through mechanisms inspired by biological evolution, such as crossover, mutation, duplication, and deletion (Koza et al., 1989; Koza, 1994; Sobania et al., 2022); and constraint-based techniques that reformulate the task as a satisfiability problem, using SMT or SAT solvers to search for candidate programs that satisfy the specified input-output constraints (Ellis et al., 2015).

Search-based program synthesis faces scalability challenges. These methods are often limited by domain-specific languages (DSLs), which restrict expressiveness and flexibility. Moreover, as program complexity increases, the search space expands rapidly, often resulting in timeouts on complex, industry-scale benchmarks. Recent advances in language models, especially large language models (LLMs), have enabled the generation of computer programs of unprecedented length and complexity. This progress has led to developer-oriented tools such as GitHub Copilot (developed by Microsoft and OpenAI) and Amazon Q Developer. Beyond code assistants, LLM-based methods have been used for a wide range of software engineering tasks, including automated test generation (Chen et al., 2024b; Harman et al., 2025; Wang et al., 2024; Shin et al., 2024; Molina et al., 2025), automated code repair (Xia & Zhang, 2022; 2023; Xia et al., 2023; Yin et al., 2024), and code documenta-

tion (Luo et al., 2024).

Nevertheless, code generated by LLMs is susceptible to various errors that differ markedly from those in human-written code. For example, Tambon et al. (Tambon et al., 2025) examined 333 bugs from code generated by three leading LLMs (CodeGen, PanGu-Coder, and Codex) and identified ten distinct bug patterns: Misinterpretations (20.77%), Syntax Error, Silly Mistake, Prompt-biased Code, Missing Corner Case (15.27%), Wrong Input Type, Hallucinated Object, Wrong Attribute, Incomplete Generation, and Non-Prompted Consideration. Another study found that, compared to human programmers, Copilot's solutions to programming tasks have a lower correctness ratio and less diversity (Dakhel et al., 2023). They also found that Copilot frequently suggests buggy, difficult-to-understand solutions, which is particularly problematic for novice programmers (Dakhel et al., 2023). Vaithilingam et al. (Vaithilingam et al., 2022), through a within-subjects user study, observed that Copilot did not improve task completion time or success rate, but often reduced the need for online searches. Most importantly for this work, achieving one-shot success, where the model generates a fully correct solution in a single attempt, remains difficult. Even when syntactically valid code is produced, it may fail test cases because of logical errors, unaddressed edge cases, or other factors. To address these issues, self-correction techniques have been proposed, enabling models to iteratively refine their outputs (Dou et al., 2024).

A wide range of approaches has been developed to enhance LLM-based code generation. These include model-scale improvements, such as Codex, a GPT-3-based model with up to 12 billion parameters, pre-trained on 159 GB of code drawn from 54 million GitHub repositories (Chen et al., 2021), and StarCoder (Li et al., 2023), which was trained on one trillion tokens spanning over 80 programming languages, including data from GitHub issues, commits, and Jupyter notebooks. It was further fine-tuned on 35 billion Python tokens to produce the final StarCoder model. More recently, the Code World Model (CWM) was introduced as a 32-billion-parameter, open-weight model designed to go beyond static code learning. CWM is mid-trained on large-scale observation-action traces from Python interpreter sessions and agent-driven Docker environments, and further optimized via multi-task reinforcement learning across domains such as verifiable coding, mathematics, and multi-turn software engineering tasks (Copet et al., 2025).

Beyond model advancements, reasoning decomposition techniques—such as Chain-of-Thought, Tree-of-Thoughts, and modularization-of-thought—divide complex problems into smaller, manageable reasoning steps (Li et al., 2025). Additional strategies include planning (Jiang et al., 2024), retrieval-augmented generation (RAG) (Du et al., 2024; Parvez et al., 2021; Lu et al., 2022), and, most pertinent to this study, self-debugging methods that direct models to analyze the code they produce (**?**Olausson et al., 2023).

In this paper, we study *iterative self-refinement* for code generation: a single trajectory of up to ten attempts in which each attempt conditions on the failure of the previous one rather than being sampled independently. Huang et al. (2024) evaluate intrinsic self-correction in reasoning (a three-step loop of generation, self-critique, and revision with no external oracle) and report that accuracy degrades across GPT-3.5, GPT-4, and Llama-2; their results are aggregate accuracy after the loop, not per-attempt dynamics. Olausson et al. (2024) are the closest measurement on code: their "repair tree" samples n_f critiques and n_r repairs per program independently of prior repairs, so the tree compares refinement against i.i.d. sampling at matched compute rather than tracing a single iterative trajectory. We hold the loop structure fixed (single agent, one critique-and-revise step per attempt, greedy decoding, identical seeds across conditions) and ask three questions prior code-empirical work has not jointly addressed: (i) does intrinsic self-correction help in code under genuine iteration, where $h{=}0$ supplies only the previous program with no execution feedback, (ii) does adding refinement history with per-attempt pass fractions ($h{=}1$) improve over the intrinsic baseline, and (iii) how does pass rate evolve attempt by attempt across up to ten iterations. We evaluate three frontier models (GPT-4, GPT-5.1, Claude Sonnet 4.5).

We evaluate our approach on four execution-based benchmarks. First, we use MBPP+, an updated version of the original MBPP dataset with additional test cases; the model is tasked with generating a function that satisfies a given signature and natural language specification (Liu et al., 2023). We also evaluate on HumanEval-X (Java and Python), which similarly tasks the model with generating functions from a signature and specification. Functional correctness is measured by executing the generated code against test cases (Chen et al., 2021). Lastly, we use APPS, a dataset of 10,000 Python problems with 131,777 test cases (21.2 per problem on average), spanning introductory, interview, and competition-level difficulties. As with the other benchmarks, correctness is determined by executing candidate solutions against test cases (Hendrycks et al., 2021).

We investigate the following research questions:

1. RQ1: Does a seed's initial pass fraction predict recovery?
2. RQ2: Does a task's initial pass@1 predict recovery of its failing seeds?
3. RQ3: How does pass rate evolve over refinement attempts?

4. RQ4: Does history improve self-repair?
5. RQ5: How does programming language affect self-correction?
6. RQ6: How does task difficulty impact self-repair?

Our results reveal several consistent patterns. First, a seed's initial pass fraction is a statistically significant but weak predictor of recovery (Spearman $\rho = 0.31$ overall under h=0), while a task's initial pass@1 is a much stronger predictor ($\rho = 0.71$): failing seeds on tasks where at least one seed has already passed are roughly $36\times$ more likely to recover than failing seeds on tasks where every seed initially fails. Second, most of the gain is front-loaded: 53% of the total pass-fraction gain across ten attempts arrives at attempt 1, and recovered and non-recovered seeds diverge *immediately*, with 37.7% of non-recovered seeds reproducing identical pass fractions for ten attempts under h=0 (rising to 49.3% under h=1). Third, providing the model with its own refinement history does not improve recovery rates (all paired tests $p > 0.35$) and degrades them on harder benchmarks (APPS Comp -7.5 pp); among seeds that do recover, history yields a small reduction in attempts to recovery ($p = 0.054$). Fourth, recovery rate decreases with task difficulty ($H = 83.2$, $p = 10^{-18}$), but the drop is a step function rather than a gradient: HumanEval is an outlier at 69% while the three harder benchmarks cluster at 35–39$\%$. Finally, the monotonic per-attempt decline is language-agnostic, occurring in both Python and Java, though models degrade faster in Java (-0.04 to -0.06 per iteration vs. -0.03 to -0.04 in Python) and stabilize earlier (iteration 5 vs. 7).

2. Related Work

2.1. Program Synthesis

Program synthesis—the task of automatically generating programs that satisfy a given specification—has been a long-standing area of study in programming languages and artificial intelligence (Gulwani et al., 2017; Manna & Waldinger, 1971; Solar-Lezama, 2008). Traditional synthesis methods typically rely on symbolic techniques, such as constraint solving (Alur et al., 2013), deductive search (Manna & Waldinger, 1980; 1992), or program enumeration (Odena et al., 2020; Shi et al., 2022).One notable example is Sketch (Solar-Lezama, 2013), which was the first system to represent the set of candidate solutions to a synthesis problem as a satisfiability (SAT) problem. It allows programmers to write partial programs with 'holes,' where a control function maps each hole to a specific value. Using synthesis semantics that define the meaning of a program P through a denotation function $CP_{\tau\emptyset}(\sigma_{in}, \Phi) \rightarrow (\sigma_{out}, \Phi')$, the search for a valid control function is framed as a constraint satisfaction problem. These semantics allow the derivation of a set of constraints on φ in terms of the input state σ, ensuring that each assignment satisfies the program's specification across all possible inputs. These techniques are particularly effective when specifications are complete, concise, and formally structured (e.g., input-output examples, partial programs, logical formulas).

However, traditional approaches often struggle to scale to complex domains because they require extensive manual specification engineering and are prone to combinatorial explosion as the search space expands. Recent advances in machine learning have shifted focus toward neural program synthesis (Ellis et al., 2019; Lee et al., 2018; Parisotto et al., 2016; Zohar & Wolf, 2018). These methods train models—typically using sequence-to-sequence or graph-based architectures—to directly map specifications to programs. Neural synthesis methods offer several advantages over purely symbolic approaches: they can generalize across related tasks and handle noisy or under-specified specifications more effectively. However, these purely neural methods often lack formal correctness guarantees and rely on learning statistical patterns rather than employing structured reasoning.

Large language models (LLMs), such as Codex (Chen et al., 2021), Code Llama (Roziere et al., 2023), and StarCoder (Li et al., 2023), represent a recent evolution in neural program synthesis. These models are trained on vast corpora of code and natural language, allowing them to generate plausible programs from natural language descriptions, examples, or partial code contexts. However, because LLM-based synthesis is fundamentally probabilistic, it often struggles to produce a correct solution in a single attempt (Huang et al., 2023; Liu et al., 2024).

2.2. Self-Correction in Large Language Models

Huang et al. (2024) coin the term *intrinsic self-correction*, wherein the model attempts to correct its initial response (e.g., seed program) based on its inherent capabilities—without external feedback. External feedback includes input from humans, models, or tools (e.g., static analysis), whereas internal feedback relies on the model's inherent knowledge and parameters to reassess its outputs. The authors propose a three-stage prompting framework: (1) initial generation, (2) prompting the model to review its previous response and generate feedback on it, and (3) prompting the model to generate a new response for the task based on that feedback. Their findings reveal a fundamental limitation: LLMs are more likely to modify correct answers into incorrect ones than the reverse, suggesting that models struggle to judge the correctness of their own reasoning. They further compare intrinsic self-correction with multi-agent debate and self-consistency (Wang et al., 2022), finding that

ensemble-style methods improve robustness by aggregating multiple reasoning paths. Unlike this line of work, which focuses on reasoning tasks and internal critique, we evaluate *self-repair* in executable programming settings, where correctness is externally verifiable through unit tests.

Is Self-Repair a Silver Bullet for Code Generation? (Olausson et al., 2024) studied self-repair in code synthesis using a structured sampling design rather than true iterative refinement. Their approach constructs a "repair tree" by generating n_p initial programs, n_f feedback strings per program, and n_r repairs per feedback, yielding approximately $n_p + n_p n_f n_r$ total samples per task. However, each node in this tree is sampled independently, without conditioning on previous repairs or maintaining a historical context. The study found that improvements from self-repair were modest compared to i.i.d. sampling and that increasing the diversity of initial programs n_p often yields larger gains than expanding the feedback or repair branches. In contrast, our framework performs *true iterative self-repair*, where each subsequent attempt builds upon the model's own evolving trajectory of code and feedback. Moreover, we introduce a compact structural representation of this trajectory through anti-unification, which prior work does not include.

Teaching Large Language Models to Self-Debug (Chen et al., 2024a) demonstrated that models can improve generated code by interpreting execution traces in natural language. Their four-stage pipeline (generation, execution, explanation, and repair) leverages test outputs to guide refinement, a process akin to *rubber-duck debugging*. While their focus is on leveraging external signals such as test case failures, our work complements this by examining the structure of the model's own evolving repair attempts—introducing a symbolic mechanism to compress and abstract that history.

3. Methodology

We evaluate three frontier LLMs (GPT-4 `gpt-4-0613`, GPT-5.1 `gpt-5.1-2025-11-13`, and Claude Sonnet 4.5 `claude-sonnet-4-5-20250929`) on four execution-based code-generation benchmarks: HumanEval (Chen et al., 2021), its Java translation HumanEval-Java (Zheng et al., 2023), MBPP+ (Austin et al., 2021; Liu et al., 2023), and APPS (Hendrycks et al., 2021). We use all 164 HumanEval (Python and Java) tasks, 120 MBPP+ tasks (28.1% of the sanitized split), 60 APPS-Introductory, and 60 APPS-Competition tasks; the multiplicative cost of the design ($n_p \times A \times h \times$ models $\times$ datasets) forced subsampling on MBPP+ and APPS. Per-dataset test-execution details and full model specifications are in Appendix A.

For each task we generate $n_p = 5$ independent seed programs at sampling temperature $\tau_{\text{gen}} = 0.7$ (Figure 1). Each seed is executed against the task's test suite to compute a

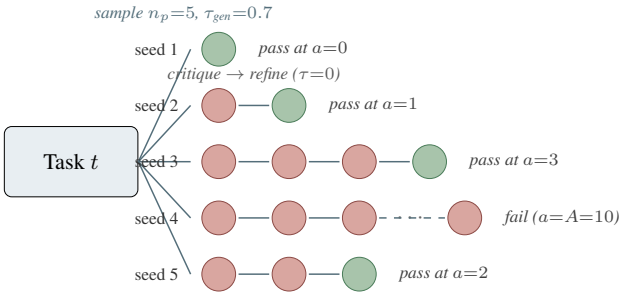

Figure 1. Iterative self-repair across $n_p = 5$ independent seeds. For each task, five seed programs are sampled at $\tau_{\text{gen}} = 0.7$ and executed against the test suite. A seed passing all tests at attempt 0 terminates immediately (seed 1); a failing seed enters a refinement loop of up to $A = 10$ iterations, each consisting of a critique step and a refined program, both with greedy decoding ($\tau = 0$). Refinement terminates when the seed passes (seeds 2, 3, 5) or when the iteration budget is exhausted (seed 4). Under h=0 the critique conditions on the current program only; under h=1 it additionally receives the full history of prior attempts and their pass fractions. Each row depicts one possible per-seed trajectory; filled green/red circles indicate per-attempt pass/fail.

pass fraction $\phi_i^{(0)} \in [0, 1]$; seeds with $\phi_i^{(0)} = 1$ are marked solved. Failing seeds enter a refinement loop indexed by $a = 1, \ldots, A$ with $A = 10$. At each iteration, the model generates a natural-language critique $f_i^{(a)}$, then a revised program $p_i^{(a)}$ conditioned on $p_i^{(a-1)}$ and $f_i^{(a)}$, then $p_i^{(a)}$ is executed to compute $\phi_i^{(a)}$. Critique and refinement both use greedy decoding ($\tau = 0$), so any iteration-to-iteration change is attributable to the model's reasoning rather than sampling noise. The same model plays all three roles (seed generation, critique, refinement); no external verifier or separate critic is introduced. This isolates *intrinsic* self-correction.

Two conditions vary the information available during critique. In $h = 0$, the prompt contains the task description and the current program $p_i^{(a-1)}$ only. In $h = 1$, the prompt additionally includes all prior attempts $\{(p_i^{(0)}, \phi_i^{(0)}), \ldots, (p_i^{(a-1)}, \phi_i^{(a-1)})\}$, each annotated with its pass fraction. Both conditions use the *same* initial seeds: seeds are generated once under $h = 0$ and reused under $h = 1$, so observed differences reflect the history treatment alone and not seed variability.

Let T be the set of tasks and let $a^*_{t,i} = \min\{a \in \{0, \ldots, A\} : \phi_i^{t,(a)} = 1\}$ (with $a^*_{t,i} = \infty$ if no attempt solves task t). **Initial pass rate** is $\text{IP} = \frac{1}{|T| n_p} \sum_{t,i} \mathbb{1}[\phi_i^{t,(0)} = 1]$. Let $\mathcal{F} = \{(t, i) : \phi_i^{t,(0)} < 1\}$ be the set of initially-failing seeds. **Recovery rate** is $\text{Rec} = |\{(t, i) \in \mathcal{F} : a^*_{t,i} \leq A\}| / |\mathcal{F}|$, and **average attempts to recovery** is $\bar{a} = |\mathcal{R}|^{-1} \sum_{(t,i) \in \mathcal{R}} a^*_{t,i}$ over the recovered set $\mathcal{R} = \{(t, i) \in \mathcal{F} : a^*_{t,i} \leq A\}$. **Final pass rate** is $\text{FP} = \frac{1}{|T| n_p} \sum_{t,i} \mathbb{1}[a^*_{t,i} \leq A] = \text{IP} + (1 - \text{IP}) \cdot \text{Rec}$.

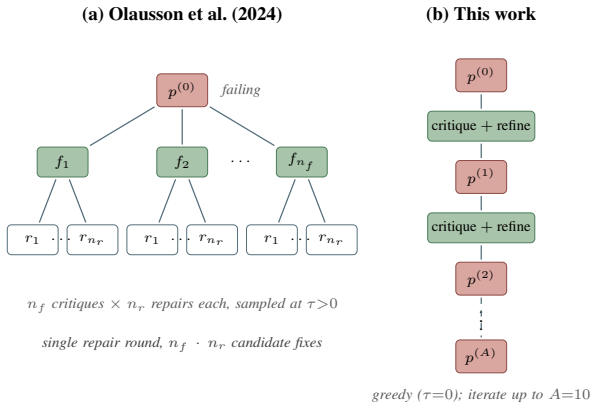

Figure 2. Self-repair sampling-budget contrast, starting from one failing initial program $p^{(0)}$. (a) Olausson et al. (Olausson et al., 2024) sample n_f critiques and n_r repairs per critique at nonzero temperature, evaluating $n_f \cdot n_r$ candidate fixes in a single repair round. (b) We generate one greedy critique and one greedy refined program per attempt, iterating up to $A = 10$ attempts. Both designs additionally start from n_p initial programs (not shown). The two trade depth for breadth in opposite directions: theirs explores many alternative fixes shallowly, ours explores one chain deeply.

We also report task-level pass@k for $k \in \{1, 5\}$ using the unbiased estimator $1 - \binom{n_p - c_t}{k} / \binom{n_p}{k}$ of Chen et al. (2021), where $c_t = \sum_i \mathbb{1}[a^*_{t,i} \leq A]$ is the count of solved seeds for task t and $n_p = 5$. Recovery rate is best read within-model across history conditions because cross-model contrasts confound the refinement signal with differences in $\mathcal{F}$. We quantify the IP–Rec relationship with Spearman's ρ over seeds (Appendix A).

4. Results

4.1. RQ1: Does a Seed's Initial Pass Fraction Predict Recovery?

Initial pass fraction is a weak but statistically significant predictor of recovery. A univariate logistic regression on the failing seeds yields $\beta = 1.75$: each 10 pp increase in IP multiplies recovery odds by $e^{0.175} = 1.19$ (Figure 5). Predicted recovery rises from 27% at IP $= 0$ to 47% at IP $= 50\%$ and 64% at IP $= 90\%$: a 37 pp spread across the full range. Pooled Spearman $\rho = 0.31$ ($p = 10^{-23}$) under h=0. A multivariate model that adds benchmark and model identity attenuates the IP coefficient only modestly ($\beta = 1.47$); benchmark difficulty contributes a comparable signal ($\beta_{\text{HumanEval}} = 1.48$, reference: APPS Comp), and model capability enters separately (GPT-5.1 $\beta = 1.35$, GPT-4 $\beta = 0.61$, Claude reference). No single factor dominates.

Most of the discriminative power lives at the boundary IP $=$ 0. Seeds that pass zero tests recover 22.5% of the time; seeds that pass at least one test recover 49.0% ($\chi^2 = 86.2$, $p = 10^{-20}$, 2.2× odds ratio). Thresholds at IP $< 5\%$ and IP $< 10\%$ yield nearly identical splits. The operative distinction is total failure versus any partial success, not a continuous gradient over IP. The signal also concentrates on harder benchmarks. AUC for IP as a continuous predictor rises from 0.559 on HumanEval (near random) to 0.691 on MBPP+, 0.603 on APPS Intro, and 0.731 on APPS Comp. The Spearman gradient mirrors it ($\rho = 0.10$, $p = 0.16$ on HumanEval; $\rho = 0.28$, $p = 10^{-5}$ on MBPP+; $\rho = 0.17$, $p = 0.002$ on APPS Intro; $\rho = 0.37$, $p = 10^{-15}$ on APPS Comp). The per-benchmark range of ρ (0.10 to 0.37) exceeds the per-model range (0.20 to 0.36), so task difficulty drives the gradient more than model identity.

Adding repair history weakens this relationship for every benchmark. AUC under h=1 decreases from 0.559 $\rightarrow$ 0.550 (HumanEval), 0.691 $\rightarrow$ 0.628 (MBPP+), 0.603 $\rightarrow$ 0.552 (APPS Intro), and 0.731 $\rightarrow$ 0.657 (APPS Comp); pooled ρ falls from 0.31 (h=0) to 0.21 (h=1). Including prior failed attempts in the prompt does not strengthen the IP-to-recovery mapping; it dampens it. The implication for early stopping is asymmetric: low IP reliably predicts failure on the hardest tasks (APPS Comp AUC 0.731) but is uninformative on the easiest (HumanEval AUC 0.559), motivating difficulty-conditioned thresholds rather than a single global rule.

4.2. RQ2: Does a Task's Initial Pass@1 Predict Recovery of Its Failing Seeds?

Whether at least one of the model's five initial samples already solves the task (the task's pass@1) predicts seed-level recovery far better than the failing seed's own initial pass fraction does. Pooled Spearman $\rho = 0.71$ ($p = 10^{-58}$, $n = 1{,}243$ failing seeds), more than double the IP correlation ($\rho = 0.31$, RQ1). We partition tasks by pre-refinement pass@1 into Q1 (pass@1 $= 0$, all five initial seeds fail) and Q2 (pass@1 > 0, at least one initial seed already passes), then measure recovery of the remaining failing seeds (Figure 6). Seeds refine independently and do not share information; a Q2 task's already-passing seed cannot directly help its failing siblings. pass@1 > 0 is therefore a signal that the task lies inside the model's capability envelope, not a mechanism that aids repair.

Failing seeds on Q1 tasks recover 21.9% of the time; failing seeds on Q2 tasks recover 90.4%, and in 84.9% of Q2 tasks *every* failing seed recovers ($\chi^2 = 450$, $p = 10^{-100}$, odds ratio 34×). The pattern is uniform across benchmarks (HumanEval 41.8% $\rightarrow$ 96.4%, APPS Comp 19.3% $\rightarrow$ 90.8%) and across models (Claude 18.1% $\rightarrow$ 89.6%, GPT-5.1 35.2% $\rightarrow$ 93.8%). At the task level, 34.5% of Q1 tasks (those where every initial seed failed) yield at least one passing seed after refinement: refinement can unlock previously-unsolved tasks, but the majority of these tasks remain un-

Dataset	Diff.	h	Model	IP	Rec.	Avg. Att.	FP	Δ
HumanEval	–	0	Claude	**99%**	17%	**1.0**	**99%**	+0
HumanEval	–	0	GPT-4	82%	81%	3.6	97%	+15
HumanEval	–	0	GPT-5.1	95%	**87%**	2.6	**99%**	+4
HumanEval	–	1	GPT-4	82%	**86%**	2.2	98%	+16
MBPP+	–	0	Claude	**88%**	29%	9.3	87%	−1
MBPP+	–	0	GPT-4	84%	**43%**	**3.5**	85%	+1
MBPP+	–	0	GPT-5.1	85%	43%	8.2	**89%**	+4
MBPP+	–	1	Claude	88%	**58%**	7.4	**92%**	+4
APPS	Intro	0	Claude	54%	**35%**	**1.8**	**70%**	+16
APPS	Intro	0	GPT-4	25%	28%	2.6	46%	+21
APPS	Intro	0	GPT-5.1	**55%**	27%	3.1	67%	+12
APPS	Comp.	0	Claude	**45%**	44%	**2.7**	**69%**	+24
APPS	Comp.	0	GPT-4	3%	13%	3.7	16%	+13
APPS	Comp.	0	GPT-5.1	30%	**49%**	2.8	65%	+35
APPS	Comp.	1	GPT-5.1	30%	**52%**	2.8	67%	+36
HumanEval-Java	–	0	Claude	**96%**	65%	**1.2**	**99%**	+3
HumanEval-Java	–	0	GPT-4	79%	48%	3.2	89%	+10
HumanEval-Java	–	0	GPT-5.1	93%	**88%**	2.1	**99%**	+7
HumanEval-Java	–	1	Claude	96%	**84%**	2.2	99%	+3

Table 1. Self-repair across datasets, models, and history settings. **IP**: initial pass rate. **Rec.**: recovery rate among initially-failing seeds. **Avg. Att.**: mean attempts to first recovery (among recovered seeds). **FP**: final pass rate. $\Delta = \mathrm{FP} - \mathrm{IP}$. We show one row per (dataset, model, h) and elide redundant h=1 rows where h=1 matches h=0 within 1 pp; the full 30-row table is Table 4 in the appendix. 5 seeds per task; up to A=10 attempts per failing seed.

solved. Recovery is overwhelmingly a property of whether the model can solve the task at all, not of any individual seed's quality.

4.3. RQ3: How Does Pass Rate Evolve Over Refinement Attempts?

Let $\mathrm{pf}_i^{(a)}$ denote the pass fraction of failing seed i at attempt $a \in \{0, 1, \ldots, 10\}$, where $a = 0$ is the initial generation and $a \geq 1$ are refinement attempts. Once $\mathrm{pf}_i^{(a)} = 1$ refinement terminates and the seed is held at 1.0. For recovered seeds we define the recovery attempt as

$$a_i^* = \min\{a \geq 1 : \mathrm{pf}_i^{(a)} = 1\},$$

the first refinement step at which the seed passes all tests. Refinement is strongly front-loaded. Mean pass fraction across $n = 1{,}243$ failing seeds rises from 35.0% at attempt 0 to 47.8% at attempt 1 (+12.8 pp), then crawls to 59.0% by attempt 10 under h=0 (Figure 7); the h=1 trajectory is virtually indistinguishable (35.8% → 46.3% → 58.6%). The first attempt alone delivers 53% of the total gain. By attempt 3, 76% of the total gain has been realized, and the remaining seven attempts contribute 5.8 pp combined. Among recovered seeds the median $a_i^* = 1$: 51.1% recover on attempt 1, 77.4% by attempt 3, 87.3% by attempt 5. Difficulty shifts the absolute pass-fraction levels but not the front-loading. HumanEval moves 48.6% → 69.0% → 83.6% at attempts 0, 1, and 10; APPS Comp moves 29.1% → 40.2% → 51.7% over the same span.

The aggregate trajectory hides a sharp dichotomy. Splitting failing seeds into recovered ($n = 521$) and non-recovered ($n = 722$) groups under h=0, the two are already separable at attempt 0 (46% vs. 27% mean pf, $p = 10^{-27}$) and diverge sharply at attempt 1 (75% vs. 28%, gap 46.5 pp, $p = 10^{-99}$; Figure 3). Non-recovered seeds flatline near 28–30% for all ten attempts. Stagnation is endemic: 37.7% of non-recovered seeds (272/722) reproduce *identical* pass fractions across attempts 1–10 under h=0, rising to 49.3% under h=1. The model emits the same wrong code repeatedly. The pattern holds on every benchmark, with attempt-1 gaps of 35.4 pp (HumanEval), 47.1 pp (MBPP+), 40.9 pp (APPS Intro), and 48.6 pp (APPS Comp), all $p < 10^{-12}$.

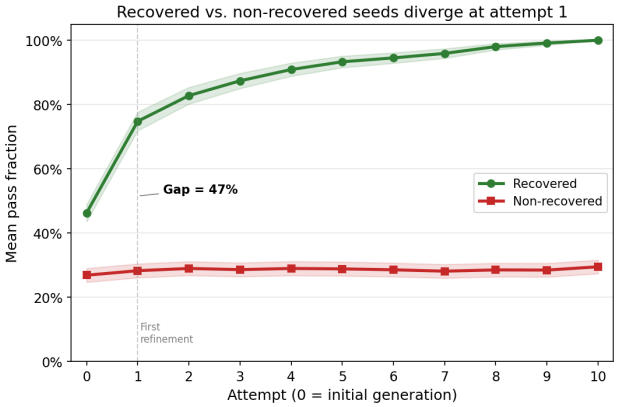

Figure 3. RQ3b: Mean pass fraction trajectory for recovered (green, $n = 521$) vs. non-recovered (red, $n = 722$) seeds under h=0. Shaded bands show 95% confidence intervals. The two groups are already distinguishable at attempt 0 (46% vs. 27%, $p = 10^{-27}$) and diverge sharply at attempt 1 (75% vs. 28%, $p = 10^{-99}$). Non-recovered seeds flatline near 28–30% for all 10 attempts.

The single strongest predictor of recovery is the first-attempt delta $\Delta\text{pf}_1 = \text{pf}_i^{(1)} - \text{pf}_i^{(0)}$ (Spearman $\rho = 0.44$, $p = 10^{-61}$, $n = 1{,}243$). Seeds with $\Delta\text{pf}_1 > 0$ recover 71.5% of the time; seeds with $\Delta\text{pf}_1 \leq 0$ recover only 23.7% ($\chi^2 = 273.8$, $p = 10^{-61}$). One refinement attempt's delta carries more information than the seed's initial state ($\rho = 0.44$ vs. RQ1's $\rho = 0.31$). Any *regression* during refinement is a strong negative signal: seeds that ever produce a worse program than a prior attempt recover 22.5% of the time vs. 57.4% for those that never do ($\chi^2 = 151.7$, $p = 10^{-35}$). Combined with the timing distribution, this supports an aggressive early-stopping rule. If attempt 1 does not improve the pass fraction, the seed has only a 24% chance of eventual recovery.

4.4. RQ4: Does History Improve Self-Repair?

History does not reliably improve recovery. Pooled across all 12 matched (dataset, model) conditions, h=0 recovers 41.9% of failing seeds vs 38.5% under h=1, a non-significant difference of −3.4 pp ($\chi^2 = 2.83$, $p = 0.093$). At the condition level, 4 of 12 pairs favour h=1, 7 favour h=0, and 1 is tied (Table 2); none of Wilcoxon ($W = 22.0$, $p = 0.37$), paired t ($t = 0.96$, $p = 0.36$), or sign-flip permutation ($p = 0.38$) approach significance. The mean condition-level delta is −5.0 pp with range $[-44.0, +28.9]$ pp. Both conditions use the same initial seeds, so this null is not a power problem from seed variance.

The negative trend concentrates by model and by benchmark. Per model, Claude is most negatively affected by history ($\Delta = -10.3$ pp), GPT-5.1 is moderately so (−6.3 pp), and GPT-4 is roughly neutral (+1.4 pp). Per dataset, MBPP+ is the only benchmark where history yields a positive effect (+9.6 pp); HumanEval shows the largest mean decrease (−15.2 pp), driven by GPT-5.1's sharp drop (86.8% → 42.9%). The single largest individual loss is Claude on APPS Comp (44.8% → 24.3%, −20.5 pp). The negative effect also scales with difficulty: APPS Comp drops −7.5 pp under h=1 vs −3.5 pp on APPS Intro. History does not expand the model's capability envelope; when it has any detectable effect it perturbs seeds on tasks already inside it, consistent with the asymmetric Q1/Q2 split in RQ2.

Dataset	Model	h=0	h=1	Δ (pp)
APPS Comp	Claude	44.8%	24.3%	−20.5
APPS Comp	GPT-4	14.1%	14.0%	−0.1
APPS Comp	GPT-5.1	60.5%	65.9%	+5.3
APPS Intro	Claude	44.1%	23.9%	−20.2
APPS Intro	GPT-4	31.4%	37.6%	+6.2
APPS Intro	GPT-5.1	39.8%	32.5%	−7.3
HumanEval	Claude	2.8%	2.8%	+0.0
HumanEval	GPT-4	81.0%	69.9%	−11.1
HumanEval	GPT-5.1	86.8%	42.9%	−44.0
MBPP+	Claude	28.9%	57.8%	+28.9
MBPP+	GPT-4	42.9%	47.6%	+4.8
MBPP+	GPT-5.1	42.9%	40.8%	−2.0

Table 2. Per-condition recovery rates under h=0 and h=1. The delta is h=1 minus h=0; positive values indicate that history improves recovery. No consistent direction emerges.

Among seeds that recover, history yields a small, marginally significant reduction in attempts to first success. The pooled mean recovery attempt is 2.56 (h=0) vs 2.39 (h=1), $\Delta = -0.16$, median 1 under both. The pooled Mann–Whitney test is non-significant ($U = 125{,}254$, $p = 0.54$), but the paired Wilcoxon on condition means is marginally significant ($W = 11.0$, $p = 0.054$). The effect is concentrated in Claude (−0.5 to −1.6 attempts depending on dataset); GPT-4 and GPT-5.1 are flat. When history helps at all, it accelerates seeds that were already going to recover rather than rescuing seeds that were not.

4.5. RQ5: How Does Programming Language Affect Self-Correction?

Initial pass rates are comparable across languages: under h=0, mean IP across models is 92% on HumanEval-Python vs 89% on HumanEval-Java, a 3 pp gap (Table 1). The interesting language effect is dynamic. While RQ3's cumulative-best pass fraction rises monotonically, the *per-attempt* pass rate (the fraction of failing seeds that produce a fully-passing program at iteration a specifically, not cumulatively) declines monotonically in both languages and both history conditions. Aggregated across all Python benchmarks and models, the per-attempt rate falls from ≈23% at attempt 1 to ≈1% at attempt 10 (Figure 4). Spearman correlations between iteration index and per-attempt rate are negative across every condition: Python $\rho = -0.88$ (h=1), −0.87

(h=0); Java $\rho = -0.52$ (h=1), -0.75 (h=0); 11 of the 12 per-model cells in Table 3 are negative, ranging from -0.96 (GPT-4 on HumanEval) to $+0.09$ (Claude on MBPP+, n.s.). The cumulative-best and per-attempt curves move in opposite directions because the population of still-failing seeds at attempt a is selected for seeds the model could not fix earlier; the residual gets harder over time.

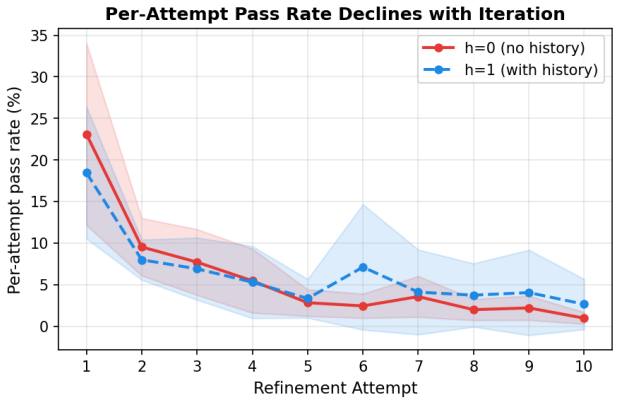

Figure 4. Aggregate per-attempt pass rate across all Python benchmarks and models, comparing h=0 and h=1. Shaded regions show 95% confidence intervals. Both conditions decline sharply after attempt 1, with diminishing returns beyond attempt 5.

Java degrades faster and stabilizes sooner. Linear-trend slopes for the per-attempt rate are -0.04 to -0.06 per iteration in Java vs -0.03 to -0.04 in Python. On HumanEval-Java the mean per-attempt rate stops meaningfully changing by iteration 5; on HumanEval-Python it continues changing through iteration 7. Java's stricter type system and compilation step appear to constrain the refinement space: candidates that fail to compile or typecheck are rejected at execution time, so the model has fewer plausible-but-wrong moves available on later attempts. The trajectory therefore stabilizes earlier in Java than in Python, and at a lower final pass rate.

4.6. RQ6: How Does Task Difficulty Impact Self-Repair?

We isolate task difficulty by comparing APPS Introductory and APPS Competition: two difficulty levels within the same benchmark suite, sharing task format (`stdin/stdout` programs), test harness, and evaluation protocol. Recovery rate is statistically indistinguishable across difficulty levels: 36.6% (115/314 failing seeds) on Intro vs. 34.8% (192/551) on Comp ($\chi^2 = 0.20$, $p = 0.65$, h=0). Mean attempts to first recovery are also comparable (2.68 vs. 2.77, median 2 for both, Mann–Whitney $p = 0.72$). Difficulty alone does not determine whether self-repair succeeds, nor how fast.

The visible difficulty effect is in predictiveness, not outcome. AUC for IP as a continuous predictor of recovery rises from 0.603 on Intro to 0.731 on Comp, and the Spearman IP–recovery correlation more than doubles ($\rho = 0.17$, $p = 0.002$ on Intro vs. $\rho = 0.39$, $p = 10^{-22}$ on Comp). On easy tasks even seeds with low IP can recover, so IP cannot discriminate; on hard tasks, low IP reliably predicts failure. The same gradient holds across the broader benchmark set (RQ1): IP discriminates better as difficulty rises, motivating difficulty-conditioned early stopping rather than a single global threshold.

The most striking effect is across models (Figure **??**). On APPS Intro, Claude (44.1%), GPT-5.1 (39.8%), and GPT-4 (31.4%) cluster within 13 pp. On APPS Comp the three models diverge: GPT-5.1 *improves* to 60.5% (1.52× its Intro rate), Claude holds at 44.8%, and GPT-4 collapses to 14.1% (0.45× Intro). The 46.4 pp gap between GPT-4 and GPT-5.1 on APPS Comp is the largest interaction effect in our data: models differ fundamentally in how they handle algorithmic complexity under self-repair, and the gap widens with difficulty rather than collapsing as the easy ceiling is removed.

5. Discussion

A seed's initial pass fraction explains little of its fate ($\rho = 0.31$), but the task's initial pass@1 is a strong predictor ($\rho = 0.71$): failing seeds on tasks where at least one initial sample already passed are ∼36× more likely to recover than failing seeds on tasks where every initial sample failed, and this gap is stable across benchmarks and models. Most recovery happens on attempt 1 (∼53% of the total pass-fraction gain), recovered and non-recovered seeds diverge immediately, and adding more attempts past the first contributes little. Adding the full sequence of prior programs and critiques as feedback does not help on average and actively hurts on harder benchmarks (-7.5 pp on APPS Comp under h=1, with Claude dropping from 44.8% to 16.5%); since we deliberately withheld failing inputs and exception traces from the history channel, this is not a noise-overload result but a sign that even the cleanest refinement history can compound self-reinforcing errors when each attempt conditions on its predecessors. Recovery barely moves between difficulty levels within a benchmark (APPS Intro 36.6% vs. APPS Comp 34.8%). The larger gaps come from benchmark-format effects: HumanEval at 69% is the outlier. Java models degrade ∼30–50% faster per iteration than Python. The defensible recommendations are to refine only failing seeds whose task already has at least one passing initial sample (these recover at ∼90% vs. ∼25% otherwise), to early-stop when attempt 1 produces no meaningful pass-fraction increase, and to default to h=0 with h=1 treated as a per-task, per-model configuration that must be empirically validated rather than assumed beneficial; conclusions drawn

from HumanEval alone will systematically over-estimate recovery in production settings.

Impact Statement

This paper presents an empirical study of how large language models behave when asked to repair their own code. The work is descriptive: it does not propose a new model and does not introduce new generation capabilities. Its likely societal effect is to make existing iterative-repair pipelines *cheaper and more honest* about when refinement helps and when it does not. We see no ethical considerations beyond those generally associated with machine-learning research.

References

Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. *Syntax-guided synthesis*. IEEE, 2013.

Alur, R., Radhakrishna, A., and Udupa, A. Scaling enumerative program synthesis via divide and conquer. In *International conference on tools and algorithms for the construction and analysis of systems*, pp. 319–336. Springer, 2017.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *International Conference on Learning Representations*, 2024a.

Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., and Yin, J. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 572–576, 2024b.

Copet, J., Carbonneaux, Q., Cohen, G., Gehring, J., Kahn, J., Kossen, J., Kreuk, F., McMilin, E., Meyer, M., Wei, Y., et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., and Jiang, Z. M. J. Github copilot ai pair programmer: Asset or liability? *Journal of Systems and Software*, 203:111734, 2023.

Dou, S., Jia, H., Wu, S., Zheng, H., Zhou, W., Wu, M., Chai, M., Fan, J., Huang, C., Tao, Y., et al. What's wrong with your code generated by large language models? an extensive study. *arXiv preprint arXiv:2407.06153*, 2024.

Du, X., Zheng, G., Wang, K., Zou, Y., Wang, Y., Deng, W., Feng, J., Liu, M., Chen, B., Peng, X., et al. Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. *arXiv preprint arXiv:2406.11147*, 2024.

Ellis, K., Solar-Lezama, A., and Tenenbaum, J. Unsupervised learning by program synthesis. *Advances in neural information processing systems*, 28, 2015.

Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., and Solar-Lezama, A. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Harman, M., Ritchey, J., Harper, I., Sengupta, S., Mao, K., Gulati, A., Foster, C., and Robert, H. Mutation-guided llm-based test generation at meta. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 180–191, 2025.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Huang, B., Lu, S., Chen, W., Wan, X., and Duan, N. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*, 2023.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations*, 2024.

Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., and Jiao, W. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

Koza, J. R. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

Koza, J. R. et al. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, volume 89, pp. 768–774, 1989.

Lee, W. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL): 1–28, 2021.

Lee, W., Heo, K., Alur, R., and Naik, M. Accelerating search-based program synthesis using learned probabilistic models. *ACM Sigplan Notices*, 53(4):436–449, 2018.

Li, J., Li, G., Li, Y., and Jin, Z. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2): 1–23, 2025.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Liu, F., Liu, Y., Shi, L., Huang, H., Wang, R., Yang, Z., Zhang, L., Li, Z., and Ma, Y. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

Lu, S., Duan, N., Han, H., Guo, D., Hwang, S.-w., and Svyatkovskiy, A. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*, 2022.

Luo, Q., Ye, Y., Liang, S., Zhang, Z., Qin, Y., Lu, Y., Wu, Y., Cong, X., Lin, Y., Zhang, Y., et al. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.

Manna, Z. and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

Manna, Z. and Waldinger, R. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674, 1992.

Manna, Z. and Waldinger, R. J. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

Molina, F., Gorla, A., and d'Amorim, M. Test oracle automation in the era of llms. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–24, 2025.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., and Dai, H. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., and Solar-Lezama, A. Is self-repair a silver bullet for code generation? *International Conference on Learning Representations*, 2024.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Parvez, M. R., Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, 2021.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Shi, K., Dai, H., Ellis, K., and Sutton, C. Crossbeam: Learning to search in bottom-up program synthesis. *arXiv preprint arXiv:2203.10452*, 2022.

Shin, J., Harzevili, N. S., Aleithan, R., Hemmati, H., and Wang, S. Retrieval-augmented test generation: How far are we? *arXiv preprint arXiv:2409.12682*, 2024.

Sobania, D., Briesch, M., and Rothlauf, F. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pp. 1019–1027, 2022.

Solar-Lezama, A. *Program synthesis by sketching*. University of California, Berkeley, 2008.

Solar-Lezama, A. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, 2013.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 404–415, 2006.

Tambon, F., Moradi-Dakhel, A., Nikanjam, A., Khomh, F., Desmarais, M. C., and Antoniol, G. Bugs in large

language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):65, 2025.

Vaithilingam, P., Zhang, T., and Glassman, E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pp. 1–7, 2022.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Wang, Z., Liu, K., Li, G., and Jin, Z. Hits: High-coverage llm-based unit test generation via method slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1258–1268, 2024.

Xia, C. S. and Zhang, L. 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.

Xia, C. S. and Zhang, L. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

Xia, C. S., Wei, Y., and Zhang, L. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494. IEEE, 2023.

Yin, X., Ni, C., Wang, S., Li, Z., Zeng, L., and Yang, X. ThinkRepair: Self-directed automated program repair. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Wang, Z., Shen, L., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5673–5684, 2023.

Zohar, A. and Wolf, L. Automatic program synthesis of long programs with a learned garbage collector. *Advances in neural information processing systems*, 31, 2018.

A. Methodology Details

A.1. Datasets

HUMANEVAL (Chen et al., 2021) consists of 164 Python programming problems, each defined by a function signature, a docstring, and a private test suite not visible to the model. We use all 164 tasks. We also evaluate on **HUMANEVAL-JAVA**, a Java translation of HumanEval from the HumanEval-X suite (Zheng et al., 2023), on the same 164 problems.

MBPP+ (Liu et al., 2023) is an enhanced version of the Mostly Basic Python Problems benchmark (Austin et al., 2021); each task's original test cases are augmented with additional automatically generated assertions. We sample 120 tasks (28.1%) from the sanitized test split of 427 problems. Each task includes 3–4 public unit tests used as the repair oracle, plus a private test harness of 100–250 assertion-based pairs that we use as a stricter correctness oracle in supplementary analyses. Subsampling is required by the multiplicative cost structure of the experiment: $n_p = 5$ seeds $\times$ $A = 10$ iterations $\times$ 2 history conditions $\times$ 3 models $\times$ 4 datasets generates a combinatorial number of API calls.

APPS (Hendrycks et al., 2021) contains competitive programming problems with `stdin`/`stdout` interfaces. Each problem is evaluated against a set of input/output test cases that are not revealed to the model. Following Olausson et al. (2024), we sample 60 `introductory`-level and 60 `competition`-level problems from the test split.

A.2. Models

GPT-4 (`gpt-4-0613`) is an earlier-generation OpenAI model accessed via the Chat Completions API. It supports an 8,192-token context window and up to 8,192 output tokens, with a knowledge cutoff of December 2023.

GPT-5.1 (`gpt-5.1-2025-11-13`) is OpenAI's flagship model at the time of our experiments. It supports a 400,000-token context window and up to 128,000 output tokens, with a knowledge cutoff of September 2024.

Claude Sonnet 4.5 (`claude-sonnet-4-5-20250929`) is Anthropic's intermediate-scale model in the Claude 4 family. It supports a 200K-token context window and up to 64K output tokens, with a knowledge cutoff of January 2025.

A.3. Test execution

Test execution differs by dataset:

- **MBPP+** and **HUMANEVAL (Python)**: the generated program is executed in-process via Python's `exec()` into a shared namespace, and each assertion-based test case is run individually with a per-test timeout.
- **HUMANEVAL-JAVA**: the generated class is compiled and executed via the JVM in a separate process, with a per-test timeout of 10 seconds.
- **APPS**: each generated program is executed as an isolated OS-level subprocess via `trace_runner.py`, receiving test input on `stdin`. The program's `stdout` is compared to the expected output by exact string match. Each test case has an independent timeout of 10 seconds, matching the protocol of Olausson et al. (2024).

A seed's pass fraction is the number of test cases passed divided by the total number of test cases. A seed is *solved* iff its pass fraction equals 1.0. To prevent infinite loops from hanging execution, every program is run in an isolated OS-level process with a per-test time limit; processes that do not terminate within the limit have the corresponding test marked failed. Seeds that time out during initial generation are counted as failures.

A.4. Iterative loop, in detail

For each failing seed $p_i^{(0)}$, the iterative loop runs for $a = 1, \ldots, A$ with $A = 10$:

1. **Feedback.** The model emits a natural-language critique $f_i^{(a)}$. Under h=0, the prompt contains the task description and the current program $p_i^{(a-1)}$ only. Under h=1, it additionally contains all prior attempts $\{(p_i^{(0)}, \phi_i^{(0)}), \ldots, (p_i^{(a-1)}, \phi_i^{(a-1)})\}$, each annotated with its pass fraction; the prompt does *not* include explicit error messages, exception traces, or test inputs/outputs. For attempts $a < 3$, the h=1 condition behaves identically to h=0 (since fewer than two prior attempts exist). Critique is generated with greedy decoding ($\tau = 0$).

2. **Refinement.** The model emits a revised program $p_i^{(a)}$ conditioned on the task description, $p_i^{(a-1)}$, and $f_i^{(a)}$. Refinement is also greedy ($\tau = 0$).

3. **Execution.** $p_i^{(a)}$ is executed against the test suite to compute $\phi_i^{(a)}$. If $\phi_i^{(a)} = 1$, the seed's trajectory terminates and the task is marked solved. Each of the n_p seeds runs its own independent repair loop; once any seed solves the task, no further refinement is performed for that task.

A.5. Spearman correlation conventions

For each failing seed i, let $x_i = \mathrm{IP}_i = |\{t \in T : \text{seed}_i \text{ passes } t\}|/|T|$ denote the fraction of tests passed initially, and let $y_i = \mathbb{1}[\text{seed}_i \text{ recovers}]$ be a binary indicator. We compute Spearman's $\rho(x, y)$ over all failing seeds; the unit of observation is a single seed.

A.6. Per-(dataset, model) Spearman correlations for RQ4

Table 3. Spearman ρ between iteration index and per-attempt pass rate (h=0). Negative values indicate monotonic decline.

Dataset	Claude	GPT-4	GPT-5.1
HumanEval	-0.52	-0.96	-0.77
MBPP+	$+0.09$	-0.55	-0.24
APPS Intro	-0.71	-0.79	-0.91
APPS Comp	-0.44	-0.46	-0.92

B. Additional Figures

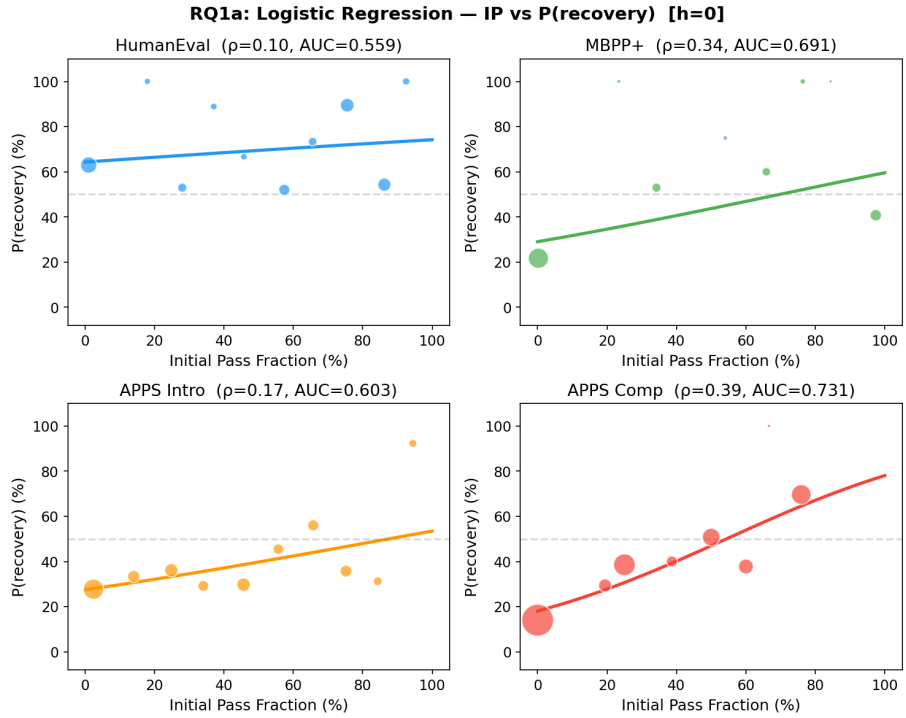

Figure 5. RQ1: Logistic regression fit of initial pass fraction (IP) vs. probability of recovery under the h=0 (no history) condition, by benchmark. Each bubble represents seeds binned by initial pass rate in 10 pp intervals; bubble size is proportional to the number of seeds in each bin. The logistic curve is steepest on APPS Competition (AUC $= 0.731$, $\rho = 0.39$), indicating that initial pass rate is most predictive for the hardest tasks. On HumanEval the curve is nearly flat (AUC $= 0.559$, $\rho = 0.10$): recovery success is largely independent of how many tests the seed initially passes. Across all benchmarks, the relationship is positive but weak—initial correctness helps but does not determine repair outcome.

Dataset	Diff.	h	Model	IP	Rec.	Avg. Att.	FP	Δ
HumanEval	–	0	Claude	**99%**	17%	**1.0**	**99%**	+0%
HumanEval	–	0	GPT-4	82%	81%	3.6	97%	**+15%**
HumanEval	–	0	GPT-5.1	95%	**87%**	2.6	**99%**	+4%
HumanEval	–	1	Claude	**99%**	17%	**1.0**	**99%**	+0%
HumanEval	–	1	GPT-4	82%	**86%**	2.2	98%	**+16%**
HumanEval	–	1	GPT-5.1	95%	75%	1.4	**99%**	+4%
MBPP+	–	0	Claude[†]	**88%**	29%	9.3	87%	−1%
MBPP+	–	0	GPT-4[‡]	84%	**43%**	**3.5**	85%	+1%
MBPP+	–	0	GPT-5.1[†]	85%	**43%**	8.2	**89%**	**+4%**
MBPP+	–	1	Claude[†]	**88%**	**58%**	7.4	**92%**	**+4%**
MBPP+	–	1	GPT-4[‡]	84%	48%	**3.5**	87%	+2%
MBPP+	–	1	GPT-5.1[†]	85%	41%	7.7	88%	+3%
APPS	Intro	0	Claude	54%	**35%**	**1.8**	**70%**	+16%
APPS	Intro	0	GPT-4	25%	28%	2.6	46%	**+21%**
APPS	Intro	0	GPT-5.1	**55%**	27%	3.1	67%	+12%
APPS	Intro	1	Claude	54%	21%	**1.7**	64%	+10%
APPS	Intro	1	GPT-4	25%	**33%**	2.6	50%	**+25%**
APPS	Intro	1	GPT-5.1	**55%**	24%	3.4	**66%**	+11%
APPS	Comp.	0	Claude	**45%**	44%	**2.7**	**69%**	+24%
APPS	Comp.	0	GPT-4	3%	13%	3.7	16%	+13%
APPS	Comp.	0	GPT-5.1	30%	**49%**	2.8	65%	**+35%**
APPS	Comp.	1	Claude	**45%**	44%	**2.4**	**69%**	+24%
APPS	Comp.	1	GPT-4	3%	13%	4.2	15%	+12%
APPS	Comp.	1	GPT-5.1	30%	**52%**	2.8	67%	**+36%**
HumanEval-Java	–	0	Claude	**96%**	65%	**1.2**	**99%**	+3%
HumanEval-Java	–	0	GPT-4	79%	48%	3.2	89%	**+10%**
HumanEval-Java	–	0	GPT-5.1	93%	**88%**	2.1	**99%**	+7%
HumanEval-Java	–	1	Claude	**96%**	**84%**	**2.2**	**99%**	+3%
HumanEval-Java	–	1	GPT-4	78%	43%	4.7	87%	**+10%**
HumanEval-Java	–	1	GPT-5.1	93%	75%	2.6	98%	+5%

Table 4. Self-repair results across datasets, models, and history settings. **IP**: initial pass rate; **Rec.**: recovery rate among initially-failing seeds; **Avg. Att.**: mean attempts to first recovery (conditional on recovery); **FP**: final pass rate; $\Delta = \mathrm{FP} - \mathrm{IP}$. 50 tasks (MBPP+), 164 tasks (HumanEval, HumanEval-Java), 60 tasks (APPS Intro, APPS Comp), 5 seeds each. [†]Evaluated against private test harness (100–250 pairs). [‡]49 tasks (task 20 excluded due to infinite-loop test case).

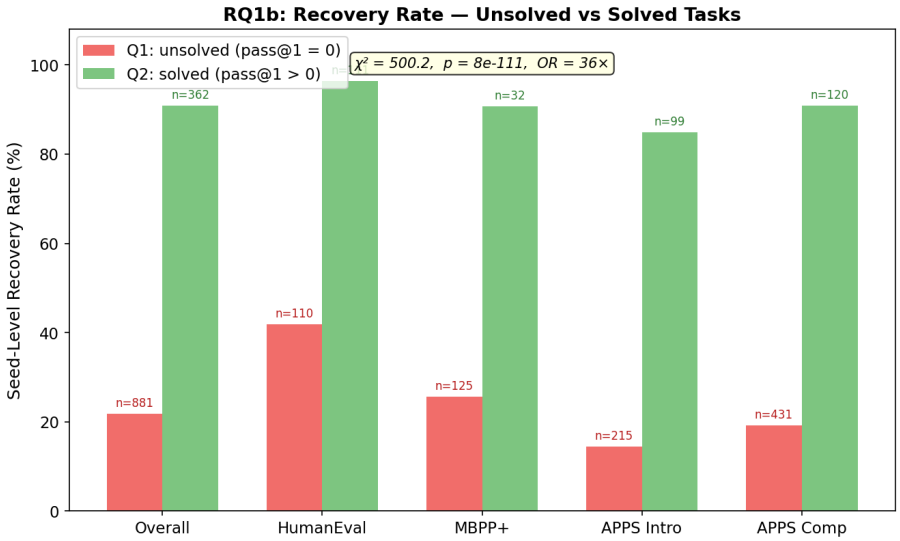

Figure 6. RQ2: Seed-level recovery rate for unsolved tasks (Q1, pass@1 $= 0$) versus partially solved tasks (Q2, pass@1 > 0), overall and per benchmark. Labels indicate the number of failing seeds in each group. The gap is consistent across all benchmarks ($\chi^2 = 500.2$, OR $= 36\times$).

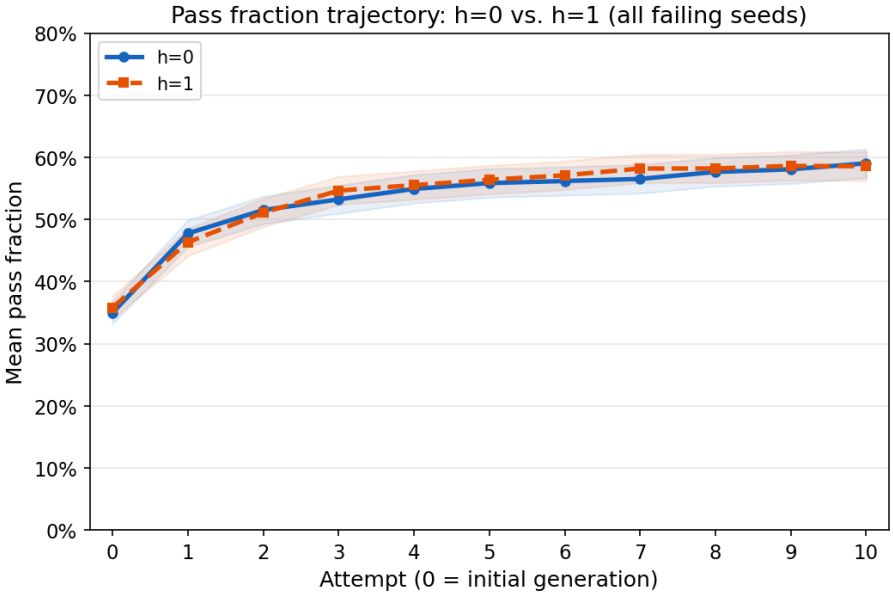

Figure 7. Mean pass fraction trajectory for all failing seeds under h=0 ($n = 1{,}243$) and h=1 ($n = 1{,}223$). Shaded bands show 95% confidence intervals. The two conditions are virtually indistinguishable, indicating that history does not affect the trajectory shape.