WHEN AI REVIEWS ITS OWN CODE: RECURSIVE SELF-TRAINING COLLAPSE IN CODE LLMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive self-training can degrade neural generative models when generated data is reused without fresh human data or external quality control. We study this risk in **code LLMs**, where AI-generated code can enter real repositories, later become training data, and create a repository-scale self-training loop. While software development traditionally interrupts this loop through *pull-request review*, tests, compilation, and human approval, AI coding tools now produce code faster than humans can review it, and code review itself is increasingly automated by AI systems. We therefore compare three recursive fine-tuning regimes: **no review**, **human-style exogenous-gate review** using model-independent filters such as compilation and static quality checks, and **AI-self-gate review** using the code LLM's own signals such as perplexity and binary self-scoring. Across multiple code LLMs and benchmarks, every tested recursive regime falls below its own starting point; exogenous filters preserve executable validity and sometimes reduce decline, while *AI-self-gate* filters can look strong early but later lose their filtering effect. In the clearest case, the binary self-gate enters a rubber-stamp regime where acceptance scores rise while benchmark correctness falls. We explain this behavior by formulating review as gated distributional reweighting, proving that AI self-gating degenerates to ungated self-training under a self-confirming acceptance condition, and giving a spectral analysis of representation-level covariance concentration under recursive retraining. These results suggest that stable recursive code LLM training requires exogenous verification rather than model-coupled self-review.

1 INTRODUCTION

Recursive self-training has been repeatedly shown to be harmful for neural generative models when the loop is not anchored by fresh human data or an external quality signal. In image generation, self-consuming training can reduce distributional variance and drive models toward low-diversity outputs, a failure mode described as model amplification disorder Alemohammad et al. (2023). In general generative modeling, training on recursively generated data can cause model collapse: distributional errors accumulate, low-probability modes are lost, and the learned distribution drifts away from the original data distribution Shumailov et al. (2024); Dohmatob et al. (2024). Similar effects have been reported for language models, where synthetic text can reduce diversity, amplify earlier mistakes, and degrade long-horizon quality unless real data or strong external filtering is preserved in the training loop Briesch et al. (2023); Seddik et al. (2024); Suresh et al. (2025). These results give a general warning: self-training is not automatically self-improvement. When a model is trained on data produced by earlier versions of the same kind of model, the loop can become self-reinforcing rather than corrective.

Code large language models (LLMs) Ben Allal et al. (2023); Li et al. (2023); Rozière et al. (2023); Hui et al. (2024) are likely to face the same risk in a concrete and testable form. A code model can generate programs, add them back into the training corpus, and fine-tune the next model on this synthetic code. If these programs were always correct, diverse, and well reviewed, the loop could cheaply scale code data. In practice, generated code often contains bugs, incomplete logic, repeated templates, missing edge cases, and narrow stylistic patterns. Recursive training can therefore copy these errors into later models, making them better at imitating their own code style while worse at solving programming tasks. This is the recursive self-training trap for code: more data is created, but it can carry the model's own functional errors into future training.

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.

This issue is no longer only an offline training concern. AI-generated code is rapidly entering real repositories through tools such as GitHub Copilot GitHub (2026a), Cursor Anysphere (2026), Amazon CodeWhisperer Amazon Web Services (2026), and Devin Cognition AI (2026). Studies report large productivity gains from Copilot Peng et al. (2023), and enterprise reports show that Copilot-suggested code is frequently accepted, committed, and merged into pull requests GitHub (2024). Future code LLMs may therefore be trained on repositories already containing AI-generated or AI-assisted code. If such code is collected without review, the loop becomes recursive self-training at repository scale: AI writes code, the code enters GitHub, GitHub enters the next training corpus, and the next model learns from earlier model outputs.

In real software development, code is usually filtered before entering a repository. A developer opens a pull request (PR), and the change may pass through compilation, tests, static analysis, and human review before merging. Modern code review is a lightweight but consequential quality-control process: it catches defects, transfers project knowledge, and affects downstream software quality Bacchelli & Bird (2013); Bosu et al. (2015); McIntosh et al. (2016); Sadowski et al. (2018). This PR process acts as an external quality gate: the author does not define the acceptance rule, and the rule does not become weaker when the author writes worse code. Failed builds, failing tests, or poor design can still block the PR. We call this model-independent acceptance mechanism a *human-style exogenous gate*. The experimental gates are automatic proxies, not literal manual review; they simulate aspects of the quality-feasible review target in Equation (G.13).

The AI coding era weakens this assumption. AI tools can generate PRs faster than humans can review them, while AI code review is becoming a practical engineering workflow. Recent studies examine LLM-assisted review with AI co-reviewers and interactive review assistants Aðalsteinsson et al. (2025); large-scale GitHub Actions analyses show AI review tools automatically commenting on real PRs Sun et al. (2025); and GitHub Copilot code review is now integrated into the PR workflow GitHub (2026b). At the same time, LLM-as-judge systems are known to be sensitive to evaluator design and can favor model-like outputs Zheng et al. (2023); Panickssery et al. (2024). Future repositories may therefore contain both AI-generated and AI-reviewed code. In this setting, the verifier may no longer be a human-quality external process, but another AI model, possibly close to the generator. If AI writes the code and AI reviews the code, the acceptance signal can become coupled to the same distribution being retrained. The review process may look like filtering, but a biased AI reviewer can gradually accept the generator's degraded outputs as normal.

This observation leads to the central question of this paper: **In recursive code LLM training, how do no review, human-style exogenous review, and AI-based review differ?** Review changes the training distribution: without review, the model trains on all generated outputs; with review, training is reweighted toward samples that satisfy an acceptance rule. The key question is who defines this rule. A *human-style exogenous gate* is fixed and model-independent, so it can reject degraded outputs as the generator worsens. An *AI self-gate* is coupled to the generator, so it can drift with the model and accept degraded model style as high quality. This concern is consistent with evidence that LLM reviewers still miss many human-flagged PR issues, even when they are strong code generators Kumar (2026); Tong & Zhang (2024); Zhao et al. (2025).

We study this question through recursive fine-tuning experiments that directly compare no review, human-style exogenous-gate review, and AI-self-gate review. The exogenous-gate setting uses external, model-independent checks to decide which generated code enters future training. The AI-self-gate setting uses the code LLM itself to evaluate its own generated code, so the same model both produces and filters the training data.

We validate this comparison across multiple code LLM families and benchmarks. Our primary endpoints are HumanEval+ and MBPP+ Liu et al. (2023), with HumanEval Chen et al. (2021); OpenAI (2021) and MBPP Austin et al. (2021) retained as secondary comparable benchmarks and LiveCodeBench Jain et al. (2024) as a robustness probe. Our primary experiments use SantaCoder Ben Allal et al. (2023), a 1.1B-parameter code model trained on permissively licensed code from The Stack Kocetkov et al. (2022). We further test StarCoder2 Lozhkov et al. (2024), Qwen2.5-Coder Hui et al. (2024), and Code Llama Rozière et al. (2023), to examine whether the same pattern holds across model families.

The results show a consistent below-baseline pattern across review regimes, with task-dependent reversals in which gate is best at a particular endpoint. In particular, unfiltered recursive fine-tuning

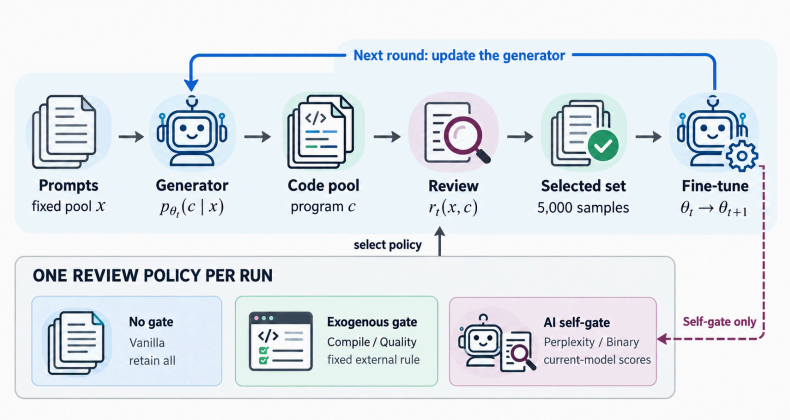

Figure 1: **Recursive training under different review policies.** Each round generates a code pool, applies one review policy, selects 5,000 samples, and fine-tunes the generator. Unlike fixed exogenous gates, AI self-gates are coupled to the current model.

degrades all tested model families; exogenous filters such as compile checks and quality rules preserve executable validity but cannot prevent long-horizon semantic drift; and AI-self-gate filters such as perplexity and binary self-scoring become less discriminative as the generator degrades. The binary classifier is the clearest AI-self-gate failure case: acceptance scores rise while benchmark correctness falls, indicating a self-confirming rubber-stamp regime.

Our contributions are summarized as follows:

1. **Code LLM self-training collapse:** We provide a systematic empirical study of recursive self-training collapse in code LLMs, covering five primary review regimes on SantaCoder, extended appendix trajectories for additional gate variants, and cross-model validation on StarCoder2, Qwen2.5-Coder, and Code Llama.
2. **Review as gated reweighting:** We formulate review-based recursive training as gated sampling, where generated samples are reweighted by an acceptance function before entering the next round. This explains why useful gates can slow collapse relative to unfiltered self-training.
3. **Exogenous gate vs. AI self-gate:** We identify exogeneity as the key difference between review regimes: human-style exogenous gates use fixed, model-independent criteria, while AI self-gates use scores coupled to the generator and can become self-confirming.
4. **Theory of collapse and gate failure:** We prove that AI self-gate training degenerates to ungated self-training under a self acceptance condition (Theorem 2.3). We also provide a spectral analysis showing representation covariance concentration under recursive retraining (Proposition 2.1).

2 Problem Formulation and Theoretical Analysis

2.1 Iterative Self-Training Without vs. With Gating

Figure 1 illustrates the recursive training pipeline considered in this paper. Let $x \in \mathcal{X}$ denote a prompt or programming task, and let $c \in \mathcal{C}$ denote a candidate code solution. At each round, the generator produces a code pool that is processed by one of three review policies: no gate, an exogenous compile/quality gate, or an AI self-gate coupled to the current model. The selected samples are then reused to fine-tune the generator for the next round.

No verification (ungated recursion). In ungated recursive self-training, every generated sample is directly reused as training data. The synthetic data distribution at iteration t is

$$m_t^{\text{ungated}}(x, c) := p_X(x) p_{\theta_t}(c \mid x). \tag{2.1}$$

The next model is obtained by maximum-likelihood estimation (**MLE**) training on this distribution:

$$\theta_{t+1}^{\text{ungated}} \in \arg\max_{\theta} \mathbb{E}_{(x,c)\sim m_t^{\text{ungated}}} \left[\log p_\theta(c \mid x) \right]. \tag{2.2}$$

Equivalently, this minimizes $\mathrm{KL}(m_t^{\text{ungated}} \| p_X p_\theta)$. Thus, ungated recursion only enforces self-consistency: the next model is trained to imitate the current model's samples. It does not impose any external constraint on correctness, compilability, or execution success.

Verification (gated recursion). In gated recursive self-training, generated code is accepted only after passing a review or verification signal. Let $\mathcal{E}(x, c) \in \{0, 1\}$ denote the acceptance event, and define

$$r(x, c) := \mathbb{P}(\mathcal{E}(x, c) = 1 \mid x, c) \in [0, 1]. \tag{2.3}$$

Here, $r(x, c)$ can represent a compile check, execution test, static quality filter, human review score, or AI-based review score. Conditioning the generator on acceptance gives the accepted distribution

$$q_{\theta_t}(c \mid x) := \mathbb{P}(c \mid x, \mathcal{E} = 1) = \frac{p_{\theta_t}(c \mid x) r(x, c)}{Z_{\theta_t}(x)}, \quad Z_{\theta_t}(x) := \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) r(x, c'). \tag{2.4}$$

The gated synthetic data distribution is therefore

$$m_t^{\text{gated}}(x, c) := p_X(x) q_{\theta_t}(c \mid x). \tag{2.5}$$

Training on accepted samples yields

$$\theta_{t+1}^{\text{gated}} \in \arg\max_{\theta} \mathbb{E}_{(x,c) \sim m_t^{\text{gated}}} \big[\log p_\theta(c \mid x) \big]. \tag{2.6}$$

Thus, gated recursion differs from ungated recursion only through the sampling measure: before training the next model, generated samples are reweighted by the acceptance signal $r(x, c)$.

Difference (see Figures 1). The two recursions differ only in the sampling measure:

$$m_t^{\text{gated}}(x, c) \propto m_t^{\text{ungated}}(x, c) \cdot r(x, c). \tag{2.7}$$

Thus, gated recursion trains on the same generated distribution as ungated recursion, but reweights samples by acceptance: high-$r(x, c)$ code receives more training mass, while low-$r(x, c)$ code is suppressed. At a fixed point of the gated recursion (pure self-training), one must satisfy

$$p_{\theta^*}(c \mid x) = q_{\theta^*}(c \mid x) \iff p_{\theta^*}(c \mid x) \propto p_{\theta^*}(c \mid x) r(x, c), \tag{2.8}$$

which implies that (for each x) probability mass concentrates on the support where $r(x, c)$ is maximal (e.g., $r(x, c) = 1$ if such code exists). In contrast, the ungated recursion enforces only self-reproduction of its own samples and does not bias toward executability. This is why a useful exogenous gate can slow collapse, while a gate that becomes constant on the generator's support loses its filtering effect and approaches ungated self-training.

2.2 Representation-Subspace Drift Under Recursive Self-Training

We next describe a representation-level view of recursive self-training collapse. Let $\mathcal{D}_t$ denote the training distribution at iteration t over prompt–code pairs $(x, c) \in \mathcal{X} \times \mathcal{C}$, where $\mathcal{D}_t$ may be ungated or gated. Fix a representation map

$$\phi : \mathcal{X} \times \mathcal{C} \to \mathbb{R}^d, \qquad z := \phi(x, c). \tag{2.9}$$

The distribution $\mathcal{D}_t$ induces a mean vector and covariance matrix in representation space:

$$\mu_t := \mathbb{E}_{(x,c) \sim \mathcal{D}_t}[z], \qquad \Sigma_t := \mathbb{E}_{(x,c) \sim \mathcal{D}_t}\big[(z - \mu_t)(z - \mu_t)^\top\big] \in \mathbb{R}^{d \times d}. \tag{2.10}$$

The covariance Σ_t captures which representation directions remain diverse under the current training distribution. If recursive self-training narrows the generated code distribution, then Σ_t should concentrate on fewer dominant directions.

Let $U_t \in \mathbb{R}^{d \times k}$ contain the top-k eigenvectors of Σ_t, and let $P_t := U_t U_t^\top$ be the corresponding rank-k projector. We measure consecutive subspace drift by

$$\text{Sim}(t, t+1) := \frac{1}{k} \text{Tr}(P_t P_{t+1}) \in [0, 1], \quad \text{Dist}(t, t+1) := \|P_t - P_{t+1}\|_2 = \sin \theta_{\max}(U_t, U_{t+1}). \tag{2.11}$$

Here, high similarity means the dominant representation subspace is stable, while large distance indicates that recursive training has shifted the main directions of variation.

To analyze the mechanism, we use a linearized covariance recursion:

$$\Sigma_{t+1} = A \Sigma_t A^\top, \qquad \Sigma_0 \succeq 0. \tag{2.12}$$

This recursion should be read as a local approximation of how one round of retraining transforms representation covariance. Let $A = U\mathrm{diag}(s_1, \ldots, s_d)V^\top$ be an SVD with $s_1 \geq s_2 \geq \cdots \geq s_d \geq 0$. Then $\Sigma_t = A^t \Sigma_0 (A^\top)^t$ and therefore

$$\|\Sigma_t\|_2 \leq \|A^t\|_2^2 \|\Sigma_0\|_2 = s_1^{2t} \|\Sigma_0\|_2. \tag{2.13}$$

More importantly, different singular directions are amplified at different rates. For any unit vector v,

$$v^\top \Sigma_t v = \|(A^\top)^t v\|_{\Sigma_0}^2, \qquad \|(A^\top)^t v\| \text{ scales as } s_i^t \text{ when } v = v_i. \tag{2.14}$$

Thus, if $s_1 \gg s_2$, the component aligned with the leading singular direction dominates after repeated retraining. Whenever Σ_0 has nonzero energy in this direction, the ratio $\lambda_1(\Sigma_t)/\lambda_2(\Sigma_t)$ is expected to increase with t because the leading component is amplified roughly by s_1^{2t} while the next component is amplified roughly by s_2^{2t}. This gives a representation-space explanation for collapse: recursive self-training can progressively concentrate variance into a low-dimensional subspace, reducing diversity in the generated code distribution.

We now formalize the concentration effect by the linearized covariance recursion in Equation (2.12).

Proposition 2.1 (Spectral concentration)**.** *Under the covariance recursion in Equation* (2.12)*, recall that $\Sigma_t = A^t \Sigma_0 (A^\top)^t$. Let $A = U\mathrm{diag}(s_1, \ldots, s_d)V^\top$ be the SVD of A, with $s_1 > s_2 \geq \cdots \geq s_d \geq 0$, and define $\widetilde{\Sigma}_0 := V^\top \Sigma_0 V \succeq 0$. Then $\Sigma_t = U\mathrm{diag}(s_1^t, \ldots, s_d^t)\widetilde{\Sigma}_0 \mathrm{diag}(s_1^t, \ldots, s_d^t)U^\top$. If $(\widetilde{\Sigma}_0)_{11} = v_1^\top \Sigma_0 v_1 \geq \alpha > 0$ and $\rho_2 := s_2/s_1 < 1$, then*

$$\frac{\lambda_1(\Sigma_t)}{\lambda_2(\Sigma_t)} \geq \frac{\alpha}{\lambda_2(\widetilde{\Sigma}_0) + \|\Sigma_0\|_2 \rho_2^{2t}}. \tag{2.15}$$

Consequently, if $\lambda_2(\widetilde{\Sigma}_0) > 0$, then $\lambda_1(\Sigma_t)/\lambda_2(\Sigma_t) \to \alpha/\lambda_2(\widetilde{\Sigma}_0)$ as $t \to \infty$; if $\lambda_2(\widetilde{\Sigma}_0) = 0$, the ratio grows without bound.

The proof is given in Appendix F. Proposition 2.1 shows that recursive retraining can amplify the dominant representation direction faster than all others, at a relative rate controlled by ρ_2^{2t}. This gives a representation-level mechanism for mode concentration: the model distribution can become increasingly low-rank even while training continues to optimize self-generated samples.

2.3 Human Gate vs. AI Self-Gate

2.3.1 Verifier parameterization: exogenous Human gate vs. endogenous AI self-gate

We now specialize the generic acceptance function $r(x, c)$ from Equation (2.3). The key is whether the acceptance rule is fixed outside the generator or coupled to the generator during recursive training.

Human gate: exogenous acceptance. A Human gate uses a fixed acceptance function $r_H(x, c) \in [0, 1]$, r_H is independent of t and θ_t. It induces the accepted conditional distribution

$$q_{\theta_t}^H(c \mid x) := \frac{p_{\theta_t}(c \mid x) r_H(x, c)}{Z_{\theta_t}^H(x)}, \quad Z_{\theta_t}^H(x) := \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) r_H(x, c'). \tag{2.16}$$

The corresponding training distribution is $m_t^H(x, c) = p_X(x) q_{\theta_t}^H(c \mid x)$. This covers model-independent review signals such as compilation, execution, quality checks, or human PR review.

AI self-gate: endogenous acceptance. An AI self-gate uses a learned reviewer with parameters ϕ_t, producing $r_{\phi_t}(x, c) \in [0, 1]$. It induces

$$q_{\theta_t, \phi_t}^A(c \mid x) := \frac{p_{\theta_t}(c \mid x) r_{\phi_t}(x, c)}{Z_{\theta_t, \phi_t}^A(x)}, \quad Z_{\theta_t, \phi_t}^A(x) := \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) r_{\phi_t}(x, c'). \tag{2.17}$$

The corresponding training distribution is $m_t^A(x, c) = p_X(x) q_{\theta_t, \phi_t}^A(c \mid x)$. In our experiments, this case corresponds to perplexity filtering and binary self-scoring, where the code LLM evaluates its own generated samples. Unlike r_H, the score r_{ϕ_t} can drift as the generator changes.

Taxonomy summary. Table 1 classifies the six filtering strategies by whether the acceptance rule is exogenous or endogenous. Vanilla has no gate and accepts all generated code. Compile is a Human gate because it directly runs Python compilation and accepts code only when compilation succeeds. Quality is also a Human gate because it uses fixed code-quality rules, described in Section G, such as repetition rate and length. Compile+Quality combines these two model-independent rules, so it is also exogenous. Perplexity and Binary Classifier are AI self-gates: both use the code LLM itself to score its own generated code. Perplexity filters by the model's own likelihood, while Binary Classifier uses the model's own logit-difference score, defined in Section H. Since the generator also provides the filtering signal, these two gates are coupled to θ_t.

Table 1: Human gate vs. AI self-gate taxonomy of all filtering strategies studied.

Method	Gate type	Acceptance signal	Coupled to θ_t?
Vanilla	– (no gate)	Accept all	No
Compile	Human gate	Compilability (`compile()`)	No
Quality	Human gate	Repetition rate + length (no compile)	No
Compile+Quality	Human gate	Compile + repetition/length filters	No
Perplexity	AI self-gate	Model perplexity (lowest 25%)	Yes
Binary Classifier	AI self-gate	Logit-difference score (top 25%)	Yes

Coupled dynamics under AI self-gating. We now formalize the failure mode where the verifier is coupled to the generator. This captures the AI self-gate setting: the code LLM generates candidate code, scores its own outputs, and then trains on the samples it accepted. For a clean statement, we analyze the extreme case where the generator and verifier share the same parameters:

$$\phi_t = \theta_t, \qquad r_{\phi_t}(x, c) = r_{\theta_t}(x, c). \tag{2.18}$$

Assumption 2.2 (Self-confirming acceptance)**.** There exists a measurable function $\kappa : \mathcal{X} \to (0, 1]$ such that for p_X-almost every x, $r_{\theta_t}(x, c) = \kappa(x)$ for $p_{\theta_t}(\cdot \mid x)$-almost every c.

Assumption 2.2 describes the rubber-stamp regime: on the code that the current model is likely to generate, the AI reviewer assigns essentially the same acceptance score. Once this happens, the gate no longer changes the training distribution.

Theorem 2.3 (Degeneracy to ungated recursion)**.** *Assume the shared-parameter self-gate in Equation* (2.18) *and Assumption 2.2. Then for p_X-almost every x,*

$$Z^A_{\theta_t,\theta_t}(x) := \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) r_{\theta_t}(x, c') = \kappa(x), \tag{2.19}$$

and for $p_{\theta_t}(\cdot \mid x)$-almost every c,

$$q^A_{\theta_t,\theta_t}(c \mid x) := \frac{p_{\theta_t}(c \mid x) r_{\theta_t}(x, c)}{Z^A_{\theta_t,\theta_t}(x)} = p_{\theta_t}(c \mid x). \tag{2.20}$$

Consequently,

$$m^A_t(x, c) := p_X(x) q^A_{\theta_t,\theta_t}(c \mid x) = p_X(x) p_{\theta_t}(c \mid x) = m^{\text{ungated}}_t(x, c) \qquad \textit{a.e.}, \tag{2.21}$$

*and the gated **MLE** update coincides with the ungated self-training update:*

$$\arg\max_\theta \mathbb{E}_{(x,c)\sim m^A_t}[\log p_\theta(c \mid x)] = \arg\max_\theta \mathbb{E}_{(x,c)\sim m^{\text{ungated}}_t}[\log p_\theta(c \mid x)]. \tag{2.22}$$

The proof is given in Appendix F. Theorem 2.3 states that once an AI self-gate becomes constant on the generator's own support, filtering becomes mathematically identical to no filtering. This is exactly the recursive self-training failure case where the model writes code, approves its own code, and then trains on that approved code.

Corollary 2.4 (Collapsed fixed point)**.** *If there exists $(\theta^\star, \phi^\star)$ such that $r_{\phi^\star}(x, c) \equiv 1$ for all (x, c), then for all x,*

$$q^A_{\theta^\star,\phi^\star}(c \mid x) = p_{\theta^\star}(c \mid x), \qquad m^A(x, c) = p_X(x) p_{\theta^\star}(c \mid x). \tag{2.23}$$

Thus, the gating operator is the identity on $p_{\theta^\star}(\cdot \mid x)$.

The proof is given in Appendix F.

2.4 When AI review matches human review.

The self-confirming case above characterizes when AI review degenerates to ungated recursion. Conversely, AI-gated training remains close to exogenous human-gated training when the AI reviewer stays uniformly calibrated to the human reviewer and the human gate retains non-vanishing acceptance mass.

Lemma 2.5 (Approximation of human-gated review)**.** *Define* $\varepsilon_t := \sup_{x\in\mathcal{X},c\in\mathcal{C}} \left|r_{\phi_t}(x,c) - r_H(x,c)\right|$. *Suppose* $Z^H_{\theta_t}(x) \geq z_0 > 0$ *for* p_X*-almost every* x. *If* $\varepsilon_t < z_0$, *then*

$$\left\|q^A_{\theta_t,\phi_t}(\cdot \mid x) - q^H_{\theta_t}(\cdot \mid x)\right\|_1 \leq \frac{2\varepsilon_t}{z_0}, \tag{2.24}$$

and consequently $\|m^A_t - m^H_t\|_1 \leq 2\varepsilon_t/z_0$.

Lemma 2.5 shows that calibration to an external review signal controls the deviation of AI-gated recursion from human-gated recursion. The full formulation, sufficient conditions for $z_0 > 0$, and proof are given in Appendix D.

2.5 Quality Gate

The ideal gate would accept exactly the programs in the *quality gate set* $\mathcal{F}_{\mathrm{qual}}(q)$, defined by quality dimensions (A)–(G) in Appendix G. Our Human-gate methods approximate this ideal using fixed, model-independent checks: compilation for basic runtime validity and quality rules for repetition and length. In contrast, AI self-gates rely on model-derived scores that can drift with the generator and lose alignment with $\mathcal{F}_{\mathrm{qual}}(q)$, as characterized in Theorem 2.3.

3 Experiments

3.1 Evaluation Setup

Evaluation setup. We use **HumanEval+** and **MBPP+** Liu et al. (2023) as the primary endpoints, with **HumanEval** Chen et al. (2021); OpenAI (2021) and **MBPP** Austin et al. (2021) as secondary benchmarks and **LiveCodeBench** Jain et al. (2024) as a robustness probe. We evaluate **SantaCoder** (1.1B) Ben Allal et al. (2023), **StarCoder2-3B** Lozhkov et al. (2024), **Qwen2.5-Coder-1.5B** Hui et al. (2024), and **Code Llama-7B** Rozière et al. (2023) for five recursive rounds. All evaluations use `bigcode-evaluation-harness` Ben Allal et al. (2022) with temperature 0.8, top-p 0.95, and $n = 200$ candidates per problem for pass@1.

Recursive training. At each round, the current model generates code from a fixed prompt pool, the review policy selects $5{,}000$ samples, and the next model is fine-tuned for 3,000 steps. Exogenous gates generate until $5{,}000$ samples are accepted, whereas AI self-gates generate 20k candidates and retain 25%. SantaCoder uses pure-code fine-tuning, while StarCoder2 and Qwen2.5-Coder use approximately 70–80% code and 20–30% code-related natural language. All methods otherwise share the same training configuration. Because the sampling procedures differ, we separately report raw gate pass rate, accepted-set execution validity, and candidates required per 5k accepted. Full details are in Appendix N.

Filtering strategies. We evaluate five primary regimes (Table 1). **No gate:** (1) **Vanilla** accepts every generated sample. **Exogenous gates:** (2) **Compile** accepts code that passes Python `compile()`; (3) **Quality** applies fixed model-independent criteria, requiring repetition rate ≤ 0.3 and length ≥ 50 tokens. **AI self-gates:** (4) **Perplexity** generates 20k candidates and retains the lowest 25% by masked-completion perplexity; (5) **Binary Classifier** retains the top 25% by the model's logit-difference score between `good` and `bad`. Thus, the regimes differ in whether the training data are unfiltered, filtered by a fixed external rule, or filtered by a score coupled to the recursively trained model. Extended Compile+Quality and Quality-20R variants are reported in Appendix L.

3.2 Main Collapse Results

Cross-model collapse patterns. Figure 2 summarizes the recursive trajectories across all four model families and the two gated review categories. Despite substantial differences in initial performance, every model moves below its own Round 0 baseline over five recursive rounds on both HumanEval+ and MBPP+. The figure also shows that gating changes the shape of degradation, but neither exogenous nor AI self-gating consistently preserves the initial performance level.

Table 2: Cross-model Round 0 and Round 5 performance. For each model, we report Vanilla recursion and the strongest exogenous (EG) and AI self-gate according to R5 HumanEval+. **Bold** indicates the strongest reported R5 value within each model and metric.

Model	Stage	HumanEval	HumanEval+	MBPP	MBPP+	LCB
SantaCoder (1.1B)	Baseline	0.189	0.171	0.352	0.294	0.018
StarCoder2-3B	Baseline	0.317	0.274	0.595	0.492	0.000
Qwen2.5-Coder-1.5B	Baseline	0.451	0.372	0.698	0.582	0.238
Code Llama-7B	Baseline	0.328	0.287	0.441	0.388	0.045
SantaCoder (1.1B)	R5 Vanilla	0.079	0.073	0.027	0.019	0.000
SantaCoder (1.1B)	R5 Compile (EG)	0.092	0.079	0.045	0.034	0.000
SantaCoder (1.1B)	R5 Binary (AI)	**0.116**	**0.104**	**0.079**	**0.058**	0.000
StarCoder2-3B	R5 Vanilla	0.128	0.104	**0.185**	**0.172**	0.000
StarCoder2-3B	R5 Quality (EG)	0.104	0.092	0.106	0.098	**0.003**
StarCoder2-3B	R5 Binary (AI)	**0.140**	**0.122**	0.090	0.066	0.000
Qwen2.5-Coder-1.5B	R5 Vanilla	0.049	0.043	0.085	0.082	0.000
Qwen2.5-Coder-1.5B	R5 Compile (EG)	0.104	0.098	**0.148**	**0.127**	**0.003**
Qwen2.5-Coder-1.5B	R5 Binary (AI)	**0.146**	**0.122**	0.082	0.071	0.000
Code Llama-7B	R5 Vanilla	0.177	0.134	**0.235**	**0.214**	**0.003**
Code Llama-7B	R5 Compile (EG)	**0.189**	**0.171**	0.124	0.090	**0.003**
Code Llama-7B	R5 PPL (AI)	0.165	0.134	0.206	0.175	0.000

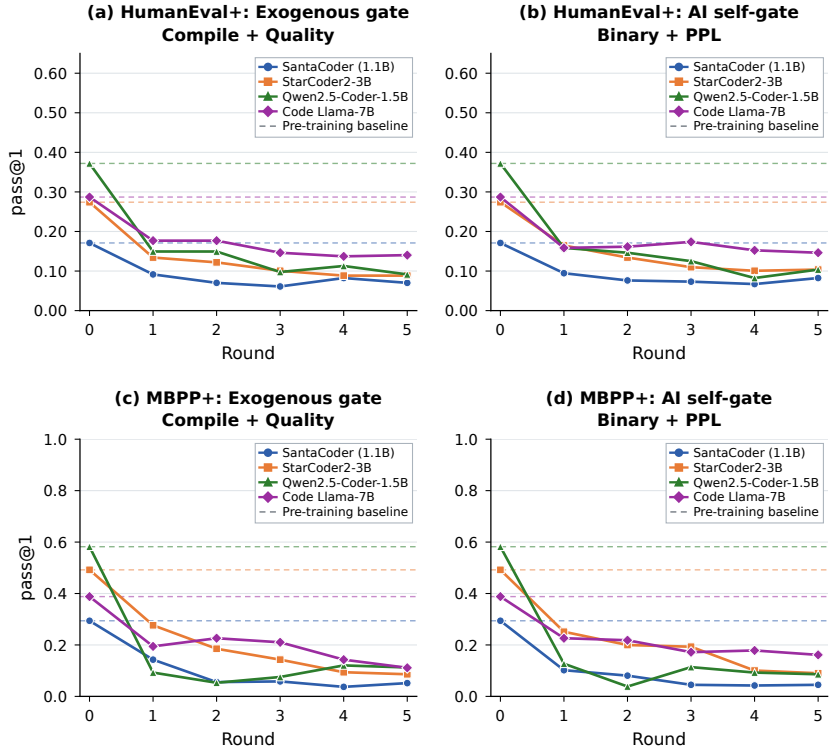

Figure 2: **Cross-model recursive degradation under different review policies.** Trajectories from Round 0 to Round 5 for all four models. Left: exogenous gates (Compile + Quality average); right: AI self-gates (Binary + PPL average). Top: HumanEval+; bottom: MBPP+. Dashed horizontal lines denote model-specific Round 0 baselines.

Different models collapse at different rates. The degradation rate varies across models and benchmarks. Qwen2.5-Coder and SantaCoder show particularly sharp declines in some settings, while Code Llama is relatively more stable. This variation suggests that recursive robustness is not determined by initial model capability alone and depends on the model–benchmark combination.

Review policies change the endpoint. Table 2 shows that no review policy consistently performs best across models and benchmarks. Different gates improve Round 5 retention in different settings,

but all reported strategies remain below their corresponding Round 0 baselines. This indicates that gating primarily changes the severity of degradation rather than preventing recursive collapse.

3.3 Effect of Review Gates

Execution-backed review. Table 3 examines the samples that enter recursive training. On Qwen2.5-Coder-1.5B at Round 5, Compile raises accepted-set execution validity from 33.4% to 89.8%, whereas the AI self-gate retains 32.0%. Thus, exogenous execution filtering substantially changes the accepted training distribution, but does not eliminate downstream degradation.

Table 3: Execution-backed review on Qwen2.5-Coder-1.5B at Round 5.

Gate	Raw gate pass	Accepted execution	Candidates / 5k	HumanEval S_5/S_0	MBPP S_5/S_0	$\Delta_{0\to5}$
No review	1.00	33.4%	5,000	0.108	0.121	−0.150
Compile only	0.898	89.8%	5,568	0.230	0.212	−0.109
AI self-gate	0.25	32.0%	20,000	0.324	0.117	−0.126

3.4 Filter Comparison at Round 1 and Round 5

Figure 3 compares HumanEval+ performance across filters at Round 1 and Round 5.

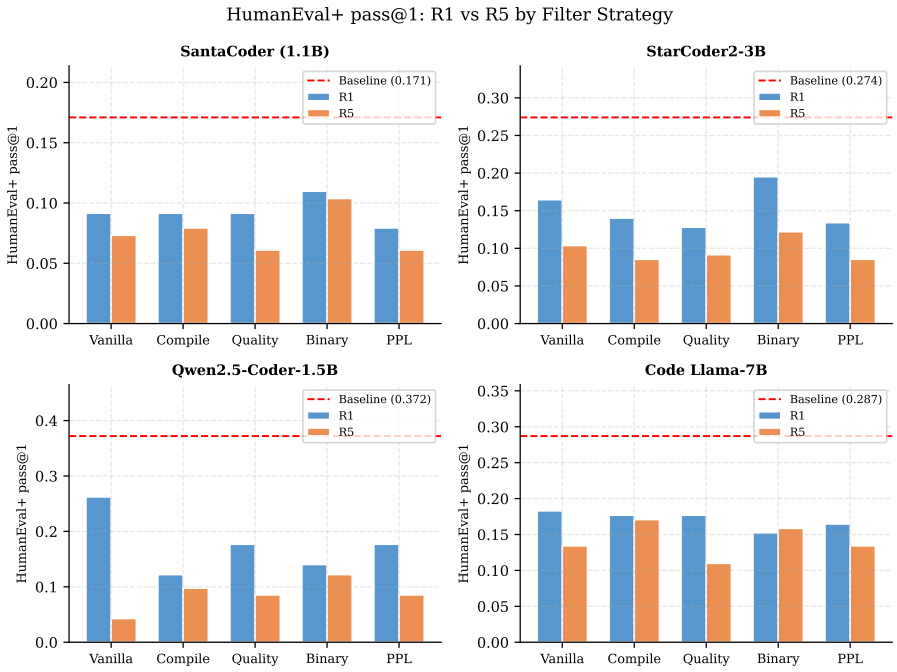

Figure 3: **HumanEval+ pass@1 at Round 1 and Round 5 across filtering strategies.** Blue bars denote R1, orange bars denote R5, and red dashed lines denote pre-training baselines.

Binary achieves the highest R5 HumanEval+ for SantaCoder, StarCoder2-3B, and Qwen2.5-Coder, while Compile is strongest for Code Llama-7B. However, all R5 results remain below their pre-training baselines, indicating that filter choice changes the degradation trajectory but does not eliminate recursive collapse.

4 Conclusion

We studied recursive self-training collapse in code LLMs under no review, exogenous-gate review, and AI-self-gate review. Across models and benchmarks, all tested regimes fall below their starting baselines. Exogenous gates preserve external validity signals but do not prevent degradation, while AI self-gates remain coupled to the evolving generator and can become self-confirming. Gating nevertheless changes both the accepted training distribution and the resulting degradation trajectory. Our theory characterizes when model-coupled review degenerates toward ungated recursion. Our results suggest that stable recursive training requires verification signals that remain externally grounded.

AI USE STATEMENT

Generative AI tools were used to assist with LaTeX formatting, template conversion, and the organization of manuscript material between the main text and appendix.

REFERENCES

Sina Alemohammad, Josue Casco-Rodriguez, Lorenzo Luzi, Ahmed Imtiaz Humayun, Hossein Babaei, Daniel LeJeune, Ali Siahkoohi, and Richard G. Baraniuk. Self-consuming generative models go mad, 2023. URL https://arxiv.org/abs/2307.01850.

Amazon Web Services. Amazon CodeWhisperer is becoming part of Amazon Q Developer. https://docs.aws.amazon.com/codewhisperer/latest/userguide/whisper-legacy.html, 2026. Accessed 2026-05-07.

Anysphere. Cursor: The best way to code with ai. https://cursor.com/, 2026. Accessed 2026-05-07.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Fannar Steinn Aðalsteinsson, Björn Borgar Magnússon, Mislav Milicevic, Adam Nirving Davidsson, and Chih-Hong Cheng. Rethinking code review workflows with llm assistance: An empirical study, 2025. URL https://arxiv.org/abs/2505.16339.

Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 712–721, 2013. doi: 10.1109/ICSE.2013.6606617. URL https://doi.org/10.1109/ICSE.2013.6606617.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Loubna Ben Allal, Niklas Muennighoff, Logesh Kumar Umapathi, Ben Lipkin, Leandro von Werra, et al. bigcode-project/bigcode-evaluation-harness: A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022. Accessed 2026-02-21.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, et al. Santacoder: Don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 146–156, 2015. doi: 10.1109/MSR.2015.21. URL https://doi.org/10.1109/MSR.2015.21.

Martin Briesch, Dominik Sobania, and Franz Rothlauf. Large language models suffer from their own output: An analysis of the self-consuming training loop, 2023. URL https://arxiv.org/abs/2311.16822.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*, 2022.

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

Cognition AI. Devin: The ai software engineer. https://devin.ai/, 2026. Accessed 2026-05-07.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Elvis Dohmatob, Yunzhen Feng, Pu Yang, François Charton, and Julia Kempe. A tale of tails: Model collapse as a change of scaling laws. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. doi: 10.48550/arXiv.2402.07043. URL https://arxiv.org/abs/2402.07043.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Damien Ferbach, Quentin Bertrand, Avishek Joey Bose, and Gauthier Gidel. Self-consuming generative models with curated data provably optimize human preferences. In *Advances in Neural Information Processing Systems*, 2024. doi: 10.48550/arXiv.2407.09499. URL https://arxiv.org/abs/2407.09499.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Shi Fu, Yingjie Wang, Yuzhu Chen, Xinmei Tian, and Dacheng Tao. A theoretical perspective: How to prevent model collapse in self-consuming training loops. In *International Conference on Learning Representations*, 2025. doi: 10.48550/arXiv.2502.18865. URL https://arxiv.org/abs/2502.18865.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. RLEF: Grounding code LLMs in execution feedback with reinforcement learning. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. doi: 10.48550/arXiv.2410.02089. URL https://arxiv.org/abs/2410.02089.

Nate Gillman, Michael Freeman, Daksh Aggarwal, Chia-Hong Hsu, Calvin Luo, Yonglong Tian, and Chen Sun. Self-correcting self-consuming loops for generative model training. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. doi: 10.48550/arXiv.2402.07087. URL https://arxiv.org/abs/2402.07087.

GitHub. Research: Quantifying github copilot's impact in the enterprise with accenture. https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-in-the-enterprise-with-accenture/, 2024. Accessed 2026-05-06.

GitHub. GitHub Copilot: Your ai pair programmer. https://github.com/features/copilot, 2026a. Accessed 2026-05-07.

GitHub. 60 million copilot code reviews and counting. https://github.blog/ai-and-ml/github-copilot/60-million-copilot-code-reviews-and-counting/, 2026b. Accessed 2026-05-06.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need, 2023. URL https://arxiv.org/abs/2306.11644.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Helia Hashemi, Jason Eisner, Corby Rosset, Benjamin Van Durme, and Chris Kedzie. LLM-rubric: A multidimensional, calibrated approach to automated evaluation of natural language texts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13806–13834. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.745. URL https://aclanthology.org/2024.acl-long.745/.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*, 2024.

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, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The Stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533*, 2022.

Deepak Kumar. Swe-prbench: Benchmarking ai code review quality against pull request feedback. *arXiv preprint arXiv:2603.26130*, 2026.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022. URL https://arxiv.org/abs/2207.01780.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, et al. Starcoder: May the source be with you! *Transactions on Machine Learning Research*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378:1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/10.1126/science.abq1158.

Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81, Barcelona, Spain, 2004. Association for Computational Linguistics. URL https://aclanthology.org/W04-1013/.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2305.01210.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with evol-instruct. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21 (5):2146–2189, 2016. doi: 10.1007/s10664-015-9381-9. URL https://doi.org/10.1007/s10664-015-9381-9.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

OpenAI. openai/human-eval: Code for the paper "evaluating large language models trained on code". https://github.com/openai/human-eval, 2021. Accessed 2026-02-21.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=4NJBV6Wp0h.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040/.

Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. Benchmarks and metrics for evaluations of code generation: A critical review. *arXiv preprint arXiv:2406.12655*, 2024.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pp. 392–395, Lisbon, Portugal, 2015. Association for Computational Linguistics. doi: 10.18653/v1/W15-3049. URL https://aclanthology.org/W15-3049/.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Caitlin Sadowski, Emma Soderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 181–190, 2018. doi: 10.1145/3183519.3183525. URL https://doi.org/10.1145/3183519.3183525.

Mohamed El Amine Seddik, Suei-Wen Chen, Soufiane Hayou, Pierre Youssef, and Merouane Debbah. How bad is training on synthetic data? A statistical analysis of language model collapse, 2024. URL https://arxiv.org/abs/2404.05090.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. The curse of recursion: Training on generated data makes models forget, 2024. URL https://arxiv.org/abs/2305.17493.

Kexin Sun, Hongyu Kuang, Sebastian Baltes, Xin Zhou, He Zhang, Xiaoxing Ma, Guoping Rong, Dong Shao, and Christoph Treude. Does ai code review lead to code changes? a case study of github actions. *arXiv preprint arXiv:2508.18771*, 2025.

Ananda Suresh, Andrew Thangaraj, and Aditya Nanda Kishore Khandavally. Rate of model collapse in recursive training. In *Proceedings of the 28th International Conference on Artificial Intelligence and Statistics*, 2025. doi: 10.48550/arXiv.2412.17646. URL https://arxiv.org/abs/2412.17646.

Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 20032–20051. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.1118. URL https://aclanthology.org/2024.emnlp-main.1118/.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 1069–1088. Association for Computational Linguistics, 2023. doi: 10.48550/arXiv.2305.07922. URL https://arxiv.org/abs/2305.07922.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. SelfCodeAlign: Self-alignment for code generation. In *Advances in Neural Information Processing Systems*, volume 37, 2024a. doi: 10.48550/arXiv.2410.24198. URL https://arxiv.org/abs/2410.24198.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-Instruct. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 52632–52657, 2024b. URL https://arxiv.org/abs/2312.02120.

Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. Codejudge-eval: Can large language models be good judges in code understanding? In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 73–95. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.7/.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *NeurIPS 2023 Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=uccHPGDlao.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. OpenCodeInterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 12834–12859. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.762. URL https://aclanthology.org/2024.findings-acl.762/.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Hu, Runxin Xu, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

Xuekai Zhu, Daixuan Cheng, Hengli Li, Kaiyan Zhang, Ermo Hua, Xingtai Lv, et al. How to synthesize text data without model collapse? In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. doi: 10.48550/arXiv.2412.14689. URL https://arxiv.org/abs/2412.14689.

A Related Work

A.1 Model Collapse

The systematic study of model collapse begins with Shumailov et al. Shumailov et al. (2024), who demonstrate that training on recursively generated data causes (1) early collapse: distributional errors accumulate and the model drifts from the true distribution; and (2) late collapse: low-frequency events permanently disappear. Their experiments on VAE, GMM, and OPT confirm that fresh human data must be periodically injected to prevent collapse. Alemohammad et al. Alemohammad et al. (2023) study self-consuming image generation models, proving that pure synthetic training loops monotonically shrink distribution variance (MAD: Model Amplification Disorder). They identify three loop types: fully synthetic, augmented (partial human mixing), and accumulating (fixed replay); only augmented and accumulating loops with sufficient human fraction avoid collapse.

Briesch et al. Briesch et al. (2023) provide the first LLM-specific self-consuming study using logically verifiable outputs (logical expressions), finding that synthetic data fraction is the primary collapse predictor. Seddik et al. Seddik et al. (2024) provide a statistical analysis showing that pure synthetic recursion leads to collapse, while sufficiently mixing real data can stabilize the process. Dohmatob et al. Dohmatob et al. (2024) reframe collapse as a change in scaling laws—synthetic training truncates the power-law tail of the data distribution, eliminating rare events. Suresh et al. Suresh et al. (2025) prove that preserving a fraction $\epsilon > 0$ of real data per generation keeps the KL divergence from the true distribution bounded, while $\epsilon \to 0$ (pure synthetic) leads to unbounded divergence—formalizing the role of exogenous quality signal.

On the constructive side, Ferbach et al. Ferbach et al. (2024) show that curation according to a reward model can turn iterative retraining into an implicit preference-optimization mechanism. This directly justifies our exogenous-gate framework: a compile gate or execution gate aligns with ground-truth correctness; AI self-gate does not. Gillman et al. Gillman et al. (2024) propose a correction function that maps synthetic samples toward higher-probability regions of the true distribution, stabilizing loops; code execution is the natural correction function in the code domain. Zhu et al. Zhu et al. (2025) demonstrate token-level diversity perturbations as a mitigation strategy. Fu et al. Fu et al. (2025) introduce recursive stability conditions for Transformer in-context learning.

A.2 Modern Code Review and Exogenous Verification

Modern code review provides the software-engineering analogue of the exogenous gate in our framework. Bacchelli and Bird Bacchelli & Bird (2013) show that defect finding is only one part of review: reviewers also enforce design norms, share knowledge, and improve maintainability. Bosu et al. Bosu et al. (2015) study which review comments developers find useful, while McIntosh et al. McIntosh et al. (2016) link code-review coverage, participation, and reviewer expertise to post-release software quality. At industrial scale, Sadowski et al. Sadowski et al. (2018) describe Google's tool-based review process over millions of changes. These studies motivate our abstraction of review as an exogenous acceptance rule: the important property is not that every decision is manual, but that acceptance is not produced by the same model distribution being retrained.

A.3 Code LLM Development

Representation era (2020–2021). CodeBERT Feng et al. (2020) established the bimodal NL-PL representation paradigm (encoder-only, 6 languages, MLM + RTD). Codex Chen et al. (2021) introduced generative code LLMs (12B, fine-tuned GPT on GitHub) and the pass@k evaluation standard. AlphaCode Li et al. (2022) scaled to competition-level difficulty at 41B parameters, using massive sampling (10^3 candidates) plus execution-based filtering—an early exogenous gate.

Fill-in-the-middle and open-source (2022–2023). InCoder Fried et al. (2022) unified generation and infilling via causal masking. CodeGen Nijkamp et al. (2022) demonstrated multi-turn multi-lingual generation at 16B. The Stack Kocetkov et al. (2022) established 3TB permissive-license code as the community standard. SantaCoder Ben Allal et al. (2023) showed that 1.1B models with high-quality filtering can match much larger counterparts; this is our primary base model. StarCoder Li et al. (2023) (15.5B) set the open-source SOTA on HumanEval. CodeT5+ Wang et al. (2023) demonstrated flexible encoder-decoder pretraining objectives.

Data-quality and instruction-tuning era (2023–2024). WizardCoder Luo et al. (2024) applied Evol-Instruct (AI-generated instruction evolution) to code, reaching HumanEval 57.3%; this is an early AI-assisted data augmentation pipeline. Gunasekar et al. Gunasekar et al. (2023) showed that 6B tokens of GPT-3.5 synthetic data can train a 1.3B model to HumanEval 50.6% (phi-1)—one-shot synthetic, not recursive. Magicoder Wei et al. (2024b) introduced OSS-Instruct: seed real code snippets → GPT-generated problems → 7B model at HumanEval 70.7%. StarCoder2 Lozhkov et al. (2024) and The Stack v2 extended to 900+ languages with improved governance. CodeRL Le et al. (2022) uses unit-test feedback and a learned critic for program synthesis, showing the value of grounding code generation in functional signals. SelfCodeAlign Wei et al. (2024a) introduced the first execution-validated self-alignment pipeline; its execution gate corresponds to our exogenous-gate archetype. Code Llama Rozière et al. (2023) (7B, 13B, 34B) fine-tuned Llama 2 on code with a mixed code-language pretraining data mixture.

State-of-the-art (2024–2025). DeepSeek-Coder Guo et al. (2024) (1B–33B) achieves HumanEval 79.3% with repository-level training. DeepSeek-Coder-V2 Zhu et al. (2024) (MoE 236B/21B active) matches GPT-4-Turbo code ability. Qwen2.5-Coder Hui et al. (2024) (0.5B–32B) reports open-source SOTA code performance, with the 32B-Instruct model matching GPT-4o-level coding ability. OpenCoder Huang et al. (2024) provides fully open recipe. OpenCodeInterpreter Zheng et al. (2024) integrates code generation with execution and refinement, further emphasizing executable feedback as an external training signal. RLEF Gehring et al. (2025) grounds code LLMs in execution feedback with RL, providing complementary evidence for execution-backed training signals. DeepSeek-R1 DeepSeek-AI et al. (2025) demonstrates GRPO-based reasoning scaling with rule-based verifiable rewards, reaching top-96th-percentile on Codeforces.

B Evaluation Benchmarks: Full Description

HumanEval. Each problem consists of a function signature, docstring, and 7–9 unit tests.

MBPP. The Mostly Basic Programming Problems dataset Austin et al. (2021) contains 374 crowd-sourced Python problems focusing on introductory-level tasks (string manipulation, arithmetic, sorting).

LiveCodeBench. LiveCodeBench Jain et al. (2024) continuously collects new programming problems from LeetCode, AtCoder, and Codeforces, preventing contamination from models that have been trained on earlier problems. In our experiments, SantaCoder achieves near-zero pass@1 on LiveCodeBench throughout retraining, reflecting the gap between introductory-level proficiency and competition-level reasoning.

Code generation metrics. Table 4 summarizes common metrics for evaluating code generation quality.

SantaCoder: Open code LMs with governance and infilling-capable training. The BigCode technical report on SantaCoder describes the development of 1.1B-parameter code LMs trained on permissively licensed data from The Stack with an emphasis on transparency and responsible release Ben Allal et al. (2023); Kocetkov et al. (2022). The report studies deployment-relevant modeling choices, notably Multi-Query Attention (MQA) for faster decoder inference Shazeer (2019) and Fill-in-the-Middle (FIM) objectives that enable arbitrary-position code infilling Bavarian et al. (2022); Fried et al. (2022). Evaluation is conducted using MultiPL-E, which extends unit-test-driven code generation benchmarks to many languages Cassano et al. (2022).

StarCoder: Open-access code LMs with long context, infilling, and responsible release. StarCoder and StarCoderBase are 15.5B-parameter open-access code LMs trained primarily on permissively licensed data from The Stack Li et al. (2023); Kocetkov et al. (2022). The models combine an 8K context window, Fill-in-the-Middle (FIM) training for infilling, and Multi-Query Attention (MQA) for faster large-batch decoding Li et al. (2023); Bavarian et al. (2022); Shazeer (2019).

StarCoder2: scaling open code LMs with The Stack v2. StarCoder2 provides 3B/7B/15B parameter checkpoints trained on 3.3–4.3T tokens, incorporating Fill-in-the-Middle (FIM) training

Metric	Measured on	Typical definition
pass@k (functional correctness) Chen et al. (2021)	k sampled candidates per problem	A problem is counted as solved if *at least one* of the k sampled programs passes the full unit-test suite; reported as pass@1, pass@10, pass@100. Chen et al. (2021)
pass@1 (single-sample correctness) Chen et al. (2021)	n sampled candidates per problem	Special case of the unbiased pass@k estimator with k=1; with $n = 200$ samples, this is the mean single-sample correctness over the sampled candidates. Chen et al. (2021)
CodeBLEU Ren et al. (2020)	Generated code vs. reference code	A weighted combination of (i) BLEU n-gram match, (ii) weighted n-gram match, (iii) AST match, and (iv) data-flow match. Ren et al. (2020)
Text overlap Papineni et al. (2002); Lin (2004); Popović (2015)	Generated text/code vs. references	Text overlap metrics: BLEU (precision-weighted n-gram overlap) Papineni et al. (2002), ROUGE (recall-oriented overlap family) Lin (2004), and chrF (character n-gram F-score) Popović (2015).
Diversity of samples Paul et al. (2024)	Multiple candidates for the same prompt/problem	Statistics over a set of sampled solutions, such as pairwise similarity distribution, distinct-n, or de-duplication rate, often reported together with pass@k to measure "diverse and correct" generation. Paul et al. (2024)

Table 4: **Common metrics for model-generated code.** The main experiments use pass@1 because recursive retraining changes single-sample correctness, while the table lists complementary metrics that capture functional correctness, lexical overlap, structural similarity, and sample diversity.

and fast decoder inference via multi-query style attention Lozhkov et al. (2024); Bavarian et al. (2022); Shazeer (2019).

C Additional Robustness and Diagnostics

This section provides additional diagnostics of the recursive training results, including external anchors, corrupted exogenous gates, seed robustness, and early-versus-late filter behavior. These analyses test how the observed degradation changes under alternative review and training conditions.

C.1 Effect of External Verified Anchors

We first test whether adding a small amount of fresh verified data can stabilize recursive training. Table 5 compares synthetic-only recursion, execution-filtered recursion, and execution-filtered recursion with a 5% external anchor.

Table 5: External anchor ablation at Round 5.

Selection policy	Anchor fraction	Raw gate pass	Candidates / 5k	S_5/S_0	Retention area	Accepted execution
Synthetic only, no gate	0%	1.00	5,000	0.364	2.46	34.6%
Execution-filtered synthetic	0%	0.894	5,593	0.247	2.11	89.4%
Execution-filtered + fresh anchor	5%	0.898	5,290 + 250 anchor	0.180	1.354	89.8%

The execution-filtered conditions substantially increase accepted-set execution validity, from 34.6% to approximately 90%. However, the 5% fresh anchor does not improve long-horizon retention in this experiment: its S_5/S_0 remains 0.180 and below the initial baseline. Thus, a small external anchor alone is insufficient to prevent recursive degradation.

C.2 Sensitivity to Exogenous-Gate Noise

We next test whether the behavior of an exogenous gate changes when its decisions are corrupted. Table 6 injects Bernoulli decision flips at increasing noise levels.

Injecting gate noise sharply changes both the effective acceptance mass and the resulting benchmark trajectory. The endpoints are not monotonic in the flip rate—for example, HumanEval retention recovers at higher noise while MBPP retention does not. We therefore interpret this experiment

Table 6: Noisy exogenous gate summary.

Gate noise	**Effective flip rate** η	**Acceptance mass** Z_t	**HumanEval** S_5/S_0	**MBPP** S_5/S_0	$\Delta_{0\to5}$
$\sigma = 0.00$	0.00	1.00	0.230	0.212	-0.109
flip $\eta = 0.05$	0.050	0.535	0.041	0.106	-0.141
flip $\eta = 0.10$	0.099	0.508	0.176	0.117	-0.126
flip $\eta = 0.20$	0.198	0.495	0.257	0.049	-0.123

as evidence that recursive behavior is sensitive to gate calibration, rather than as a monotonic noise–performance relationship.

C.3 Robustness Across Additional Runs

We additionally examine the available robustness runs for SantaCoder and Qwen2.5-Coder under the three core review regimes. Table 7 reports their Round 5 performance, retention area, and degradation slope.

Table 7: Focused single-seed robustness summary at Round 5.

Model	**Regime**	**HumanEval R5**	**MBPP R5**	**Retention area**	**Slope** $\Delta_{0\to5}$
SantaCoder-1.1B	No review	0.079	0.027	2.166	-0.139
SantaCoder-1.1B	Exogenous gate	0.092	0.045	2.106	-0.110
SantaCoder-1.1B	AI self-gate	0.116	0.079	2.119	-0.107
Qwen2.5-Coder-1.5B	No review	0.049	0.085	1.852	-0.150
Qwen2.5-Coder-1.5B	Exogenous gate	0.104	0.148	1.380	-0.109
Qwen2.5-Coder-1.5B	AI self-gate	0.146	0.082	1.648	-0.126

All reported runs still exhibit substantial degradation by Round 5. The relative gate behavior is model-dependent: AI self-gating gives the highest HumanEval endpoint for both models, whereas the exogenous gate gives the highest MBPP endpoint for Qwen2.5-Coder. This again supports the main finding that gating changes the degradation trajectory without producing a universally dominant review policy.

D Formal Gated-Training Analysis

Human-style target and automatic proxies. The ideal human-style target is broader than any single automatic check. It would accept code that is semantically correct for the task, executes safely, avoids obvious inefficiency or brittle behavior, satisfies security and compliance constraints, and remains maintainable. The Quality-Feasible Code Set in Appendix G formalizes this target as a conjunction of such requirements. Our implemented Compile and Quality gates are not literal manual review. They are automatic, model-independent proxies for parts of that target: compilation checks syntactic/executable validity, while the static quality filter removes short or repetitive generations. This distinction matters for interpretation. The experiments test whether a fixed external proxy signal behaves differently from a model-coupled self-review signal; they do not claim that the simple proxy fully matches a human pull-request reviewer.

D.1 When AI review matches exogenous review: equivalence and a non-collapse condition

Uniform calibration error. Recall the exogenous-gate conditional $q^H_{\theta_t}$ in Equation (2.16) and the AI-self-gate conditional $q^A_{\theta_t,\phi_t}$ in Equation (2.17). Here, $r_H(x, c)$ denotes the human-style external review rule, while $r_{\phi_t}(x, c)$ denotes AI review: the learned reviewer scores whether generated code should be accepted for future training. We measure how well the AI reviewer matches the exogenous reviewer by the worst-case calibration error

$$\varepsilon_t := \sup_{x\in\mathcal{X},c\in\mathcal{C}} \left| r_{\phi_t}(x, c) - r_H(x, c) \right| . \tag{D.1}$$

Non-collapse exogenous acceptance mass. To avoid a degenerate case where almost no generated code is accepted by the exogenous reviewer, we require the human-style gate to accept a non-vanishing fraction of the generator's probability mass.

Assumption D.1 (Non-collapse exogenous acceptance mass)**.** There exists $z_0 \in (0, 1]$ such that

$$\begin{aligned} Z^H_{\theta_t}(x) &= \sum_{c'} p_{\theta_t}(c' \mid x) r_H(x, c') \\ &\geq z_0 \quad \text{for } p_X\text{-a.e. } x \text{ and all } t. \end{aligned} \tag{D.2}$$

Assumption D.1 says that the external reviewer still finds enough acceptable code in the generator distribution. Without this lower bound, the accepted distribution can become unstable because it is normalized by a vanishing acceptance mass.

Quality gate sufficient condition. A simple sufficient condition for Assumption D.1 follows from the quality gate set $\mathcal{F}_{\mathrm{qual}}(x)$. Suppose the exogenous reviewer gives nontrivial acceptance probability to quality gate code: $c \in \mathcal{F}_{\mathrm{qual}}(x) \Rightarrow r_H(x, c) \geq \delta_H$ for some $\delta_H \in (0, 1]$. If the generator also assigns nontrivial mass to quality gate code, $p_{\theta_t}(\mathcal{F}_{\mathrm{qual}}(x) \mid x) \geq \beta_H$ for some $\beta_H > 0$, then Assumption D.1 holds with $z_0 = \delta_H \beta_H$, since

$$\begin{aligned} Z^H_{\theta_t}(x) &= \sum_{c'} p_{\theta_t}(c' \mid x) r_H(x, c') \\ &\geq \sum_{c' \in \mathcal{F}_{\mathrm{qual}}(x)} p_{\theta_t}(c' \mid x) r_H(x, c') \\ &\geq \delta_H p_{\theta_t}(\mathcal{F}_{\mathrm{qual}}(x) \mid x) \\ &\geq \delta_H \beta_H. \end{aligned} \tag{D.3}$$

Thus, the exogenous gate remains stable when it accepts quality gate code and the generator has not lost all probability mass on such code.

Theorem D.2 (Uniform approximation of exogenous-gated review)**.** *Suppose the AI reviewer satisfies Equation* (D.1) *with $\varepsilon_t < z_0$, and suppose Assumption D.1 holds. Then for p_X-almost every x,*

$$\left\| q^A_{\theta_t, \phi_t}(\cdot \mid x) - q^H_{\theta_t}(\cdot \mid x) \right\|_1 \leq \frac{2\varepsilon_t}{z_0 - \varepsilon_t}. \tag{D.4}$$

Consequently,

$$\begin{aligned} \left\| m^A_t - m^H_t \right\|_1 &= \int_{\mathcal{X}} p_X(x) \left\| q^A_{\theta_t, \phi_t}(\cdot \mid x) - q^H_{\theta_t}(\cdot \mid x) \right\|_1 dx \\ &\leq \frac{2\varepsilon_t}{z_0 - \varepsilon_t}. \end{aligned} \tag{D.5}$$

The proof is in Appendix F. Theorem D.2 shows when AI review remains safe: the AI reviewer must stay close to exogenous review, and the exogenous gate must accept nonzero generator mass. If ε_t grows or z_0 shrinks, the bound weakens and AI review can approach ungated self-training.

Equivalently, AI-gated recursion avoids collapse only when $\varepsilon_t \ll z_0$ and $r_H(x, \cdot)$ is not approximately constant on $\mathrm{supp}(p_{\theta_t}(\cdot \mid x))$. These conditions mean that AI review remains calibrated to an exogenous review standard, and that standard still separates good from bad generated code.

E Representation-Subspace Drift Under Recursive Self-Training

This recursion (2.12) should be read as a local approximation of how one round of retraining transforms representation covariance. For the closed-form concentration statement below, we analyze the self-adjoint local case $A = U \mathrm{diag}(a_1, \ldots, a_d) U^\top$ with $a_1 \geq a_2 \geq \cdots \geq a_d \geq 0$. Then $\Sigma_t = A^t \Sigma_0 (A^\top)^t$ and therefore

$$\|\Sigma_t\|_2 \leq \|A^t\|_2^2 \|\Sigma_0\|_2 = a_1^{2t} \|\Sigma_0\|_2. \tag{E.1}$$

More importantly, different eigen directions are amplified at different rates. For any unit vector v,

$$\begin{aligned} v^\top \Sigma_t v &= \|(A^\top)^t v\|^2_{\Sigma_0}, \\ \|(A^\top)^t v\| &\text{ scales as } a_i^t \text{ when } v = u_i. \end{aligned} \tag{E.2}$$

Thus, if $a_1 \gg a_2$, the component aligned with the leading eigen direction dominates after repeated retraining. Whenever Σ_0 has nonzero energy in this direction, the ratio $\lambda_1(\Sigma_t)/\lambda_2(\Sigma_t)$ is expected to increase with t because the leading component is amplified roughly by a_1^{2t} while the next component is amplified roughly by a_2^{2t}. This gives a representation-space explanation for collapse: recursive self-training can progressively concentrate variance into a low-dimensional subspace, reducing diversity in the generated code distribution.

We now formalize the concentration effect by the linearized covariance recursion in Equation (2.12).

Proposition E.1 (Spectral concentration)**.** *Under the covariance recursion in Equation* (2.12)*, assume the local operator is symmetric positive semidefinite,* $A = U\mathrm{diag}(a_1, \ldots, a_d)U^\top$*, with* $a_1 > a_2 \geq \cdots \geq a_d \geq 0$*. Assume also that* Σ_0 *is diagonal in the same eigenbasis,* $\widetilde{\Sigma}_0 := U^\top \Sigma_0 U = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_d^2) \succeq 0$*. If* $\sigma_1^2 \geq \alpha > 0$ *and* $\rho_2 := a_2/a_1 < 1$*, then whenever* $\lambda_2(\Sigma_t) > 0$*,*

$$\frac{\lambda_1(\Sigma_t)}{\lambda_2(\Sigma_t)} \geq \frac{\alpha}{\|\Sigma_0\|_2 \rho_2^{2t}}. \tag{E.3}$$

If $\lambda_2(\Sigma_t) = 0$*, the ratio is infinite. Thus the leading-to-second eigenvalue ratio grows at least as fast as* ρ_2^{-2t} *up to the initial covariance scale.*

The proof is given in Appendix F. Proposition E.1 shows, under this self-adjoint local approximation, that recursive retraining can amplify the dominant representation direction faster than all others. This gives a representation-level mechanism for mode concentration: the model distribution can become increasingly low-rank even while training continues to optimize self-generated samples.

F Complete Proofs

F.1 Proof of Theorem 2.3 (AI Self-Gate Degeneracy)

Proof. Fix any x such that Assumption 2.2 holds and $\kappa(x) > 0$ (this is p_X-a.e. x). By Assumption 2.2, $r_{\theta_t}(x, c) = \kappa(x)$ for $p_{\theta_t}(\cdot \mid x)$-a.e. c. Therefore,

$$\begin{aligned} Z^A_{\theta_t,\theta_t}(x) &= \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) r_{\theta_t}(x, c') \\ &= \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) \kappa(x) \\ &= \kappa(x) \sum_{c' \in \mathcal{C}} p_{\theta_t}(c' \mid x) \\ &= \kappa(x). \end{aligned} \tag{F.1}$$

For any c in the $p_{\theta_t}(\cdot \mid x)$-support where $r_{\theta_t}(x, c) = \kappa(x)$,

$$\begin{aligned} q^A_{\theta_t,\theta_t}(c \mid x) &= \frac{p_{\theta_t}(c \mid x) r_{\theta_t}(x, c)}{Z^A_{\theta_t,\theta_t}(x)} \\ &= \frac{p_{\theta_t}(c \mid x)\kappa(x)}{\kappa(x)} \\ &= p_{\theta_t}(c \mid x), \end{aligned} \tag{F.2}$$

which proves $q^A_{\theta_t,\theta_t}(\cdot \mid x) = p_{\theta_t}(\cdot \mid x)$ as measures (for p_X-a.e. x). Multiplying by $p_X(x)$ yields $m_t^A(x, c) = p_X(x) p_{\theta_t}(c \mid x) = m_t^{\mathrm{ungated}}(x, c)$ almost everywhere. Substituting $m_t^A = m_t^{\mathrm{ungated}}$ into the respective MLE objectives shows the two argmax sets coincide. □

F.2 Proof of Theorem D.2 (Uniform-Approximation Equivalence)

Proof. Fix x with $Z^H_{\theta_t}(x) \geq z_0$. First note that

$$\begin{aligned} \left| Z^A_{\theta_t,\phi_t}(x) - Z^H_{\theta_t}(x) \right| &= \left| \sum_{c'} p_{\theta_t}(c' \mid x) \big(r_{\phi_t}(x, c') - r_H(x, c') \big) \right| \\ &\leq \varepsilon_t, \end{aligned} \tag{F.3}$$

hence $Z^A_{\theta_t,\phi_t}(x) \geq Z^H_{\theta_t}(x) - \varepsilon_t \geq z_0 - \varepsilon_t > 0$. Now write

$$q^A_{\theta_t,\phi_t}(c \mid x) - q^H_{\theta_t}(c \mid x) = p_{\theta_t}(c \mid x) \left(\frac{r_{\phi_t}(x,c)}{Z^A_{\theta_t,\phi_t}(x)} - \frac{r_H(x,c)}{Z^H_{\theta_t}(x)} \right), \tag{F.4}$$

so by adding and subtracting $r_H(x,c)/Z^A_{\theta_t,\phi_t}(x)$ and using triangle inequality,

$$\begin{aligned} \left\| q^A_{\theta_t,\phi_t}(\cdot \mid x) - q^H_{\theta_t}(\cdot \mid x) \right\|_1 &\leq \sum_c p_{\theta_t}(c \mid x) \frac{|r_{\phi_t}(x,c) - r_H(x,c)|}{Z^A_{\theta_t,\phi_t}(x)} \\ &\quad + \sum_c p_{\theta_t}(c \mid x) r_H(x,c) \left| \frac{1}{Z^A_{\theta_t,\phi_t}(x)} - \frac{1}{Z^H_{\theta_t}(x)} \right| \\ &\leq \frac{\varepsilon_t}{Z^A_{\theta_t,\phi_t}(x)} + Z^H_{\theta_t}(x) \cdot \frac{|Z^A_{\theta_t,\phi_t}(x) - Z^H_{\theta_t}(x)|}{Z^A_{\theta_t,\phi_t}(x) Z^H_{\theta_t}(x)} \\ &\leq \frac{\varepsilon_t}{z_0 - \varepsilon_t} + \frac{\varepsilon_t}{z_0 - \varepsilon_t} \\ &= \frac{2\varepsilon_t}{z_0 - \varepsilon_t}, \end{aligned} \tag{F.5}$$

where $\varepsilon_t < z_0$ ensures the denominator is positive. Integrating over x w.r.t. p_X yields (D.5). □

F.3 Proof of Proposition E.1 (Spectral Concentration)

Proof of Proposition E.1. Assume the noiseless recursion $\Sigma_t = A^t \Sigma_0 (A^\top)^t$ and the self-adjoint local form $A = U \mathrm{diag}(a_1, \ldots, a_d) U^\top$ with $a_1 > a_2 \geq \cdots \geq a_d \geq 0$. Since A is symmetric positive semidefinite,

$$A^t = U \mathrm{diag}(a_1^t, \ldots, a_d^t) U^\top. \tag{F.6}$$

With $\widetilde{\Sigma}_0 = U^\top \Sigma_0 U = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_d^2)$, this gives

$$\begin{aligned} \Sigma_t &= U \mathrm{diag}(a_1^t, \ldots, a_d^t) \widetilde{\Sigma}_0 \mathrm{diag}(a_1^t, \ldots, a_d^t) U^\top \\ &= U \mathrm{diag}(\sigma_1^2 a_1^{2t}, \ldots, \sigma_d^2 a_d^{2t}) U^\top. \end{aligned} \tag{F.7}$$

The top eigenvalue is lower bounded by the energy in the first eigen direction:

$$\lambda_1(\Sigma_t) \geq \sigma_1^2 a_1^{2t} \geq \alpha a_1^{2t}. \tag{F.8}$$

Every direction orthogonal to u_1 is amplified by at most a_2^{2t}, so the second eigenvalue satisfies

$$\lambda_2(\Sigma_t) \leq \|\Sigma_0\|_2 a_2^{2t} = \|\Sigma_0\|_2 a_1^{2t} \rho_2^{2t}. \tag{F.9}$$

Combining (F.8)–(F.9) gives the explicit ratio bound

$$\frac{\lambda_1(\Sigma_t)}{\lambda_2(\Sigma_t)} \geq \frac{\alpha}{\|\Sigma_0\|_2 \rho_2^{2t}}. \tag{F.10}$$

If $\lambda_2(\Sigma_t) = 0$, the ratio is infinite and the same conclusion holds. In particular, when $a_1 \gg a_2$ (so $\rho_2 = a_2/a_1 \ll 1$), the ratio grows at least as fast as ρ_2^{-2t}, yielding spectral concentration along the leading eigen direction u_1. □

G Quality Gate Set: Full Definition

G.1 Quality Dimensions for Code Solutions

Let q denote a task/query and let $\mathcal{D}_q$ be the induced input distribution (test-time instances) for q. For a candidate program $x \in \mathcal{X}$, let $R(x,i)$ denote the execution output on input $i \sim \mathcal{D}_q$, and let $y(q,i)$ denote the task-specific semantic ground truth. We define the following measurable quality dimensions, each of which can be used either as an evaluation metric or as a constraint in a quality gate set.

(A) Semantic correctness. We quantify semantic success by the probability of producing the correct semantic output under $\mathcal{D}_q$:

$$m_{\mathrm{corr}}(x;q) := \Pr_{i\sim\mathcal{D}_q}\big[R(x,i) = y(q,i)\big]. \tag{G.1}$$

When only a reference answer is available (instead of $y(q,i)$), the event $R(x,i) = y(q,i)$ can be replaced by "execution matches the reference after normalization."

(B) Runtime safety (no crashes/undefined behavior). Let $\mathsf{Crash}(x,i)$ be an indicator of runtime failure on input i (e.g., uncaught exceptions, out-of-bounds access, non-termination detected by a timeout, or undefined behavior in low-level languages). We define

$$m_{\mathrm{safe}}(x;q) := 1 - \Pr_{i\sim\mathcal{D}_q}\big[\mathsf{Crash}(x,i)\big]. \tag{G.2}$$

Equivalently, one may impose a hard safety constraint:

$$\Pr_{i\sim\mathcal{D}_q}\big[\mathsf{Crash}(x,i)\big] \le \delta_{\mathrm{safe}}, \tag{G.3}$$

for a user-specified tolerance $\delta_{\mathrm{safe}} \in (0,1)$.

(C) Robustness to perturbations and distribution shift. Let $\mathcal{T}(i)$ denote a set of admissible perturbations of an input i (e.g., format variations, paraphrases, boundary-value modifications). We define robust correctness as

$$m_{\mathrm{rob}}(x;q) := \Pr_{i\sim\mathcal{D}_q}\Big[\forall i' \in \mathcal{T}(i),\ R(x,i') = y(q,i')\Big]. \tag{G.4}$$

Alternatively, using a loss $\ell(\cdot,\cdot)$, we define a worst-case risk:

$$\mathcal{R}_{\mathrm{rob}}(x;q) := \mathbb{E}_{i\sim\mathcal{D}_q}\Big[\sup_{i'\in\mathcal{T}(i)} \ell\big(R(x,i'), y(q,i')\big)\Big]. \tag{G.5}$$

(D) Resource usage and efficiency. Let $T(x,i)$ be the runtime cost (e.g., wall-clock time) and optionally $M(x,i)$ the memory footprint or $C(x,i)$ the number of external API calls. A simple efficiency score is the negative expected runtime:

$$m_{\mathrm{res}}(x;q) := -\mathbb{E}_{i\sim\mathcal{D}_q}\big[T(x,i)\big]. \tag{G.6}$$

One may also enforce probabilistic latency constraints:

$$\Pr_{i\sim\mathcal{D}_q}\big[T(x,i) \le \tau\big] \ge 1 - \delta_{\mathrm{time}}, \tag{G.7}$$

or tail-latency bounds (e.g., the 0.99-quantile):

$$\mathrm{Quantile}_{0.99}\big(T(x,i)\big) \le \tau_{0.99}. \tag{G.8}$$

(E) Security and compliance. Let $\mathsf{Bad}(x,i)$ indicate unsafe or policy-violating behavior (e.g., invoking dangerous system calls, writing to sensitive paths, unintended network access, or insecure deserialization). We define

$$m_{\mathrm{sec}}(x;q) := 1 - \Pr_{i\sim\mathcal{D}_q}\big[\mathsf{Bad}(x,i)\big], \tag{G.9}$$

or impose a hard constraint $\Pr[\mathsf{Bad}] \le \delta_{\mathrm{sec}}$ for $\delta_{\mathrm{sec}} \in (0,1)$.

(F) Maintainability and testability. Maintainability can be captured by static code metrics such as cyclomatic complexity, code duplication, and dependency burden:

$$m_{\mathrm{maint}}(x) := -\Big(\lambda_1 \mathrm{Cyclomatic}(x)\ + \lambda_2 \mathrm{Dup}(x) + \lambda_3 \mathrm{Deps}(x)\Big), \tag{G.10}$$

with weights $\lambda_1, \lambda_2, \lambda_3 \ge 0$. If a unit-test distribution $\mathcal{T}_{\mathrm{unit}}$ is available, test pass rate provides an executable proxy:

$$m_{\mathrm{test}}(x) := \Pr_{t\sim\mathcal{T}_{\mathrm{unit}}}\big[x \text{ passes test } t\big]. \tag{G.11}$$

(G) Readability and style (including documentation consistency). Let $S_{\mathrm{read}}(x) \in [0,1]$ be a readability/style score from either rule-based or learned evaluators. We set

$$m_{\mathrm{read}}(x) := S_{\mathrm{read}}(x). \tag{G.12}$$

When inline documentation $\mathrm{doc}(x)$ is present, one may additionally measure documentation–implementation consistency via an alignment score $\mathrm{Align}\big(x, \mathrm{doc}(x)\big)$, where larger values indicate better semantic agreement.

G.2 Quality-Feasible Code Set

Let q denote a task/query, $\mathcal{X}$ the program space, and $\mathcal{D}_q$ the input distribution induced by q. Let $V : \mathcal{X} \to \{0,1\}$ be a feasibility verifier, $R(x,i)$ the execution result of program x on input i, and $y(q,i)$ the task-specific semantic ground truth. We further define runtime event predicates $\mathsf{Crash}(x,i)$ (crash/exception/undefined behavior) and $\mathsf{Bad}(x,i)$ (unsafe or policy-violating behavior), a runtime cost $T(x,i)$ (e.g., wall-clock time), and a readability/style scorer $S_{\mathrm{read}}(x) \in [0,1]$. For user-specified thresholds $\delta_{\mathrm{safe}}, \delta_{\mathrm{sec}} \in (0,1)$ and $\tau, r_0 > 0$, we define the *quality-feasible* solution set as

$$\begin{aligned}\mathcal{F}_{\mathrm{qual}}(q) := \Big\{ x \in \mathcal{X} : & \underbrace{V(x) = 1}_{\text{feasibility}}, \\ & \underbrace{\Pr_{i \sim \mathcal{D}_q}\big[R(x,i) = y(q,i)\big] \geq \delta_{\mathrm{corr}}}_{\text{semantic correctness}}, \\ & \underbrace{\Pr_{i \sim \mathcal{D}_q}\big[\mathsf{Crash}(x,i)\big] \leq \delta_{\mathrm{safe}}}_{\text{runtime safety}}, \\ & \underbrace{\mathbb{E}_{i \sim \mathcal{D}_q}\big[T(x,i)\big] \leq \tau}_{\text{resource/efficiency}}, \\ & \underbrace{\Pr_{i \sim \mathcal{D}_q}\big[\mathsf{Bad}(x,i)\big] \leq \delta_{\mathrm{sec}}}_{\text{security/compliance}}, \\ & \underbrace{S_{\mathrm{read}}(x) \geq r_0}_{\text{readability/style}} \Big\},\end{aligned} \tag{G.13}$$

where $\delta_{\mathrm{corr}} \in (0,1]$ specifies the minimum acceptable semantic success rate. If only a reference answer is available (rather than $y(q,i)$), the event $R(x,i) = y(q,i)$ can be replaced by "execution matches the reference after normalization."

G.3 Supporting Assumptions

Assumption G.1 (Non-trivial quality-feasible completeness / coverage)**.** Let $\mathcal{D}$ denote the target distribution over queries q, and for each q let $\mathcal{D}_q$ be the induced input distribution. Let $V : \mathcal{X} \to \{0,1\}$ be a feasibility verifier and define the quality gate set $\mathcal{F}_{\mathrm{qual}}(q) \subseteq \mathcal{X}$ as in the main text (incorporating feasibility, semantic correctness, runtime safety, efficiency, security/compliance, and readability/style). Then for $\mathcal{D}$-almost every query q, there exists a non-empty subset $\mathcal{F}_{\mathrm{qual}}(q)$ and constants $\alpha, \beta \in (0,1]$ such that

$$\mu\big(\mathcal{F}_{\mathrm{qual}}(q)\big) \geq \alpha, \qquad P_{\theta_0}\big(\mathcal{F}_{\mathrm{qual}}(q) \mid q\big) \geq \beta, \tag{G.14}$$

where μ is a reference measure on $\mathcal{X}$ (or on the relevant solution equivalence class for q), and $P_{\theta_0}(\cdot \mid q)$ is the initial generator's conditional distribution.

Assumption G.2 (Quality-feasible margin)**.** Let $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ be a metric (or pseudometric) on $\mathcal{X}$, and define

$$d(x, \mathcal{F}_{\mathrm{qual}}(q)) := \inf_{y \in \mathcal{F}_{\mathrm{qual}}(q)} d(x,y). \tag{G.15}$$

There exists $\gamma > 0$ such that for $\mathcal{D}$-almost every q, for all $x \in \mathcal{F}_{\mathrm{qual}}(q)$ and all $x' \in \mathcal{X}$,

$$d(x,x') \leq \gamma \Longrightarrow x' \in \mathcal{F}_{\mathrm{qual}}(q). \tag{G.16}$$

Equivalently, the γ-ball around any $x \in \mathcal{F}_{\mathrm{qual}}(q)$ is contained in $\mathcal{F}_{\mathrm{qual}}(q)$.

Assumption G.3 (Local contraction toward the quality gate set)**.** Let $C : \mathcal{X} \to \mathcal{X}$ denote a correction (or calibration) operator applied to candidate programs. There exist constants $\rho \in (0, 1)$ and $\eta \geq 0$ such that for $\mathcal{D}$-almost every q and all x in the region of interest,

$$d\big(C(x), \mathcal{F}_{\mathrm{qual}}(q)\big) \;\leq\; \rho\, d\big(x, \mathcal{F}_{\mathrm{qual}}(q)\big) \;+\; \eta. \tag{G.17}$$

When $\eta = 0$, C is a strict contraction toward $\mathcal{F}_{\mathrm{qual}}(q)$; when $\eta > 0$, the term η captures an irreducible residual error (noise floor).

H AI Self-Verification Protocol

H.1 Implemented AI Self-Gates

This appendix details the two AI self-gate methods used in our experiments: **Perplexity** and **Binary Classifier**. We use these methods as representative AI self-verification protocols because both derive acceptance scores from the code LLM itself, rather than from human labels, execution tests, or external static analyzers. Thus, the same model family that generates candidate code also evaluates which samples enter future training.

Basic formulation. Let q denote the programming problem and let x denote a code sample generated by the current code LLM. We construct an evaluation prompt $\Pi_{\mathrm{eval}}(q, x)$ that contains the problem statement, the candidate code, and an explicit request for assessment. The model then produces an evaluation response

$$r \sim p_\theta(r \mid \Pi_{\mathrm{eval}}(q, x)). \tag{H.1}$$

This is a self-evaluation procedure because the same parameterized model p_θ first generates x and then, under a different prompt or scoring rule, evaluates x.

Binary-classifier self-gate. The binary self-gate forces the model into a two-label judgment. The acceptance score is

$$S_{\mathrm{bin}}(x \mid q) = \log P_\theta(\mathsf{good} \mid q, x) - \log P_\theta(\mathsf{bad} \mid q, x). \tag{H.2}$$

Samples are ranked by $S_{\mathrm{bin}}(x \mid q)$, and the top 25% are retained for the next round of recursive fine-tuning. This uses only the model parameters: no human label, execution result, or external quality metric is used.

Perplexity self-gate. Code LLMs are generative models trained to predict code tokens and are commonly used for code completion or masked/fill-in-the-middle generation. We therefore use the model's own conditional likelihood as an AI self-gate. For each generated sample x under problem context q, we compute

$$S_{\mathrm{ppl}}(x \mid q) = -\frac{1}{|x|} \sum_{i=1}^{|x|} \log P_\theta(x_i \mid q, x_{<i}). \tag{H.3}$$

Equivalently, the perplexity is

$$\mathrm{PPL}_\theta(x \mid q) = \exp\big(S_{\mathrm{ppl}}(x \mid q)\big). \tag{H.4}$$

Generated samples with lower $S_{\mathrm{ppl}}(x \mid q)$, or lower $\mathrm{PPL}_\theta(x \mid q)$, are treated as more acceptable. In our implementation, samples are ranked by perplexity and the lowest 25% are retained for future training. This is an AI self-gate because the same code LLM family generates x through completion/filling and then scores x by its own token likelihood.

H.2 Other Candidate AI Self-Verifier Protocols

The following protocols are natural extensions of AI self-verification, but are not used as main filtering methods in our experiments.

Chain-of-thought judge followed by forced verdict. A stronger AI verifier can first generate an assessment trace and then output a forced final verdict. The evaluation prompt may request the model to summarize the intended algorithm, identify possible bugs or edge-case failures, and then output a final label from a fixed set. The model generates a reasoning trace z and a verdict y:

$$p_\theta(y, z \mid q, x) = p_\theta(z \mid q, x) p_\theta(y \mid q, x, z). \tag{H.5}$$

Self-consistency over judge samples. Since one judge response can be noisy, the model can sample multiple evaluations:

$$r^{(1)}, r^{(2)}, \ldots, r^{(M)} \sim p_\theta(\cdot \mid \Pi_{\text{eval}}(q, x)). \tag{H.6}$$

The final self-judgment can be obtained by majority vote,

$$\hat{y}_{\text{vote}}(x \mid q) = \text{mode}\{y^{(1)}, \ldots, y^{(M)}\}, \tag{H.7}$$

or by averaged label probabilities:

$$\hat{y}_{\text{avg}}(x \mid q) = \arg\max_{y \in \mathcal{Y}} \frac{1}{M} \sum_{m=1}^{M} P_\theta^{(m)}(y \mid q, x). \tag{H.8}$$

Constrained multi-label judge. For code LLMs, the evaluation prompt should be constrained. A weak prompt such as "Is this code good?" is underspecified. A more controlled prompt asks the model to check whether the implementation matches the specification, handles corner cases, and avoids likely bugs, then end with exactly one label from a fixed set such as `CORRECT`, `LIKELY_CORRECT`, `UNCERTAIN`, `LIKELY_BUGGY`, or `BUGGY`. This still remains an AI self-gate if the score is produced by the same model family that generated the code.

Why AI self-evaluation degrades. LLM evaluators exhibit significant self-preference bias Panickssery et al. (2024): they assign higher scores to outputs in their own generation style, regardless of actual quality. As the model collapses and its style converges to a narrow distribution, the evaluator's preferences converge with it, causing the gate to accept progressively worse outputs with progressively higher confidence. This is the self-confirming regime formalized in Assumption 2.2 and Theorem 2.3. For production-quality judging protocols that mitigate this bias, see Panickssery et al. (2024); Hashemi et al. (2024); Zheng et al. (2023).

I Mixed Training

Code LLMs differ in their pretraining data mixtures, and we adapt the recursive fine-tuning format accordingly. The key distinction is whether a model was trained mainly on raw source code or on a broader mixture of code and code-related natural language.

SantaCoder. SantaCoder Ben Allal et al. (2023) is pretrained on Python, Java, and JavaScript from The Stack Kocetkov et al. (2022). Its data consists primarily of source-code files, with natural language appearing only inside comments, docstrings, and identifiers. We therefore fine-tune SantaCoder using pure code samples in the recursive loop.

StarCoder and StarCoder2. StarCoder Li et al. (2023) and StarCoder2 Lozhkov et al. (2024) are trained on broader code corpora that include source code together with code-related natural language, such as issues, commits, pull-request text, and notebook content. For these models, recursive fine-tuning uses a mixed code–natural-language format to better match the original training distribution.

Qwen2.5-Coder. Qwen2.5-Coder Hui et al. (2024) follows a multi-stage code-model training pipeline, including large-scale code pretraining, code-related natural-language data, and instruction tuning. We therefore also use a mixed code–natural-language fine-tuning format for Qwen2.5-Coder rather than pure code-only retraining.

J Compile Error Analysis

Figure 5 and Table 8 report the compile and execution-pass analysis of generated code under vanilla recursion and the compile-based exogenous gate. We evaluate 500 generated samples with a 5s timeout. Adding the exogenous compile gate substantially increases the relaxed pass rate: vanilla drops from 55.4% at R1 to 34.6% at R5, while compile filtering maintains a much higher relaxed pass rate of 88.2% at R1 and 89.4% at R10. The error distribution also changes sharply. Under

vanilla recursion, the "Other error" category grows from 42.8% to 64.2%, indicating increasing runtime failure. In contrast, the compile-based exogenous gate keeps "Other error" low, from 9.2% at R1 to 7.4% at R10. This shows that even a simple exogenous compile gate significantly improves executable validity and prevents runtime-error proliferation.

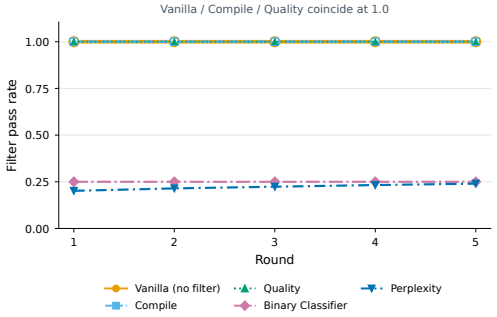

Figure 4: **Filter pass rate over rounds, averaged across all four models.** No-gate and exogenous-gate regimes remain stable by construction or fixed rules, while AI self-gates can lose discriminative power over recursive rounds.

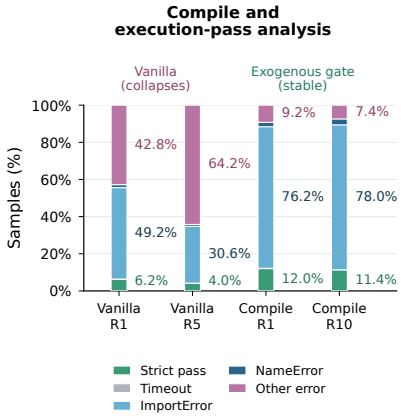

Figure 5: **Compile and execution-pass analysis across recursive retraining rounds.** Results compare vanilla recursion with the compile-based exogenous gate on 500 generated samples with a 5s timeout.

Table 8: Compile and execution-pass analysis on 500 generated samples with a 5s timeout. The table compares ungated vanilla recursion with the compile-based exogenous gate across strict pass, relaxed pass, and error categories.

Dataset	ok (strict)	Timeout	ImportError	NameError	Other error	Relaxed pass
Vanilla R1	6.2%	0.2%	49.2%	1.6%	42.8%	55.4%
Vanilla R5 (collapsed)	4.0%	0.2%	30.6%	1.0%	64.2%	34.6%
Compile R1	12.0%	0.2%	76.2%	2.4%	9.2%	88.2%
Compile R10	11.4%	0.0%	78.0%	3.2%	7.4%	89.4%

K SantaCoder: All Strategies Across Rounds

Main collapse result. All strategies decline from the pre-retraining SantaCoder baseline (HumanEval+ 0.171, MBPP+ 0.294) across 5 recursive rounds. Figure 6 and Table 10 summarize the results. Vanilla collapses most severely: HumanEval+ falls from 0.092 at R1 to 0.073 at R5, while MBPP+ reaches 0.008 at R4 and remains only 0.019 at R5 (from baseline 0.294; -93.5% at R5). Binary Classifier achieves the highest R5 HumanEval+ (0.104) but shows erratic MBPP+ behavior, dropping to 0.019 at R4. Quality filtering provides the most stable MBPP+ trajectory across rounds.

Per-round results for SantaCoder. Table 9 reports HumanEval+, MBPP+, LiveCodeBench, and filter pass rate for SantaCoder across all five strategies and five rounds. Figure 7 complements the table by contrasting the early checkpoint with the extended SantaCoder checkpoint, making the short-horizon and long-horizon effects of each filter visible side by side.

Perplexity self-gating loses calibration. Among all AI self-gate methods, perplexity filtering produces a more stable trajectory than binary classification on MBPP+. The filter pass rate for PPL slowly increases from 0.167 at R1 to 0.235 at R5 (Table 9), indicating gradual degradation of the self-calibration signal (Figure 4). This aligns with Theorem 2.3: once the generating model collapses, its perplexity signal becomes miscalibrated.

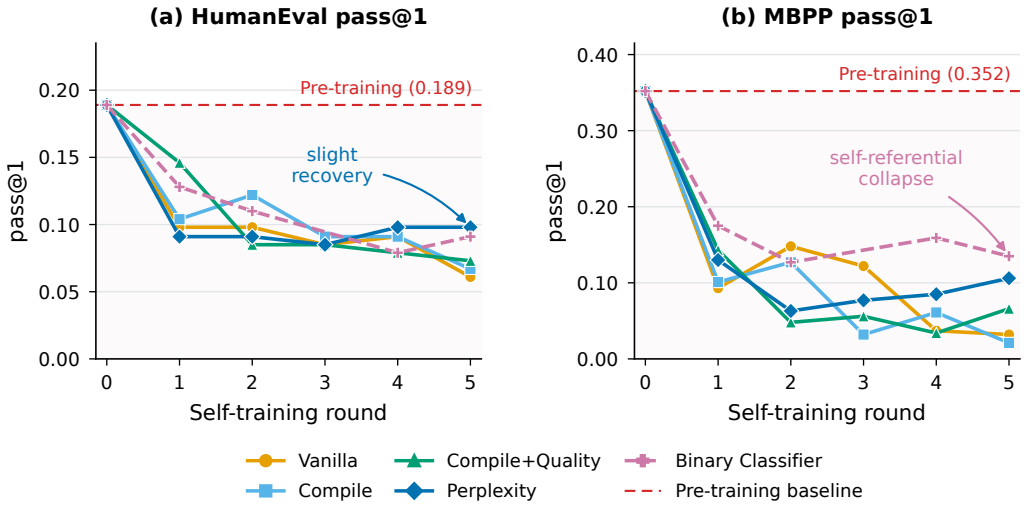

Figure 6: **Recursive self-training results on SantaCoder.** HumanEval pass@1 (left) and MBPP pass@1 (right) over rounds 0–5. Exogenous-gate methods are Compile and Compile+Quality; AI-self-gate methods are Perplexity and Binary Classifier. The red dashed line denotes the SantaCoder pretraining baseline.

L Per-Round Degradation Trajectories (SantaCoder, All Strategies)

Figure 8 reports the per-round degradation trajectories of all six filtering strategies on SantaCoder. Each panel corresponds to one strategy and shows HumanEval pass@1 and MBPP pass@1 across recursive retraining rounds. Dotted horizontal lines denote the pre-retraining baseline for each benchmark. Quality-20R is evaluated for 20 rounds with sparse checkpoints to study long-horizon behavior, while the other strategies are evaluated for 5–10 rounds.

Figure 8 shows that all strategies degrade below the pre-retraining baseline as recursive training proceeds. Vanilla collapses rapidly without any filtering, while exogenous-gate strategies generally preserve higher scores for longer. Quality-20R remains more stable over long horizons than the AI-self-gate methods. In contrast, Binary Classifier reaches a high early MBPP score but then drops sharply, matching the self-confirming gate failure described in Theorem 2.3.

M Cross-Model Degradation Trajectories

Heatmap of HumanEval+ across rounds. Figure 9 provides a compact overview of all 20 model×filter combinations across rounds 1–5. Brighter cells indicate higher pass@1; the progressive darkening from left to right within each model block illustrates the general collapse trend. StarCoder2-3B with binary filtering achieves the highest round-1 score (0.195) among all combinations, while Qwen2.5-Coder-1.5B with vanilla self-training collapses most severely, dropping to 0.043 at Round 5.

M.1 MBPP+ Collapse Pattern

MBPP+ degrades more severely than HumanEval+ across all models and filter strategies. Figure 10 shows MBPP+ trajectories for all models under all five filters. The most dramatic collapse is observed for SantaCoder under Vanilla self-training, where MBPP+ drops from the baseline 0.294 to 0.008 at Round 4 (-97.3%). Code Llama-7B is notably more robust to MBPP+ collapse, with MBPP+ under Vanilla remaining at 0.214 at Round 5, reflecting its stronger pretraining foundation.

Table 9: SantaCoder performance across recursive rounds. HE+ = HumanEval+ pass@1, MBPP+ = MBPP+ pass@1, LCB = LiveCodeBench pass@1, and FPR = filter pass rate. Round 0 baselines are HE+=0.171, MBPP+=0.294, and LCB=0.018.

Filter	Round	HE+	MBPP+	LCB	FPR
Vanilla	R1	0.092	0.172	0.000	1.000
	R2	0.092	0.109	0.000	1.000
	R3	0.073	0.074	0.000	1.000
	R4	0.067	0.008	0.000	1.000
	R5	0.073	0.019	0.000	1.000
Compile	R1	0.092	0.103	0.000	1.000
	R2	0.067	0.053	0.000	1.000
	R3	0.055	0.082	0.000	1.000
	R4	0.098	0.037	0.000	1.000
	R5	0.079	0.034	0.000	1.000
Quality	R1	0.092	0.183	0.000	1.000
	R2	0.073	0.058	0.000	1.000
	R3	0.067	0.034	0.000	1.000
	R4	0.067	0.037	0.000	1.000
	R5	0.061	0.069	0.000	1.000
PPL	R1	0.079	0.143	0.000	0.167
	R2	0.073	0.101	0.003	0.201
	R3	0.073	0.042	0.000	0.220
	R4	0.073	0.066	0.000	0.229
	R5	0.061	0.032	0.000	0.235
Binary	R1	0.110	0.061	0.000	0.250
	R2	0.079	0.061	0.000	0.250
	R3	0.073	0.048	0.000	0.250
	R4	0.061	0.019	0.000	0.250
	R5	**0.104**	0.058	0.000	0.250

Table 10: SantaCoder: all strategies at R1 and R5 (15k steps). Baseline: HE=0.189, MBPP=0.352. **Bold** = best at R5; underline = second best per column.

Method	Gate type	HE R1	MBPP R1	LCB R1	HE@R5	MBPP@R5	LCB@R5
Baseline (SantaCoder)	–	0.189	0.352	0.018	–	–	–
Vanilla	– (no gate)	0.098	0.201	0.000	0.079	0.027	0.000
Compile	Exogenous gate	0.110	0.122	0.000	0.092	0.045	0.000
Quality	Exogenous gate	0.116	0.212	0.000	0.067	**0.079**	0.000
Perplexity	AI self-gate	0.104	0.146	0.000	0.092	0.048	0.000
Binary Classifier	AI self-gate	**0.134**	0.077	0.000	**0.116**	0.079	0.000

M.2 LIVECODEBENCH: NEAR-ZERO COLLAPSE

Figure 12 shows LiveCodeBench pass@1 trajectories under the Vanilla filter for all four models. Despite Qwen2.5-Coder starting from the highest LCB baseline (0.238), it collapses to 0.000 by Round 5. Code Llama-7B starts at 0.045 and reaches approximately 0.003 at R5 under Vanilla. StarCoder2-3B and SantaCoder are already near zero throughout. This confirms that recursive self-training disproportionately degrades harder, out-of-distribution reasoning tasks even when introductory benchmarks show some residual performance.

This section reports cross-model degradation trajectories under vanilla recursion, exogenous-gate filtering, and AI-self-gate filtering. Figure 14 compares filtering strategies on StarCoder2-3B and Qwen2.5-Coder-1.5B. Figure 13 further reports compile-gated runs and a genuine extended vanilla Qwen2.5-Coder trajectory.

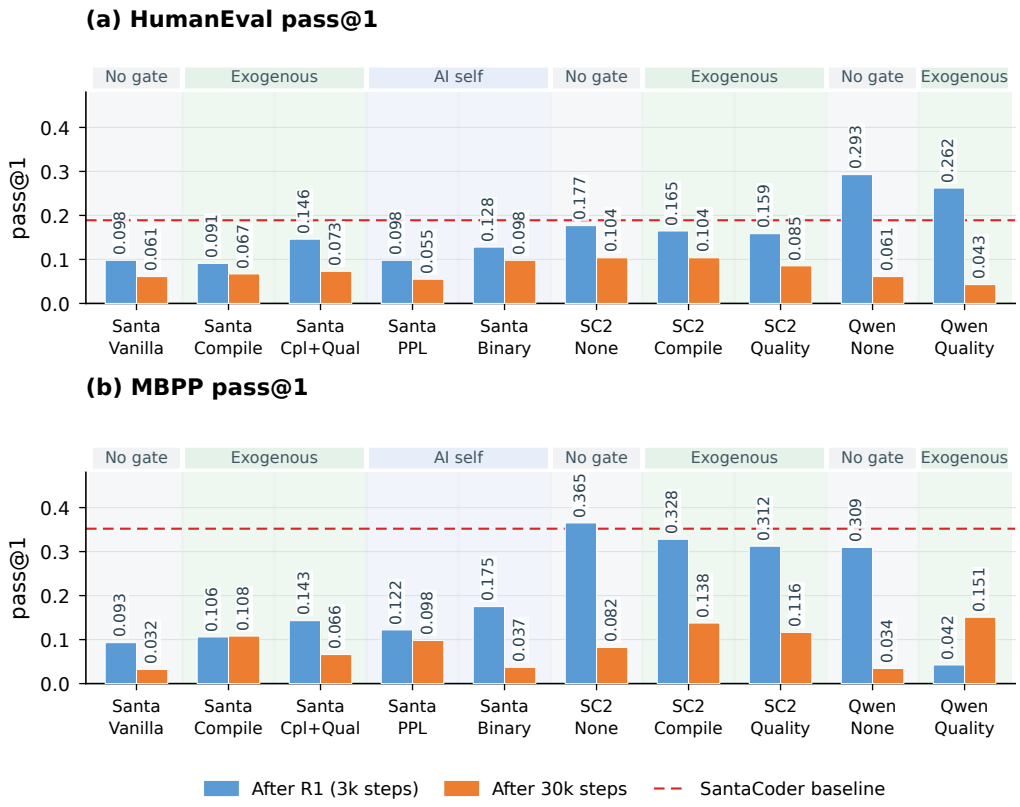

Figure 7: **Filtering strategies across models: early vs. extended recursive training.** Blue bars show pass@1 after round 1; orange bars show the extended 30k-step checkpoint where available. Gate labels appear above the panels; background shading is gray for no gate, green for exogenous gates, and light blue for AI self-gates. Red dashed lines mark the SantaCoder baseline.

Figures 14 and 13 show that recursive self-training collapse is not specific to SantaCoder. Under vanilla recursion, the tested model families fall far below their pre-retraining baselines, with Qwen2.5-Coder dropping fastest among the shown vanilla trajectories despite starting from the highest baseline. Exogenous-gate filtering can delay degradation, especially in the early rounds, but it does not fully prevent long-horizon decline. AI-self-gate filtering by PPL is also unstable: it remains below the pretraining baseline and often underperforms compile or quality filtering on MBPP. Overall, the cross-model results support the same narrower conclusion as the main experiments: all recursive regimes degrade below baseline, exogenous gates preserve executable validity, and AI self-gates do not reliably stop recursive degradation.

Figure 11 compares available vanilla recursive self-training trajectories across SantaCoder, StarCoder2-3B, and Qwen2.5-Coder. All shown models degrade rapidly from their step-0 pre-training baseline on both HumanEval and MBPP. Qwen2.5-Coder starts from the highest baseline but collapses the fastest, dropping sharply within the first few retraining steps. SantaCoder also falls quickly and reaches low scores after only a few rounds. StarCoder2-3B degrades more slowly than the smaller models, but it still remains far below its initial baseline after extended retraining. These results show that larger or stronger code models may slow the collapse rate, but vanilla recursive self-training still causes consistent long-horizon degradation.

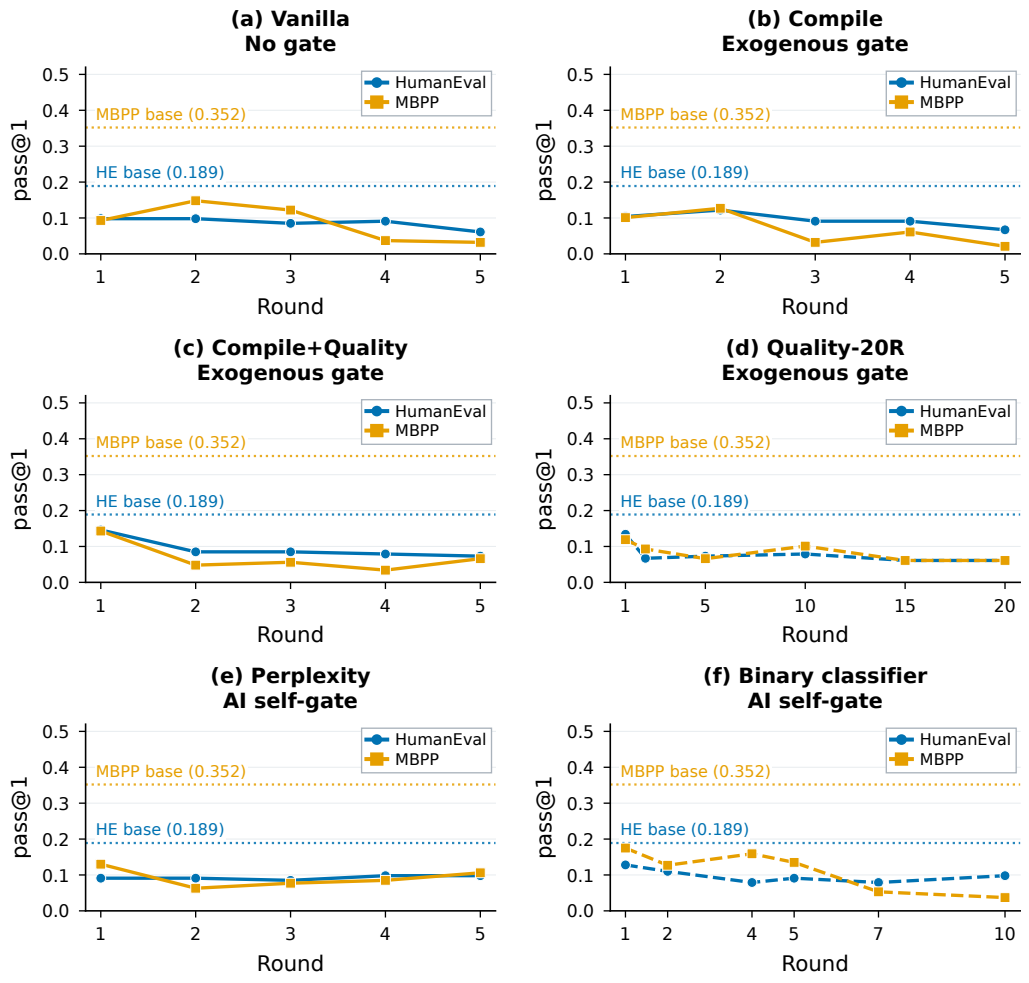

Figure 8: **SantaCoder per-strategy degradation trajectories.** Each panel shows one filtering strategy. HumanEval pass@1 is shown with blue circles, and MBPP pass@1 is shown with orange squares. Dotted horizontal lines denote the pre-retraining baseline for each benchmark. The panels include Vanilla, Compile, Compile+Quality, Quality-20R, Perplexity, and Binary Classifier.

M.3 Cross-Model Analysis

Figure 14 shows that both StarCoder2-3B and Qwen2.5-Coder-1.5B exhibit recursive self-training collapse regardless of filtering strategy, providing direct empirical corroboration that the collapse mechanism is not specific to SantaCoder.

Collapse magnitude. Under vanilla self-retraining (no filter), StarCoder2-3B declines from a HumanEval+ baseline of 0.274 to 0.104 by round 5 (-62.0%) and from an MBPP+ baseline of 0.492 to 0.172 (-65.0%). The steeper decline in Qwen2.5-Coder is consistent with the stronger initial distribution hypothesis: a model pretrained on a richer code corpus exhibits a sharper distributional shift when contaminated with self-generated outputs.

Effect of filtering strategies. The benefit of a gate depends on which failure mode the benchmark rewards. Compile filtering improves Qwen2.5-Coder at R5 relative to Vanilla on both HumanEval+ (0.098 vs. 0.043) and MBPP+ (0.127 vs. 0.082), and it is the strongest Code Llama-7B strategy on HumanEval+ (0.171). For StarCoder2-3B, however, compile and quality filters lag behind Vanilla at R5 on HumanEval+ and MBPP+, showing that syntactic validity alone cannot guarantee semantic

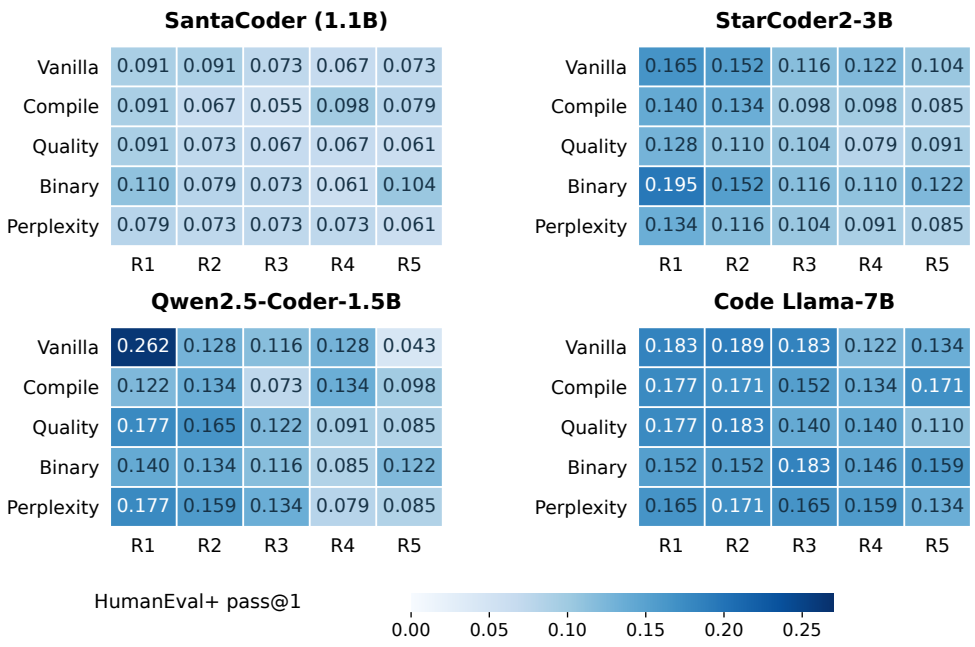

Figure 9: **HumanEval+ pass@1 heatmap over recursive rounds.** Each panel shows one model; rows are filters and columns are rounds 1–5. Darker cells indicate higher pass@1, and annotations give the values. All panels use the same color scale.

preservation. Binary filtering achieves the highest R5 HumanEval+ for three of four models, which means self-scoring can still rank some useful samples in early and medium horizons. Its MBPP+ behavior is weaker and less stable, matching our central story: AI self-gates may improve one visible metric while losing broader semantic coverage. Perplexity (PPL) filtering shows a slow increase in pass rate from approximately 0.17 to 0.24 (averaged across models; Figure 4), consistent with the endogenous degeneracy result of Theorem 2.3.

Collapse attractor. Despite large differences in architecture size, pretraining data, and starting pass@1, all four models show qualitatively similar long-horizon degradation patterns. SantaCoder (0.171 HumanEval+ baseline) and Qwen2.5-Coder (0.372 baseline) both reach approximately 0.04–0.10 HumanEval+ pass@1 under Vanilla by Round 5. StarCoder2-3B and Code Llama-7B maintain somewhat higher residual performance (0.10–0.14), likely due to stronger capacity and pretraining data quality. This convergence to a common degradation floor is consistent with the theoretical prediction of a collapse attractor.

Key takeaway. No filtering strategy evaluated here—Compile, Perplexity, Quality, or Binary—prevents collapse across all model families. Binary filtering achieves the best R5 HumanEval+ for three models, but this comes with erratic MBPP+ behavior. Exogenous gates are more interpretable and preserve executable validity, yet simple compile/static filters are still too weak to guarantee semantic robustness. LiveCodeBench approaches near-zero for all models by Round 5 regardless of strategy, confirming that recursive self-training systematically degrades harder, out-of-distribution tasks.

N Implementation Details

Data generation and filtering. For exogenous-gate methods, we generate samples in batches of 2000 until 5000 samples pass the filter. Thus, 10 recursive rounds correspond to 30k training steps.

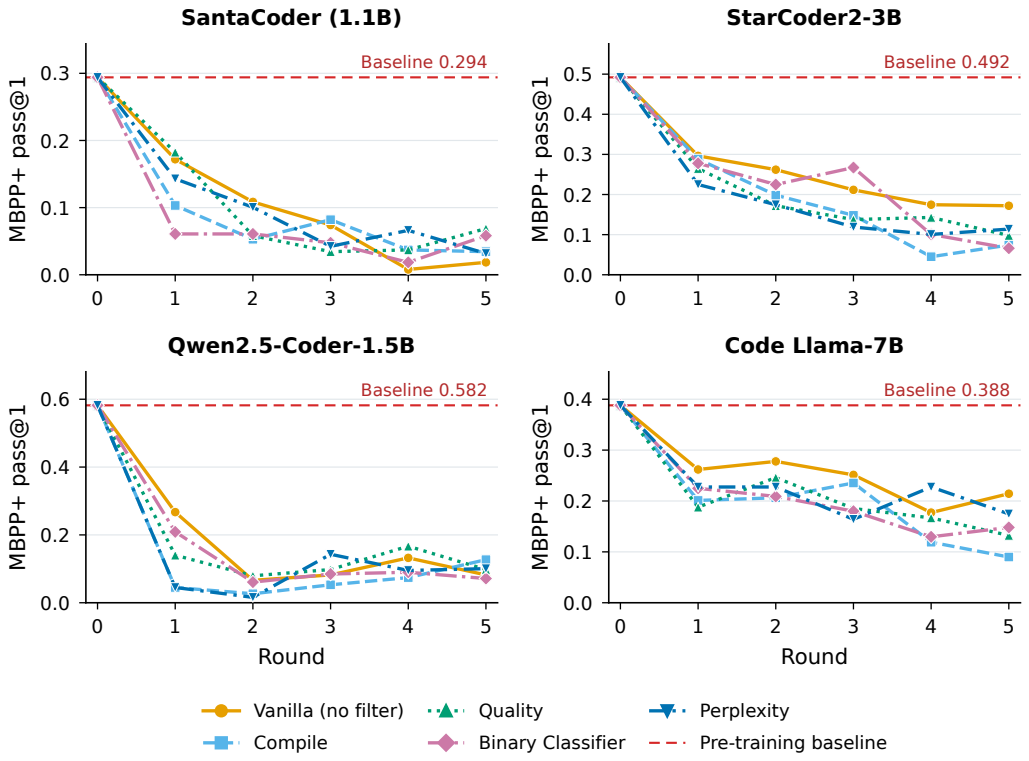

Figure 10: **MBPP+ collapse across all models and filter strategies.** Each subplot shows one model; all five filter trajectories from Round 0 (baseline) to Round 5. Red dashed horizontal lines mark model-specific pre-training baselines. MBPP+ degrades more severely than HumanEval+ for all models, particularly SantaCoder and Qwen2.5-Coder.

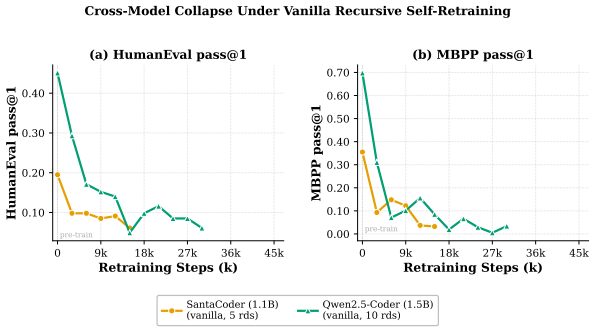

Figure 11: **Cross-model HumanEval and MBPP collapse under vanilla recursive self-retraining.** Per-round trajectories degrade from their pre-retraining baselines across model families.

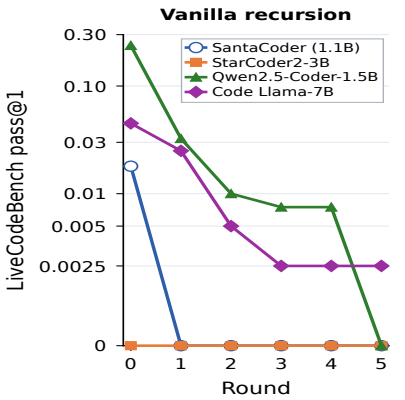

Figure 12: **LiveCodeBench collapse under Vanilla self-training.** All four models approach near-zero pass@1 by Round 5, providing additional evidence beyond HumanEval and MBPP.

Evaluation and hardware. For execution-pass analysis, we evaluate 500 generated samples with a 5s timeout. All experiments are conducted on NVIDIA RTX 6000 Ada Generation GPUs.

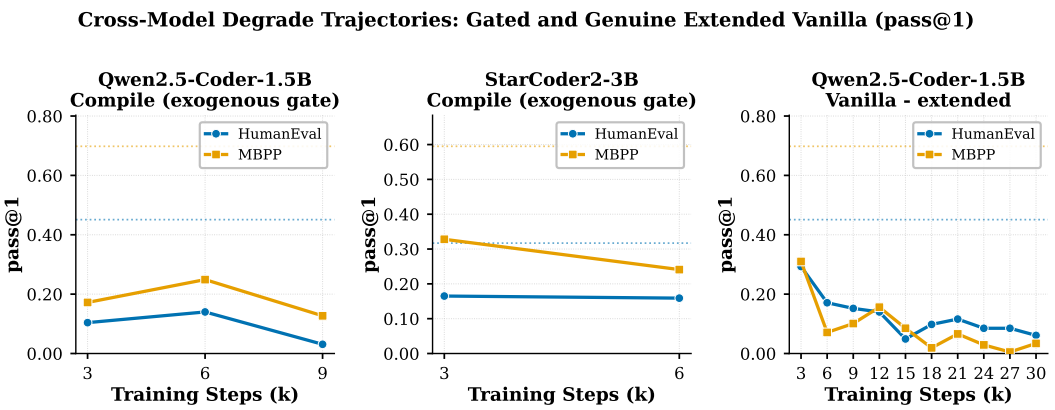

Figure 13: **Cross-model degradation trajectories for gated and extended vanilla runs.** The panels report compile-gated runs for Qwen2.5-Coder-1.5B and StarCoder2-3B, plus the extended vanilla run for Qwen2.5-Coder-1.5B. Blue curves denote HumanEval, orange curves denote MBPP, and dotted lines denote pre-retraining baselines.

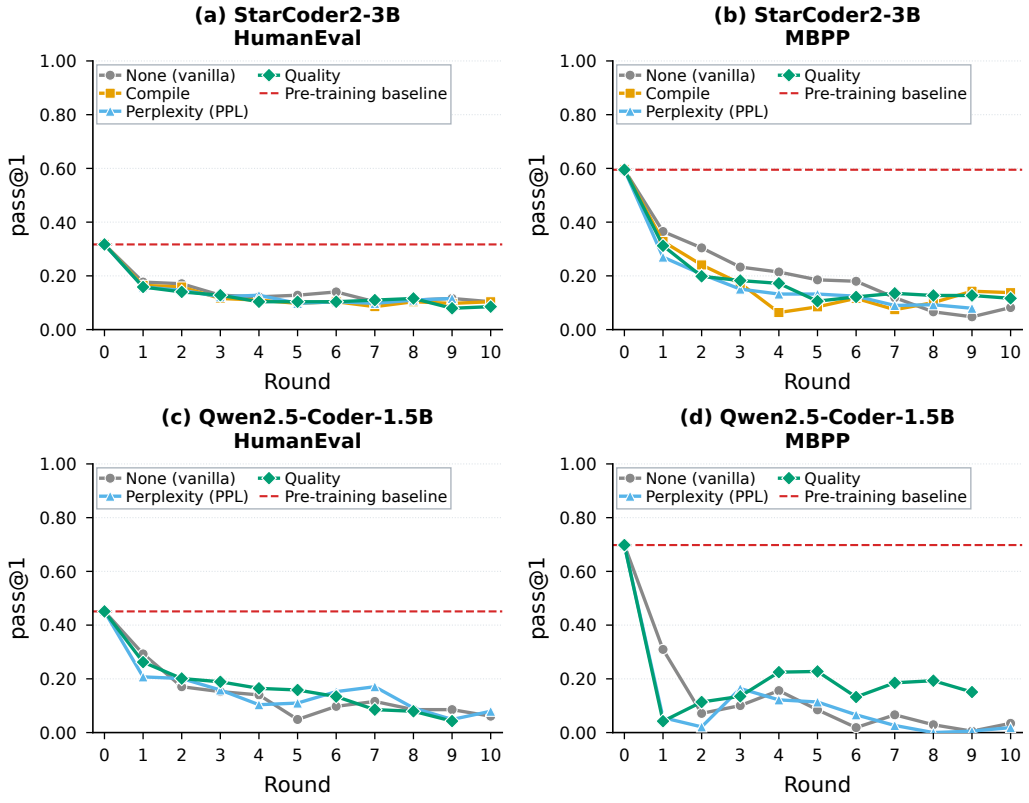

Figure 14: **Per-strategy collapse trajectories across models.** HumanEval and MBPP `pass@1` over recursive self-training rounds for StarCoder2-3B (top row) and Qwen2.5-Coder-1.5B (bottom row) under all filtering strategies. Dashed red line indicates each model's pre-training baseline. All strategies converge well below the pre-training baseline by round 5.

O PER-ROUND FULL RESULTS TABLES (ALL MODELS)

This appendix provides complete per-round results for all four models across all five filtering strategies and five rounds. All metrics are pass@1. The full results appear in Tables 11, 12, 13, and 14.

O.1 SANTACODER (1.1B)

Table 11: SantaCoder (1.1B): full per-round results. FPR = filter pass rate.

Filter	Round	HE	HE+	MBPP	MBPP+	LCB	Train Loss	FPR
–	R0	0.189	0.171	0.352	0.294	0.018	–	–
Vanilla	R1	0.0976	0.0915	0.2011	0.1720	0.000	0.0601	1.000
Vanilla	R2	0.1037	0.0915	0.1243	0.1085	0.000	0.0549	1.000
Vanilla	R3	0.0854	0.0732	0.0794	0.0741	0.000	0.0517	1.000
Vanilla	R4	0.0732	0.0671	0.0106	0.0079	0.000	0.0478	1.000
Vanilla	R5	0.0793	0.0732	0.0265	0.0185	0.000	0.0506	1.000
Compile	R1	0.1098	0.0915	0.1217	0.1032	0.000	0.0534	1.000
Compile	R2	0.0915	0.0671	0.0688	0.0529	0.000	0.0518	1.000
Compile	R3	0.0610	0.0549	0.1005	0.0820	0.000	0.0507	1.000
Compile	R4	0.1037	0.0976	0.0503	0.0370	0.000	0.0571	1.000
Compile	R5	0.0915	0.0793	0.0450	0.0344	0.000	0.0537	1.000
Quality	R1	0.1159	0.0915	0.2116	0.1825	0.000	0.0488	1.000
Quality	R2	0.0976	0.0732	0.0635	0.0582	0.000	0.0564	1.000
Quality	R3	0.0854	0.0671	0.0397	0.0344	0.000	0.0517	1.000
Quality	R4	0.0976	0.0671	0.0476	0.0370	0.000	0.0498	1.000
Quality	R5	0.0671	0.0610	0.0794	0.0688	0.000	0.0435	1.000
PPL	R1	0.1037	0.0793	0.1455	0.1429	0.000	0.0389	0.167
PPL	R2	0.0976	0.0732	0.1164	0.1005	0.003	0.0383	0.201
PPL	R3	0.1098	0.0732	0.0476	0.0423	0.000	0.0332	0.220
PPL	R4	0.0976	0.0732	0.0847	0.0661	0.000	0.0382	0.229
PPL	R5	0.0915	0.0610	0.0476	0.0317	0.000	0.0357	0.235
Binary	R1	0.1341	0.1098	0.0767	0.0608	0.000	0.0618	0.250
Binary	R2	0.1098	0.0793	0.0926	0.0608	0.000	0.0537	0.250
Binary	R3	0.0793	0.0732	0.0582	0.0476	0.000	0.0553	0.250
Binary	R4	0.0732	0.0610	0.0265	0.0185	0.000	0.0526	0.250
Binary	R5	0.1159	0.1037	0.0794	0.0582	0.000	0.0115	0.250

O.2 STARCODER2-3B

O.3 QWEN2.5-CODER-1.5B

O.4 CODE LLAMA-7B

Table 12: StarCoder2-3B: full per-round results.

Filter	Round	HE	HE+	MBPP	MBPP+	LCB	Train Loss	FPR
–	R0	0.317	0.274	0.595	0.492	0.000	–	–
Vanilla	R1	0.1768	0.1646	0.3651	0.2963	0.000	0.2005	1.000
Vanilla	R2	0.1707	0.1524	0.3042	0.2619	0.000	0.1366	1.000
Vanilla	R3	0.1280	0.1159	0.2328	0.2116	0.000	0.1169	1.000
Vanilla	R4	0.1220	0.1220	0.2143	0.1746	0.000	0.1026	1.000
Vanilla	R5	0.1280	0.1037	0.1852	0.1720	0.000	0.1013	1.000
Compile	R1	0.1646	0.1402	0.3280	0.2884	0.005	0.0480	1.000
Compile	R2	0.1585	0.1341	0.2407	0.1984	0.003	0.0259	1.000
Compile	R3	0.1159	0.0976	0.1720	0.1481	0.003	0.0410	1.000
Compile	R4	0.1098	0.0976	0.0635	0.0450	0.000	0.0393	1.000
Compile	R5	0.0976	0.0854	0.0847	0.0741	0.000	0.0307	1.000
Quality	R1	0.1585	0.1280	0.3122	0.2646	0.008	0.0519	1.000
Quality	R2	0.1402	0.1098	0.1984	0.1720	0.005	0.0409	1.000
Quality	R3	0.1280	0.1037	0.1825	0.1376	0.000	0.0271	1.000
Quality	R4	0.1037	0.0793	0.1720	0.1429	0.000	0.0338	1.000
Quality	R5	0.1037	0.0915	0.1058	0.0979	0.003	0.0330	1.000
PPL	R1	0.1585	0.1341	0.2698	0.2249	0.005	0.0385	0.244
PPL	R2	0.1463	0.1159	0.2063	0.1746	0.000	0.0342	0.227
PPL	R3	0.1220	0.1037	0.1508	0.1190	0.000	0.0309	0.232
PPL	R4	0.1280	0.0915	0.1323	0.1005	0.000	0.0287	0.237
PPL	R5	0.0976	0.0854	0.1323	0.1138	0.003	0.0121	0.242
Binary	R1	0.2317	0.1951	0.3016	0.2778	0.005	0.0454	0.250
Binary	R2	0.1768	0.1524	0.2566	0.2249	0.000	0.0401	0.250
Binary	R3	0.1524	0.1159	0.2910	0.2672	0.000	0.0137	0.250
Binary	R4	0.1341	0.1098	0.1190	0.1005	0.000	0.0381	0.250
Binary	R5	0.1402	0.1220	0.0899	0.0661	0.000	0.0397	0.250

Table 13: Qwen2.5-Coder-1.5B: full per-round results.

Filter	Round	HE	HE+	MBPP	MBPP+	LCB	Train Loss	FPR
–	R0	0.451	0.372	0.698	0.582	0.238	–	–
Vanilla	R1	0.2927	0.2622	0.3095	0.2672	0.033	0.1091	1.000
Vanilla	R2	0.1707	0.1280	0.0714	0.0661	0.010	0.0338	1.000
Vanilla	R3	0.1524	0.1159	0.1005	0.0820	0.008	0.0463	1.000
Vanilla	R4	0.1402	0.1280	0.1561	0.1323	0.008	0.0445	1.000
Vanilla	R5	0.0488	0.0427	0.0847	0.0820	0.000	0.0403	1.000
Compile	R1	0.1463	0.1220	0.0503	0.0450	0.030	0.0467	1.000
Compile	R2	0.1524	0.1341	0.0317	0.0265	0.008	0.0423	1.000
Compile	R3	0.0793	0.0732	0.0608	0.0529	0.008	0.0441	1.000
Compile	R4	0.1707	0.1341	0.0820	0.0741	0.005	0.0458	1.000
Compile	R5	0.1037	0.0976	0.1481	0.1270	0.003	0.0418	1.000
Quality	R1	0.2195	0.1768	0.1561	0.1402	0.053	0.0501	1.000
Quality	R2	0.2012	0.1646	0.0873	0.0794	0.013	0.0402	1.000
Quality	R3	0.1463	0.1220	0.1217	0.0979	0.015	0.0397	1.000
Quality	R4	0.1220	0.0915	0.1958	0.1667	0.008	0.0287	1.000
Quality	R5	0.0976	0.0854	0.1243	0.0979	0.008	0.0302	1.000
PPL	R1	0.2073	0.1768	0.0556	0.0450	0.020	0.0287	0.176
PPL	R2	0.2012	0.1585	0.0212	0.0159	0.003	0.0259	0.186
PPL	R3	0.1585	0.1341	0.1640	0.1429	0.000	0.0276	0.196
PPL	R4	0.1037	0.0793	0.1217	0.0952	0.008	0.0267	0.218
PPL	R5	0.1098	0.0854	0.1138	0.1005	0.005	0.0277	0.236
Binary	R1	0.1646	0.1402	0.2460	0.2090	0.043	0.0448	0.250
Binary	R2	0.1646	0.1341	0.0714	0.0608	0.005	0.0459	0.250
Binary	R3	0.1585	0.1159	0.1217	0.0847	0.013	0.0102	0.250
Binary	R4	0.0976	0.0854	0.1032	0.0899	0.000	0.0439	0.250
Binary	R5	0.1463	0.1220	0.0820	0.0714	0.000	0.0374	0.250

Table 14: Code Llama-7B: full per-round results.

Filter	Round	HE	HE+	MBPP	MBPP+	LCB	Train Loss	FPR
–	R0	0.328	0.287	0.441	0.388	0.045	–	–
Vanilla	R1	0.2256	0.1829	0.2989	0.2619	0.025	0.0383	1.000
Vanilla	R2	0.2073	0.1890	0.3492	0.2778	0.005	0.0327	1.000
Vanilla	R3	0.1890	0.1829	0.2857	0.2513	0.003	0.0328	1.000
Vanilla	R4	0.1341	0.1220	0.2037	0.1772	0.003	0.0264	1.000
Vanilla	R5	0.1768	0.1341	0.2354	0.2143	0.003	0.0276	1.000
Compile	R1	0.2195	0.1768	0.2302	0.2011	0.008	0.0346	1.000
Compile	R2	0.2012	0.1707	0.2354	0.2063	0.000	0.0318	1.000
Compile	R3	0.1768	0.1524	0.2725	0.2354	0.003	0.0268	1.000
Compile	R4	0.1707	0.1341	0.1349	0.1190	0.005	0.0270	1.000
Compile	R5	0.1890	0.1707	0.1243	0.0899	0.003	0.0284	1.000
Quality	R1	0.2012	0.1768	0.2143	0.1878	0.033	0.0359	1.000
Quality	R2	0.2256	0.1829	0.3042	0.2460	0.010	0.0321	1.000
Quality	R3	0.1707	0.1402	0.2328	0.1852	0.005	0.0290	1.000
Quality	R4	0.1707	0.1402	0.2037	0.1667	0.010	0.0056	1.000
Quality	R5	0.1220	0.1098	0.1429	0.1323	0.005	0.0278	1.000
PPL	R1	0.2012	0.1646	0.2963	0.2275	0.010	0.0286	0.220
PPL	R2	0.2012	0.1707	0.2804	0.2275	0.018	0.0231	0.245
PPL	R3	0.2134	0.1646	0.1958	0.1640	0.000	0.0216	0.247
PPL	R4	0.1890	0.1585	0.2884	0.2275	0.008	0.0216	0.246
PPL	R5	0.1646	0.1341	0.2063	0.1746	0.000	0.0212	0.248
Binary	R1	0.2012	0.1524	0.2778	0.2249	0.030	0.0240	0.250
Binary	R2	0.1890	0.1524	0.2646	0.2090	0.005	0.0325	0.250
Binary	R3	0.2256	0.1829	0.1958	0.1799	0.008	0.0190	0.250
Binary	R4	0.1890	0.1463	0.1481	0.1296	0.010	0.0276	0.250
Binary	R5	0.1829	0.1585	0.1667	0.1481	0.003	0.0184	0.250