

SpecForge: Recursive Self-Improvement for Coding Agents via Behavioral Specifications

Yutong Cheng^{1*} Haifeng Chen² Peng Gao¹ Wei Cheng^{2✉}

¹Virginia Tech ²NEC Laboratories America

{yutongcheng, penggao}@vt.edu {haifeng, weicheng}@nec-labs.com

Abstract

Coding agents based on frozen language models can improve their performance by refining the external context they rely on, without updating model weights. In this paper, we study *recursive self-improvement (RSI)* through *behavioral specifications*: natural-language code documentation that describes the behavior of program entities and guides agents in generating correct code. We formulate a self-improvement loop in which an agent generates code from its current specifications, evaluates the generated code using execution-verified tests, and refines the specifications based on observed failures. The improved specifications then guide subsequent code generation, allowing the agent to progressively improve its understanding of the codebase. However, recursive refinement is challenging because program entities are interdependent. Improving the specification of one entity can inadvertently degrade the correctness of others, a phenomenon we call *output coupling*. This creates a *whack-a-mole* effect in which local improvements cause regressions elsewhere, preventing consistent progress. We introduce SPECFORGE, a multi-frontier tree search framework that addresses this challenge through three mechanisms: (i) Pareto-guided state selection preserves complementary specification states and prevents premature commitment; (ii) dependency-constrained entity selection prioritizes callee-before-caller refinement; and (iii) error-diversified expansion explores multiple refinements using different subsets of test failures. On DEVEVAL+, SPECFORGE achieves the best performance across five backbone models, including a 94.9% solve rate with Claude-4.5-Sonnet and a 5.7-percentage-point average improvement over the strongest baseline. The improved specifications also make the agent better at tasks outside the loop: they transfer to cross-language translation and to new-feature implementation, where they outperform the benchmark-provided requirement documents. The code and data are available at <https://github.com/ccsnow127/SpecForge>.

1 Introduction

Large language models are increasingly used as autonomous coding agents. Systems such as Cursor [1], Codex [2], and Claude Code [3] can generate code, implement new features, and modify existing software with limited human intervention. Their effectiveness, however, depends heavily on how well they understand the behavior of the codebase they work with. This understanding is essential for tasks such as repository-level code generation [4, 5], legacy system migration [6], cross-language reimplementations [7, 8], and new-feature implementation [9]. One way to provide such understanding is through *behavioral specifications*: natural-language code documentation that describes what individual program entities (e.g., functions, classes, and methods) do, including their inputs, outputs, and important behavioral constraints. Unlike traditional documentation intended

*Work done during the internship at NEC Laboratories America. ✉Corresponding author.

primarily for human readers, these specifications serve as actionable knowledge that coding agents can use to generate and modify code. Importantly, because specifications are expressed in natural language, agents can also revise them based on feedback, creating an opportunity for self-improvement without changing the underlying language model.

This motivates *specification-mediated recursive self-improvement (RSI)*, in which a coding agent iteratively improves its own behavioral specifications using feedback from code generation and execution, without modifying the underlying model weights. Starting from initial specifications, the agent generates code conditioned on these specifications alone and evaluates the generated implementations through unit tests. Test failures reveal missing or ambiguous behavioral information, which the agent uses to refine the specifications. The refined specifications then guide the next round of code generation, forming a recursive self-improvement loop. In our setting, the tests are automatically generated from the reference implementation and validated through execution, retaining only tests that pass. This provides reliable, execution-grounded feedback without requiring human annotations. Importantly, the objective is not to improve the reconstructed code itself, but to produce more accurate behavioral specifications. Their quality is measured not by readability or similarity to human-written documentation [10, 11], but by whether an agent can reconstruct behaviorally correct code from them. The resulting specifications capture reusable knowledge of the existing codebase, which can support downstream tasks such as cross-language translation and new-feature implementation, even when no reference implementation of the desired output is available.

Despite its promise, specification-based RSI faces a fundamental challenge: *program entities are not independent*. Existing language-space optimization methods, such as MIPROv2 [12], GEPA [13], and ACE [14], demonstrate that language models can improve their own context through iterative refinement and feedback [15, 16]. However, directly applying this approach to behavioral specifications is difficult because changes to one entity can affect the behavior of others. We refer to this phenomenon as *output coupling*. It arises in two ways. First, *dependency coupling* occurs when the correctness of a caller depends on the behavior of its callees. Refining a caller before its callees are correctly specified can therefore produce misleading feedback. Second, *generation coupling* occurs because a coding model may condition on the entire specification bundle, so modifying one specification can change how other entities are generated, even without direct dependencies. Consequently, an update that fixes one entity may break another, creating a *whack-a-mole* effect: improvements are repeatedly undone by subsequent refinements, and the optimization process may oscillate rather than converge. For recursive self-improvement to be effective, it must therefore address three requirements: preserving previous progress, respecting dependencies between entities, and handling noisy or incomplete feedback during refinement.

To address these challenges, we propose SPECFORGE, a *multi-frontier tree search* framework for recursive self-improvement through behavioral specifications. Instead of following a single sequence of specification revisions, SPECFORGE explores multiple candidate specification states through three complementary mechanisms: ❶ **Where to search: Pareto-guided state selection** maintains an archive of non-dominated specification states with complementary strengths across program entities. This preserves alternative refinement paths and prevents premature commitment to a single solution. ❷ **What to refine: Dependency-constrained entity selection** uses the program’s call graph to restrict refinement to entities whose callees have already been resolved. A discounted-UCB bandit then adaptively allocates the refinement budget among eligible entities. ❸ **How to refine: Error-diversified expansion** generates multiple refinement candidates from different subsets of failing tests, allowing the search to explore alternative explanations for observed failures and reducing sensitivity to noisy feedback. Together, these mechanisms mitigate refinement interference and enable improvements to accumulate across recursive iterations.

We evaluate SPECFORGE on DEVVAL+, where behavioral specification quality is measured by the correctness of code reconstructed from the generated specifications. Across five backbone language models, SPECFORGE consistently outperforms strong documentation, prompt-optimization, and coding-agent baselines. In particular, it achieves a 94.9% solve rate with Claude-4.5-Sonnet and improves over the strongest baseline by 5.7 percentage points on average. We further evaluate whether the optimized specifications remain useful beyond the original code-generation task. Results on cross-language translation and new-feature implementation show that these specifications provide reusable behavioral knowledge that improves coding-agent performance in different software engineering settings. These findings suggest that refining behavioral specifications provides a practical approach to recursive self-improvement for coding agents, without requiring model weight updates.

Our main contributions are summarized as follows:

- We study *specification-mediated recursive self-improvement* for coding agents, in which behavioral specifications are iteratively improved through code generation and execution feedback. We identify *output coupling* as a central challenge that prevents naive iterative refinement from accumulating progress.
- We introduce SPECFORGE, a dependency-guided multi-frontier tree search framework that combines Pareto-guided state selection, dependency-constrained entity selection, and error-diversified expansion to mitigate refinement interference and enable cumulative specification improvement.
- We demonstrate that SPECFORGE consistently outperforms strong baselines on DEVEVAL+ across five backbone models, and that its optimized specifications improve downstream cross-language translation and new-feature implementation.

2 Problem Formulation

We study *behavioral specification optimization*: given source code, the goal is to produce, for every program entity, a natural-language specification from which a code-generation model can reconstruct a correct implementation without seeing the code. Unlike documentation, whose objective is human readability, a specification is judged by *downstream behavioral utility*—whether it contains enough precise semantic information for an LLM to synthesize correct code.

Consider a code module with n program entities $\mathcal{E} = \{e_1, \dots, e_n\}$, such as functions, classes, or methods. Let $G = (\mathcal{E}, E)$ be the call graph, where $(e_i, e_j) \in E$ indicates that entity e_i invokes entity e_j . For evaluation, we assume a set of unit tests $\mathcal{T}$, and each test $t \in \mathcal{T}$ is attributed to a target entity through a mapping $\alpha : \mathcal{T} \rightarrow \mathcal{E}$. We denote by $\mathcal{T}_i = \{t \in \mathcal{T} : \alpha(t) = e_i\}$ the tests associated with entity e_i . Let $\mathcal{D} = (d_1, \dots, d_n)$ denote a specification bundle, where d_i is the specification for entity e_i . Given a code generation model $\mathcal{G}$, the generated implementation for entity e_i is $c_i = \mathcal{G}(\mathcal{D}, e_i)$, and the full generated module is $\mathcal{C}(\mathcal{D}) = (c_1, \dots, c_n)$.

Definition 1 (Behavioral Specification Optimization). *The quality of the specification for entity e_i is measured by its entity pass rate $\phi_i(\mathcal{D}) = \frac{1}{|\mathcal{T}_i|} \sum_{t \in \mathcal{T}_i} \mathbb{1}[\text{PASS}(c_i, t)]$. An entity e_i is resolved under $\mathcal{D}$ if $\phi_i(\mathcal{D}) = 1$. We write the resolved set as $\mathcal{S}(\mathcal{D}) = \{e_i \in \mathcal{E} : \phi_i(\mathcal{D}) = 1\}$. The goal is to find the specification bundle that maximizes the number of resolved entities: $\mathcal{D}^* = \arg \max_{\mathcal{D}} |\mathcal{S}(\mathcal{D})|$.*

In addition to the solved-set objective, we also use the pass-rate vector $\phi(\mathcal{D}) = (\phi_1(\mathcal{D}), \dots, \phi_n(\mathcal{D}))$ to capture fine-grained per-entity progress, and the module-level pass rate $\bar{\phi}(\mathcal{D}) = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \mathbb{1}[\text{PASS}(c_{\alpha(t)}, t)]$ to measure overall behavioral correctness. This formulation yields a black-box optimization problem over the space of natural-language specifications: a candidate specification bundle can only be judged through generated code and execution outcomes, while test failures provide structured feedback about what the specifications failed to make explicit.

Output Coupling. The central challenge is that program entities cannot be optimized independently. First, *dependency coupling* arises because if e_i calls e_j , then the correctness of c_i depends on the realized behavior of c_j . Second, *generation coupling* arises because the code generator conditions on the full specification bundle $\mathcal{D}$, so changing d_j may alter the generated code for e_i even without a direct call edge. Together, these effects produce a *whack-a-mole* phenomenon: refining the specification of one entity may improve that entity while destabilizing others. Formal definitions are deferred to Appendix §A.

Search Formulation. The optimization is difficult because specification quality is only observable *after* code generation and execution. This makes behavioral specification optimization a structured search problem over specification states. Different partial solutions resolve different subsets of entities, and these resolved subsets determine which subsequent refinements are meaningful under dependency constraints.

3 Proposed Method

We propose SPECFORGE, a *multi-frontier tree search* framework for behavioral specification optimization. SPECFORGE instantiates the self-improvement loop of §1 as search: each node is a specification state evaluated by executing the code generated from it, each edge is one self-revision driven by the failures of that code, and every revision starts from a state that earlier revisions produced.

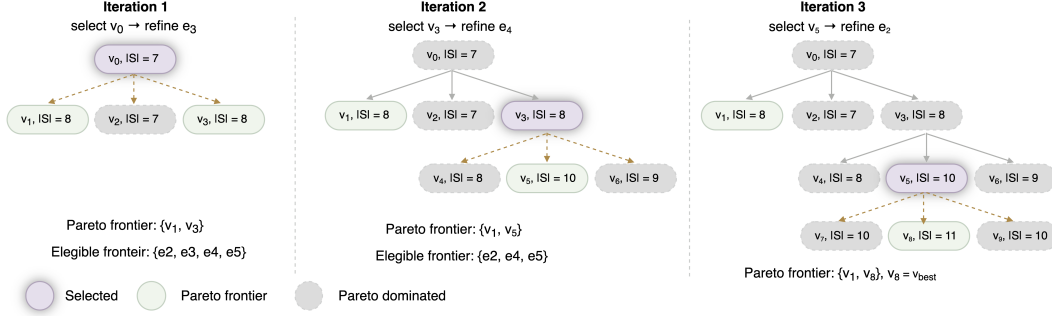


Figure 1: **Search tree view of behavioral specification optimization.** Each node is a specification state. Different states solve different subsets of entities and therefore expose different eligible next refinements. Rather than committing to a single rewrite trajectory, our solution preserves multiple competitive frontiers and expands them selectively.

Figure 1 illustrates the search process: specification optimization proceeds not as a single rewrite trajectory, but as search over multiple partial solutions with different future refinement opportunities. By managing this search so that local refinements accumulate into stable global progress, SPECFORGE directly targets the whack-a-mole behavior induced by output coupling.

State Representation. SPECFORGE organizes optimization as tree search over specification states. A node is $v = (\mathcal{D}_v, \phi_v, \mathcal{S}_v)$, where $\mathcal{D}_v$ is the current specification bundle, $\phi_v \in [0, 1]^n$ is the per-entity pass-rate vector, and $\mathcal{S}_v$ is the set of resolved entities. An edge $v \xrightarrow{e} v'$ denotes refining exactly one entity e :

$$d'_e \sim P_\theta(\cdot \mid d_e, \mathcal{F}_e(v)), \quad d'_i = d_i \quad \forall i \neq e, \quad (1)$$

where $\mathcal{F}_e(v)$ denotes the failure feedback for entity e at state v . This single-entity transition is the basic search step in SPECFORGE: it keeps refinements localized, makes feedback easier to attribute, and allows the method to explore multiple competing partial solutions in parallel.

Multi-Frontier Tree Search. Starting from a root node v_0 corresponding to the initial specifications, SPECFORGE maintains a Pareto archive $\mathcal{A}$ of non-dominated states (initialized as $\{v_0\}$) together with a shared entity-selection bandit. Each iteration resolves three coupled decisions: *where* to search (§3.1), *what* to refine (§3.2), and *how* to refine (§3.3). Concretely, the iteration (i) samples a parent node v from $\mathcal{A}$; (ii) selects an eligible entity e to refine via a discounted-UCB policy over per-entity gap-closing rates; and (iii) expands v into W children, each obtained by substituting a refined specification d'_e into $\mathcal{D}_v$, regenerating the code of the full module from the updated specifications, and re-evaluating the full test suite to produce $\phi_{v'}$. The W children are then folded back into both upper levels: non-dominated children enter $\mathcal{A}$, introducing new frontier diversity for future state selection, and the best child’s $\Delta\phi_e$ updates the bandit for future entity selection. Once B iterations, SPECFORGE returns the specification bundle $\mathcal{D}_v$ achieving the largest $|\mathcal{S}|$ over all visited states. Figure 2 and Algorithm 1 summarize the procedure.

3.1 Pareto-Guided State Selection

The first step is to decide where to search next, i.e., which previously visited state to expand. Because improving one entity may degrade another, scalar ranking is often too coarse. We therefore maintain an archive $\mathcal{A}$ of non-dominated states under coordinate-wise dominance over pass-rate vectors:

$$\phi_{v'} \succeq \phi_v \iff \phi_i(\mathcal{D}_{v'}) \geq \phi_i(\mathcal{D}_v) \text{ for all } i. \quad (2)$$

A new child enters the archive if no existing state dominates it; any states that it dominates are removed. This archive preserves *complementary frontier states*. One state may already solve low-level utility entities, while another may have progressed further on higher-level orchestration logic. Crucially, different archive states induce different feasible next refinements because they solve different subsets of entities. Maintaining multiple such states prevents premature commitment to one unstable trajectory and thus helps to escape local optima (refer to §B.1 for detailed analysis).

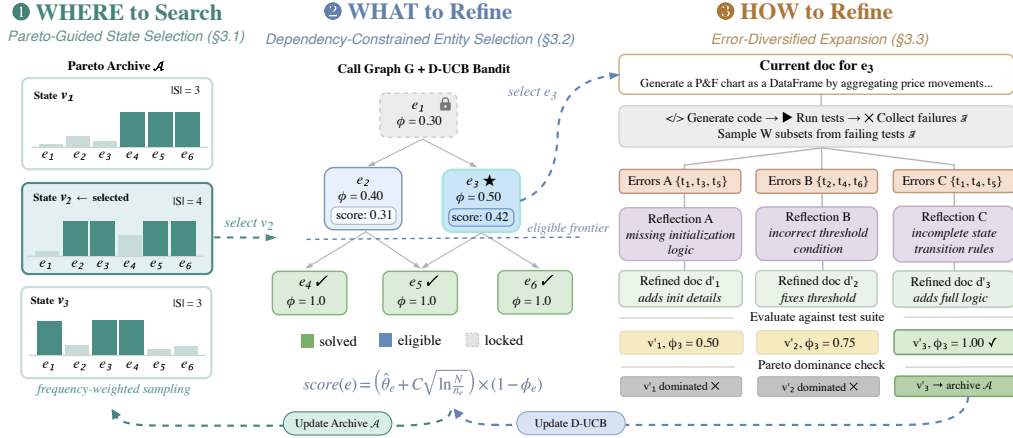


Figure 2: **Overview of SPECFORGE.** Each iteration resolves three decisions: *where* to search, *what* to refine, and *how* to refine. ❶ **Pareto-guided state selection** samples from an archive of non-dominated specification states with complementary strengths. ❷ **Dependency-constrained entity selection** only refines entities whose callees are already solved. ❸ **Error-diversified expansion** generates multiple children from different subsets of failing tests. Strong children are fed back into both the archive and the entity-selection policy.

Frequency-Weighted Sampling. To select which archive node to expand, we sample from the Pareto front of $\mathcal{A}$ (Algorithm 2). For each entity e_i , we identify the archive node(s) attaining the highest ϕ_i and assign credit to those node(s), splitting it evenly in the case of ties. A node’s sampling weight is the total credit it accumulates across entities. This biases expansion toward nodes that are frontier-best on more entity-wise dimensions, thereby preserving complementary search frontiers rather than collapsing prematurely to a single trajectory.

3.2 Dependency-Constrained Entity Selection

Given a selected state v , the second step decides *what* entity to refine. Refining a caller before its callees are stabilized often yields misleading feedback. We therefore restrict candidates to an *eligibility frontier*:

$$\mathcal{E}_{\text{elig}}(v) = \{e \in \mathcal{E} : e \notin \mathcal{S}_v, \forall (e, e_j) \in E, e_j \in \mathcal{S}_v\}. \quad (3)$$

That is, an entity is eligible only if it remains unsolved and all of its callees are already resolved. This enforces a callee-before-caller refinement order and mitigates dependency-induced interference. Recursive calls are handled by contracting each strongly connected component into a jointly refined meta-entity (Appendix C.1).

When multiple entities are eligible, SPECFORGE allocates budget adaptively using a discounted UCB policy [17]. The key reward is normalized by the remaining gap: $\hat{r}_e = \frac{\Delta\phi_e}{1-\phi_e(\mathcal{D}_v)}$, where $\Delta\phi_e$ is the observed improvement after refining entity e at the selected state v , and $\phi_e(\mathcal{D}_v)$ is the pass rate of e at v . This separates intrinsic repairability from the trivial effect of having a large remaining error. The resulting score is

$$\text{score}(e) = \underbrace{\left(\hat{\theta}_e + C \sqrt{\frac{\ln N_\gamma}{n_\gamma(e)}} \right)}_{\text{UCB estimate of gap-closing rate}} \times \underbrace{(1 - \phi_e(\mathcal{D}_v))}_{\text{remaining gap}}, \quad (4)$$

where $n_\gamma(e)$ is the discounted pull count of entity e , $N_\gamma = \sum_{e'} n_\gamma(e')$ is the total discounted pull count over all entities, and $\hat{\theta}_e$ is the discounted empirical mean of $\hat{r}_e$, i.e., the estimated gap-closing rate of refining e under the current search regime (detailed in Appendix C.2). The bandit maintains a single set of statistics shared across all state nodes, with discounting ensuring that estimates remain responsive as the search context evolves. The policy therefore balances uncertainty, past success, and remaining improvement potential.

3.3 Error-Diversified Expansion

Given a selected state v and entity e , the third step expands v into W child states, each representing a distinct refinement of d_e . The key challenge is that failing tests are diagnostically heterogeneous:

some directly expose the behavioral detail missing from the specification, while others are redundant, weakly informative, or induced by cascading dependency errors. This is consistent with prior findings that test cases differ substantially in fault-localization value, LLM-generated tests often cluster around similar patterns, and adding more tests can even hurt diagnosis when uninformative cases dilute the signal [18, 19, 20]. More broadly, LLMs are known to be sensitive to irrelevant or noisy context in both in-context learning and RAG settings [21, 22, 23].

Formally, let $\mathcal{F}_e(v)$ denote the failing tests for entity e at state v , and suppose it admits an unknown partition $\mathcal{F}_e(v) = \mathcal{F}_e^+ \cup \mathcal{F}_e^-$, where $\mathcal{F}_e^+$ contains informative failures and $\mathcal{F}_e^-$ noisy ones. Let $q(\mathbf{e}) = P(\text{successful refinement} \mid \mathbf{e})$ denote the success probability when conditioning on an error batch $\mathbf{e}$. The key point is that noise is not merely unhelpful but actively corrupting: under mild assumptions, mixing noisy failures into an informative batch strictly degrades refinement quality, i.e., for any nonempty $\mathbf{e}^+ \subseteq \mathcal{F}_e^+$ and $\mathbf{e}^- \subseteq \mathcal{F}_e^-$, $q(\mathbf{e}^+ \cup \mathbf{e}^-) < q(\mathbf{e}^+)$ when $|\mathbf{e}^-| > 0$ (Appendix §B.2).

Since the partition is unknown, SPECFORGE hedges by sampling W distinct subsets $\{\mathbf{e}_1, \dots, \mathbf{e}_W\}$ from $\mathcal{F}_e(v)$ and generating one refinement candidate per subset. If at least one sampled batch is cleaner than the full failure set, diversified expansion achieves strictly higher *best one-step branch quality* than repeatedly conditioning all W branches on $\mathcal{F}_e(v)$. Formally, if there exists $j^* \in \{1, \dots, W\}$ such that $q(\mathbf{e}_{j^*}) > q(\mathcal{F}_e(v))$, then Appendix §B.3 shows

$$\max_{1 \leq j \leq W} q(\mathbf{e}_j) > \max_{1 \leq j \leq W} q(\mathcal{F}_e(v)) = q(\mathcal{F}_e(v)). \quad (5)$$

The gain arises not from improving every branch, but from increasing the chance that *at least one* branch is conditioned on a diagnostically cleaner signal than the full failure set. Implementation details are given in Appendix C.3.

4 Experiments

Our experiments test the claims behind specification-mediated self-improvement: that improvements accumulate instead of oscillating (§4.2 and §4.3), that each component of SPECFORGE contributes to the stability of the loop (§4.4), and that the improved specifications make the agent better at tasks outside the loop (§4.5).

4.1 Setup

Benchmark. We evaluate SPECFORGE on DEV EVAL+, a benchmark for evaluating behavioral specifications through code reconstruction and execution-based verification across modules with complex inter-entity dependencies. It consists of **20 Python modules**, **247 entities**, and **760 test cases**. The modules are drawn from DevEval [5] and supplemented with additional GitHub repositories to broaden domain coverage, including finance, biological analysis, and signal processing. DEV EVAL+ is structurally nontrivial, with an average dependency depth of 4.2, making it a suitable testbed for studying output coupling under repository-scale generation (full details in Appendix H).

Baselines. We compare SPECFORGE against five groups of baselines:

- *Static documentation methods:* RepoAgent [10] and DocAgent [11], which generate human-oriented documentation without execution feedback; we evaluate their documentation in the specification role.
- *Prompt optimization methods:* MIPROv2 [12], GEPA [13], and ACE [14], which optimize the instruction context used to generate specifications, rewriting the full specification bundle monolithically at each iteration based on code-generation feedback.
- *Commercial coding agents:* Cursor [1] and Claude Code [3], evaluated in four variants: (1) *base*, using default settings; (2) *+Topo*, which supplies reverse topological order as additional context; (3) *+Test*, which prompts the agent to generate tests and iteratively refine code; and (4) *+Topo+Test*, which combines both augmentations.²
- *Iterative refinement baselines:* Sequential refinement with execution feedback, using either random entity selection (ITERATIVE) or reverse topological order (ITERATIVE+TOPO).
- *Behavioral specification optimization:* DOCSEARCH [24], the only baseline that optimizes the same objective, formulated there as agent-oriented documentation generation: the text is optimized for the correctness of code generated from it, under execution feedback. It performs a dependency-

²Cursor supports Gemini-2.5-Flash, Gemini-3.1-Pro, and Claude-4.5-Sonnet, but not Qwen3-235B-A22B or GPT-4o; Claude Code supports only Claude models. Missing entries in Table 1 follow from these platform constraints.

Table 1: Main results on DEVEVAL+.

Method	Qwen3-235B-A22B		Gemini-2.5-Flash		GPT-4o		Gemini-3.1-Pro		Claude-4.5-Sonnet	
	S	$\bar{\phi}$	S	$\bar{\phi}$	S	$\bar{\phi}$	S	$\bar{\phi}$	S	$\bar{\phi}$
<i>Static Methods (No Optimization)</i>										
RepoAgent	22.3	34.9	19.2	30.6	27.8	39.9	33.7	45.5	35.3	47.3
DocAgent	16.1	31.1	14.5	27.2	22.2	37.2	26.8	41.8	28.9	44.7
<i>Prompt Optimization (Monolithic)</i>										
MIPROv2	28.5	30.4	30.3	32.3	29.8	31.4	42.8	43.8	50.0	52.0
GEPA	34.8	35.6	32.1	33.1	35.6	37.6	40.7	42.5	50.4	52.9
ACE	58.9	59.8	61.5	62.4	62.8	64.3	68.6	70.1	74.9	76.6
<i>Coding Agents (Commercial)</i>										
Cursor	—	—	15.8	25.1	—	—	27.5	37.1	30.6	40.0
+ Topo	—	—	21.2	30.6	—	—	37.6	46.6	37.7	46.7
+ Test	—	—	31.0	39.0	—	—	44.9	50.8	47.8	52.3
+ Topo + Test	—	—	39.4	46.3	—	—	51.1	56.4	52.1	56.8
Claude Code	—	—	—	—	—	—	—	—	40.1	50.1
+ Topo	—	—	—	—	—	—	—	—	45.1	55.0
+ Test	—	—	—	—	—	—	—	—	55.3	62.2
+ Topo + Test	—	—	—	—	—	—	—	—	61.9	68.4
<i>Iterative Refinement (Entity-wise)</i>										
Iterative	41.0	45.0	37.8	45.1	49.7	53.8	57.5	59.9	59.5	60.6
+ Topo	43.5	46.8	40.5	46.9	54.3	57.9	67.5	68.8	64.4	64.5
<i>Behavioral Specification Optimization</i>										
DOCSEARCH [†]	81.5	84.9	82.2	85.7	85.5	88.4	87.2	89.5	90.4	91.8
SPECFORGE (ours)	88.9	91.5	89.6	92.1	90.9	93.2	91.0	93.1	94.9	96.1

guided bi-level search that refines entities in reverse topological order with error-diversified beam expansion, and commits a refinement only if no entity regresses. We run the official implementation on DEVEVAL+; see §G for details.

Across all methods, final performance is evaluated with ground-truth test suites that remain hidden from the search procedure. During optimization, methods receive only LLM-generated tests. Additional details on test generation appear in §J, while implementation details and prompts for all baselines are provided in §G. Additional experiments, such as the fidelity of the self-generated improvement signal (§E) and module-level case studies (§D), are included in the Appendix.

Metrics. We report two complementary metrics: **Solve Rate** |S|, the proportion of entities achieving all tests passed, and **Pass Rate** $\bar{\phi}$, the fraction of test cases passed across the entire module. The former measures complete per-entity correctness; the latter provides a finer-grained, continuous measure of progress at the test-case level.

4.2 Main Results

Table 1 presents comprehensive comparisons across methods and base LLMs (50 rollouts). SPECFORGE achieves the highest solve rates across all models: 88.9% with Qwen3-235B-A22B, 89.6% with Gemini-2.5-Flash, 90.9% with GPT-4o, 91.0% with Gemini-3.1-Pro, and 94.9% with Claude-4.5-Sonnet. The results reveal a clear hierarchy of failure modes. **Static methods** (14.5–35.3%) expose a *documentation–specification gap*: documentation optimized for human readability appears imprecise about behavior to serve as a specification for code-synthesis models, which interpret underspecified phrases literally but resolve ambiguities unpredictably. **Monolithic prompt optimizers** improve substantially—ACE reaches 62.8% on GPT-4o via its generate-reflect-curate pipeline—but suffer from *attribution failure* (cannot identify which entity caused a test failure) and *coupling amplification* (a single rewrite alters all entities simultaneously). **Commercial agents** with full augmentation (Cursor+Topo+Test: 52.1%, Claude Code+Topo+Test: 61.9% on Claude-4.5-Sonnet) confirm that topological ordering and test feedback are valuable signals, yet both still trail SPECFORGE by 33.0–42.8%. **Iterative refinement** with topological ordering reaches 54.3% on GPT-4o but plateaus due to the whack-a-mole phenomenon: without Pareto archiving and diversified exploration, correct ordering alone cannot overcome interference at the same topological level. **DOCSEARCH** is the strongest baseline (81.5–90.4%) and the only one that optimizes the same objective. It already orders refinement callee-before-caller and diversifies the error context, but it follows a single search path and commits a refinement only if no entity regresses. SPECFORGE exceeds it on every backbone, by

[†]We re-run DOCSEARCH [24] with its official implementation under the protocol of this paper on the revised DEVEVAL+ release. This release, designed to stress-test SPECFORGE with larger modules, denser test suites, and additional corner cases, thus yields DOCSEARCH scores that deviate modestly (by 5.6 points on average) from those reported in its original paper.

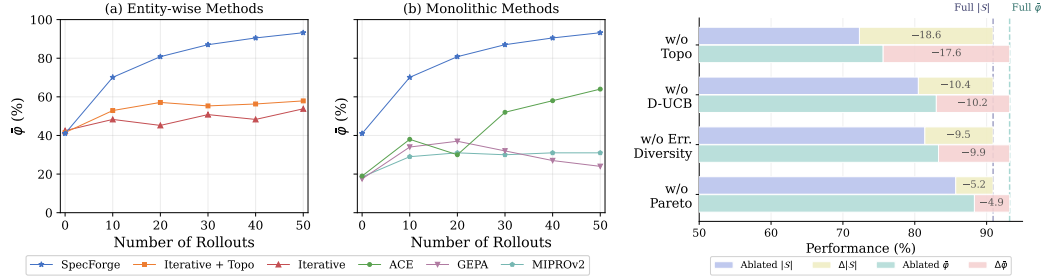


Figure 3: **Convergence analysis** (GPT-4o, DEVEVAL+). (a) Iterative refinement baselines oscillate due to refinement interference. (b) Monolithic methods exhibit catastrophic instability. SPECFORGE achieves steady upward progress in both comparisons. (c) **Ablation study** (GPT-4o, 10-module subset). Components ordered by contribution (large to small); dashed lines mark the full model ($|\mathcal{S}|=90.9\%$, $\phi=93.2\%$). Reverse topological ordering is the largest single contributor (-18.6%).

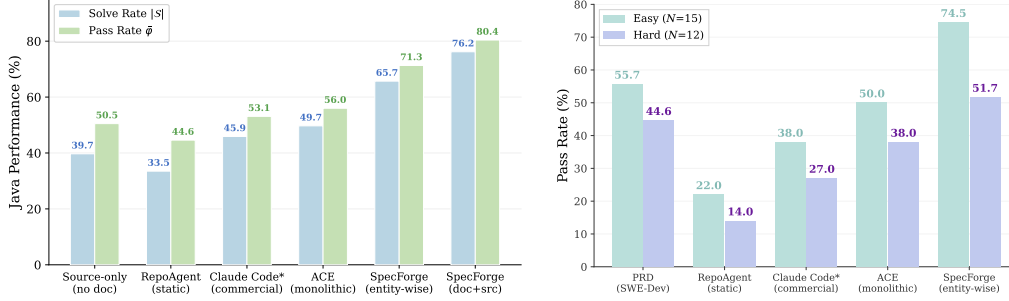


Figure 4: **(Left) Cross-language translation.** Optimized specifications ($+26.0\%$ over Source-only) transfer effectively across languages; combining specifications with source code achieves the best results (76.2%). **(Right) New-feature implementation** on SWE-DEV. SPECFORGE-generated specifications surpass the default PRD provided by the SWE-DEV benchmark, achieving the highest pass rate on both difficulty levels despite never accessing the PRD or target tests.

3.8–7.4 points in solve rate (5.7 on average) and 3.6–6.6 points in pass rate. The margin is largest on the two weakest backbones (7.4 points on both Qwen3-235B-A22B and Gemini-2.5-Flash) and narrows to 3.8–4.5 points on the two strongest, consistent with the role of the Pareto archive: when no candidate passes a no-regression test, a single-path search must abandon the entity, whereas SPECFORGE can continue from another frontier. The improvement pattern is remarkably consistent across LLMs: despite substantial differences in base capability, SPECFORGE delivers absolute solve-rate gains of 23.5–49.1 percentage points over the best iterative baseline across all models, consistent with the hypothesis that the method targets a broadly shared structural difficulty.

4.3 Token Budget Efficiency Analysis

Figure 3 plots cross-module average pass rate $\bar{\phi}$ against budget step, evaluated on ground-truth tests at every 10 rollouts. Among entity-wise methods (Figure 3(a)), Iterative refinement oscillates between 45–54% as random entity selection triggers generation coupling—modifying one entity’s specification perturbs code generation for unrelated entities. Iterative+Topo eliminates the most direct interference by enforcing callee-before-caller order, rising to 57% by $B=20$, but plateaus once all remaining candidates are callers at the same topological level where sibling coupling persists. Among monolithic methods (Figure 3(b)), MIPROv2 plateaus at 31% by $B=20$ as its Bayesian surrogate exhausts the candidate space under single-instance optimization; GEPA peaks at 37% ($B=20$) then declines monotonically to 24% because its graph evolution lacks running-best retention, making exploration into worse regions irreversible; ACE reaches the highest monolithic score (64%) but exhibits repeated regressions (e.g., 38% to 30% between $B=10$ and 20) where each monolithic rewrite simultaneously fixes and breaks entities, recovering only through playbook accumulation of failure-derived rules. SPECFORGE rises steadily from 41% to 93% across both panels: the steepest gains occur in the first 10 rollouts ($+29\%$) as D-UCB exploration combined with topological constraints rapidly solves leaf entities, with diminishing but sustained progress thereafter as the Pareto archive preserves complementary frontier states, converting exploration cost into eventual progress rather than permanent regression.

4.4 Component Contribution Analysis

Figure 3(c) isolates each component’s contribution. **Reverse topological order** (−18.6%) is the structural foundation: without dependency constraints, the search refines callers before their callees are stabilized, allowing interference to propagate through the entire call chain. **D-UCB bandit** (−10.4%) demonstrates that adaptive budget allocation is critical: its gap-closing rate estimate $\hat{\theta}_e$ learns entity-level reparability and directs budget accordingly. **Error diversity** (−9.5%) addresses exploration collapse: diversified batch sampling increases the probability of encountering informative error combinations (Appendix §D). **Pareto archive** (−5.2%) preserves frontier diversity, enabling recovery from suboptimal early refinements; its value increases on harder modules with deeper dependency chains.

4.5 Downstream Utility of Optimized Specifications

We evaluate whether SPECFORGE-optimized specifications transfer beyond the original code-generation setting through two downstream tasks: cross-language translation (§4.5.1) and new-feature implementation (§4.5.2). Both tasks compare the outputs of four methods from the main evaluation—RepoAgent, Claude Code*³, ACE, and SPECFORGE—with GPT-4o as the code-generation backbone.

4.5.1 Cross-Language Translation

Experimental Setup. We evaluate cross-language transfer on DEVEVAL+-JAVA, a JUnit 5 test harness covering a 16-module subset of DEVEVAL+ (208 entities, 680 test cases). We exclude modules whose Python implementations depend on libraries without direct Java equivalents. Each harness mirrors the original Python tests, preserving input data and expected outputs to ensure semantic equivalence (Appendix H.6). We evaluate **Source-only** translation (Python source → Java, no specification) alongside the four methods above. For SPECFORGE we additionally test a **Spec+Source** variant that provides both the specification and the source code.

Results and Analysis. Figure 4(left) reports both solve rate $|\mathcal{S}|$ and pass rate $\bar{\phi}$ for each condition. Source-only translation achieves 39.7% / 50.5% ($|\mathcal{S}| / \bar{\phi}$), while static documentation (RepoAgent, 33.5% / 44.6%) performs *worse* on both metrics—imprecise human-oriented descriptions introduce ambiguities absent from source code. Performance improves steadily with optimization sophistication: Claude Code* (45.9% / 53.1%), ACE (49.7% / 56.0%), and SPECFORGE (65.7% / 71.3%, +26.0% solve rate over Source-only). Per-entity optimization produces precise behavioral specifications describing *what* each function computes rather than *how* it computes in Python, freeing the LLM to choose Java-appropriate implementations. The Spec+Source combination achieves the best results (76.2% / 80.4%), as specifications and source code provide complementary “what” and “how” signals. The quality gradient (static < commercial < monolithic < entity-wise) is preserved across languages, confirming that SPECFORGE’s precision reflects higher-quality behavioral specifications.

4.5.2 New Feature Implementation

Whereas cross-language translation evaluates specifications on *re-implementing* existing code in a different language, real-world development more often requires implementing *new* functionality atop an existing codebase, with specifications of the existing code as guidance. We therefore evaluate on SWE-DEV [9], a benchmark in which an LLM must implement new functions—specified by a Product Requirements Document (PRD)—within real open-source packages, validated against developer-written test suites.

Experimental Setup. We select 10 packages from SWE-DEV, comprising 27 feature-implementation instances (15 easy and 12 hard). For each package, we run all four methods on the *existing* source code to produce behavioral specifications of the current codebase; these specifications are then provided as context when the LLM implements new functions. We compare two conditions, which share the stubbed source and differ only in the accompanying text (Appendix I): (i) **Spec+Source**: stubs paired with the signature and docstring of each target function, plus the specifications of the existing codebase produced by each of the four methods; (ii) **PRD+Source**: stubs paired with the default PRD provided by the SWE-DEV benchmark. We report Pass@1 following SWE-DEV’s evaluation protocol, broken down by difficulty level.

³Claude Code* denotes the +Topo+Test configuration running on Claude-4.5-Sonnet, as Claude Code does not support GPT-4o.

Results and Analysis. Figure 4(right) reports Pass@1 broken down by difficulty level. Among the four methods, the quality gradient observed in the main evaluation and cross-language experiments persists: RepoAgent (22.0% easy, 14.0% hard), Claude Code* (38.0%, 27.0%), and ACE (50.0%, 38.0%) yield progressively larger improvements, while SPECFORGE achieves the highest performance (74.5% easy, 51.7% hard), surpassing even the SWE-DEV reference PRD (55.7% easy, 44.6% hard) by +18.8 and +7.1 percentage points, respectively. Appendix I further analyzes why SPECFORGE outperforms the PRD.

5 Related Work

Recursive Self-Improvement. Self-improving LLM systems differ first in the substrate into which an improvement is written. *Weight-space* methods write into model parameters: the model is fine-tuned on data it generated itself [25, 26], on rewards derived from its own outputs [27], or on rewards from a grounded environment [28]. The gain is amortized across tasks, but it cannot be read, diffed, or reverted. *Context-space* methods keep the weights frozen and write into the harness instead, through verbal self-refinement [29, 30, 16], prompt search and textual gradients [15, 31, 32, 33, 12, 13], experience memory [14], skill and tool synthesis [34], or search over the whole scaffold [35]; at the frontier, the procedure that performs the update is itself rewritten [36, 37, 38]. The improvement signal ranges from the model’s own judgment [39, 40] to external verifiers such as tests [41, 42, 43], compilers [44, 45], and environments [34, 46]. SPECFORGE is a context-space method with an execution-based improvement signal and a fixed update procedure. The key difficulty in such a recursive self-improvement loop is that behavioral specifications are inherently coupled through the code generated from them, so that an improvement to one can cause a global regression and make the iterative process oscillate.

Coding Agents. Coding is among the most rapidly advancing capabilities of LLM agents, in large part because code is executable, verifiable, and searchable: a candidate can be run, checked against tests, and improved by search. Agents such as Cursor, Codex, and Claude Code [1, 2, 3] now resolve issues and implement features across entire repositories [47, 48, 4, 5, 9]. The same property is carrying agents beyond conventional software, wherever the artifact can be expressed as a program or other interpretable text: games [49], explanatory videos rendered from animation code [50, 51], 3D scenes written as Blender scripts [52], and symbolic music [53]. SPECFORGE is complementary to these agents: it supplies behavioral specifications that a coding agent can condition on, which in our experiments improve cross-language translation and new-feature implementation (§4.5).

Code Documentation. Documentation generation has evolved from template-based heuristics [54, 55, 56] through neural summarization [57, 58] to modern LLM-based repository-level approaches. RepoAgent [10] extracts structural patterns from repositories to generate API references with hierarchical organization, while DocAgent [11] employs retrieval-augmented generation to produce context-aware explanations grounded in relevant code snippets. Both optimize for subjective quality metrics—clarity, completeness, and helpfulness—evaluated through human assessment or proxy metrics like BLEU scores against reference documentation. These approaches uniformly assume documentation serves human developers and measure quality through readability heuristics rather than downstream task performance. See Appendix K for further related work.

6 Conclusion

We presented SPECFORGE, a multi-frontier tree search framework that makes specification-mediated self-improvement stable and compounding. A frozen coding agent improves by rewriting the behavioral specifications in its harness from the failures of the code it writes from them; unlike documentation written for human readers, a specification is judged only through the code it induces. Within this formulation, the central challenge is *output coupling*: because generated entities are behaviorally interdependent, locally beneficial refinements can trigger global regressions, rendering standard monolithic or sequential refinement unstable. SPECFORGE addresses this challenge through Pareto-guided frontier preservation, dependency-constrained entity selection, and error-diversified expansion, enabling stable progress under coupled search dynamics. Empirically, it consistently outperforms strong baselines on DEVEVAL+, exhibits superior search efficiency and robustness, and produces specifications that improve downstream performance in both cross-language translation and new-feature implementation, indicating that what the loop improves is reusable knowledge of the codebase rather than a fit to the tests that guided it.

References

- [1] Anysphere. Cursor: The AI Code Editor. <https://cursor.com/>, 2025.
- [2] OpenAI. Introducing Codex. <https://openai.com/index/introducing-codex/>, 2025.
- [3] Anthropic. Claude Code. <https://code.claude.com/docs/en/overview>, 2025.
- [4] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *EMNLP*, 2023.
- [5] Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, et al. Prompting large language models to tackle the full software development lifecycle: A case study. In *COLING*, 2025.
- [6] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.
- [7] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *NeurIPS*, 2020.
- [8] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *ICSE*, 2024.
- [9] Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv*, 2026.
- [10] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Feng, et al. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.
- [11] Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. DocAgent: A multi-agent system for automated code documentation generation. In *ACL*, pages 460–471, 2025.
- [12] Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *EMNLP*, 2024.
- [13] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnab Singhi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *ICLR*, 2026.
- [14] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026.
- [15] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024.
- [16] Xidong Feng, Ziyu Wan, Mengyue Yang, Ziyang Wang, Girish A. Koushik, Yali Du, Ying Wen, and Jun Wang. Natural language reinforcement learning, 2024.
- [17] Aurélien Garivier and Eric Moulines. On upper-confidence bound policies for switching bandit problems. In *COLT, ALT’11*, page 174–188, Berlin, Heidelberg, 2011. Springer-Verlag.
- [18] Kefan Li and Yuan Yuan. Large language models as test case generators: Performance evaluation and enhancement. *arxiv:2404.13340*, 2024.

- [19] Bo Jiang, Zhenyu Zhang, T. H. Tse, and T. Y. Chen. How well does test case prioritization integrate with statistical fault localization? *IEEE COMPSAC*, July 2009.
- [20] Kecheng Wang, Tiantian Wang, and Xiaohong Su. Test case selection using multi-criteria optimization for effective fault localization. *Computing*, 2018.
- [21] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In *ICML*, 2023.
- [22] Hongfu Gao, Feipeng Zhang, Wenyu Jiang, Jun Shu, Feng Zheng, and Hongxin Wei. On the noise robustness of in-context learning for text generation. In *NeurIPS*, 2024.
- [23] Yang Guo, Yutian Tao, Yifei Ming, Robert D. Nowak, and Yingyu Liang. Retrieval-augmented generation as noisy in-context learning: A unified theory and risk bounds. *arXiv*, 2025.
- [24] Yutong Cheng, Haifeng Chen, Wenchao Yu, Xujiang Zhao, Peng Gao, and Wei Cheng. Escaping whack-a-mole: Optimizing documentation as repo-specific playbooks for coding agents. In *ICML*, 2026.
- [25] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *ACL*, 2023.
- [26] Adam Zweiger, Jyothish Pari, Han Guo, Ekin Akyürek, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. In *NeurIPS*, 2025.
- [27] Yuxin Zuo, Kaiyan Zhang, Li Sheng, Shang Qu, Ganqu Cui, Xuekai Zhu, Haozhan Li, Yuchen Zhang, Xinwei Long, Ermo Hua, Biqing Qi, Youbang Sun, Zhiyuan Ma, Lifan Yuan, Ning Ding, and Bowen Zhou. TTRL: Test-time reinforcement learning. In *NeurIPS*, 2025.
- [28] Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Wenyi Zhao, Yu Yang, Xinyue Yang, Jiadai Sun, Shuntian Yao, Tianjie Zhang, Wei Xu, Jie Tang, and Yuxiao Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *ICLR*, 2025.
- [29] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.
- [30] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.
- [31] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic” differentiation” via text. *arXiv preprint arXiv:2406.07496*, 2024.
- [32] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. In *NeurIPS*, 2024.
- [33] Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware prompt optimization framework. In *ACL*, 2025.
- [34] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- [35] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *ICLR*, 2025.
- [36] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *ICML*, 2024.

- [37] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *COLM*, 2024.
- [38] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *ICLR*, 2026.
- [39] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- [40] Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. In *ICML*, 2024.
- [41] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *ICLR*, 2023.
- [42] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *ICLR*, 2024.
- [43] Yutong Cheng, Wei Cheng, Wenchao Yu, Dawei Zhou, and Haifeng Chen. Small but mighty: Test suites for self-evolving agents via quality-diversity evolutionary search. In *Workshop of NeurIPS (EvoRobust)*, 2026.
- [44] 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 *NeurIPS*, 2022.
- [45] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. In *ICLR*, 2023.
- [46] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *ICLR*, 2024.
- [47] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *ICLR*, 2024.
- [48] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024.
- [49] Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development. *arXiv preprint arXiv:2602.11103*, 2026.
- [50] Max Ku, Thomas Chong, Jonathan Leung, Krish Shah, Alvin Yu, and Wenhui Chen. Theorem-ExplainAgent: Towards video-based multimodal explanations for LLM theorem understanding. In *ACL*, 2025.
- [51] Yanzhe Chen, Kevin Qinghong Lin, and Mike Zheng Shou. Code2Video: A code-centric paradigm for educational video generation. *arXiv preprint arXiv:2510.01174*, 2025.
- [52] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A. Ross, Cordelia Schmid, and Alireza Fathi. SceneCraft: An LLM agent for synthesizing 3D scenes as Blender code. In *ICML*, 2024.
- [53] Qixin Deng, Qikai Yang, Ruibin Yuan, Yipeng Huang, Yi Wang, Xubo Liu, Zeyue Tian, Jiahao Pan, Ge Zhang, Hanfeng Lin, Yizhi Li, Yinghao Ma, Jie Fu, Chenghua Lin, Emmanouil Benetos, Wenwu Wang, Guangyu Xia, Wei Xue, and Yike Guo. ComposerX: Multi-agent symbolic music composition with LLMs. In *ISMIR*, 2024.
- [54] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, 2010.

- [55] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, 2010.
- [56] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC*, 2013.
- [57] Srinivasan Iyer, Ioannis Konstantas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *ACL*, August 2016.
- [58] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *ACL*, 2020.
- [59] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, May 2002.
- [60] Yutong Cheng, Osama Bajaber, Saimon Amanuel Tsegai, Dawn Song, and Peng Gao. CTINexus: Automatic cyber threat intelligence knowledge graph construction using large language models. In *EuroS&P*, 2025.
- [61] Yutong Cheng, Yang Liu, Changze Li, Dawn Song, and Peng Gao. CTIConnect: A benchmark for retrieval-augmented LLMs over heterogeneous cyber threat intelligence. In *KDD*, 2026.
- [62] Yutong Cheng, Changze Li, Qian Cui, Wei Ding, Lingzhi Wang, Yan Chen, and Peng Gao. CTIFoundry: An agent-native corpus scaffold for cyber threat intelligence. *arXiv preprint arXiv:2608.18613*, 2026.
- [63] Junjielong Xu, Qiulai Fu, Zhou Ruixing Zhu, Yutong Cheng, Zhijing Li, Yuchi Ma, and Pinjia He. Hue: A user-adaptive parser for hybrid logs. In *ESEC/FSE*, 2023.
- [64] Rizky Ramadhana Putra, Raihan Sultan Pasha Basuki, Yutong Cheng, and Peng Gao. NL2Logic: AST guided translation of natural language into first-order logic with large language models. In *EACL*, 2026.

Appendix

Appendix Contents

A	Output Coupling and Refinement Interference	17
B	Theoretical Analysis	17
B.1	Why Multi-Frontier Search Escapes Certain Frontier-Local Optima	17
B.2	Proof of Signal Corruption	18
B.3	Best-Branch Advantage of Diversified Expansion	19
C	Algorithm Details	20
C.1	Recursive and Cyclic Call Graphs	22
C.2	D-UCB Bandit Details	23
C.3	Expansion Implementation Details	23
D	Case Study: Module-Level Search Dynamics	24
D.1	Iterative Refinement: Diagnosis Without Action	25
D.2	ACE: Regression and Recovery Tax	25
D.3	SpecForge: Callee-Before-Caller Unlocking	25
D.4	Key Takeaways	26
E	Fidelity of the Self-Generated Improvement Signal	26
F	Prompt Templates	27
F.1	Stage 1: Diagnosis Prompt	28
F.2	Stage 2: Prescription Prompt	28
F.3	Code Generation Prompt	29
F.4	Test Case Generation Prompt	29
F.5	Iterative Refinement Prompt (Baseline)	30
F.6	Commercial Agent Prompts	30
G	Baseline Implementation Details	32
H	DevEval+ Benchmark	33
H.1	Overview	33
H.2	Module Composition	33
H.3	Data Collection and Curation	33
H.4	Dependency Structure	34
H.5	Test Suite Construction	34
H.6	Cross-Language Subset (DevEval+-Java)	34
I	New Feature Implementation Setup (SWE-DEV)	35

J	Test Case Generation	36
J.1	Overview	36
J.2	Coverage-Driven Test Generation	36
J.3	Quality Considerations	37
J.4	Hyperparameters	37
K	Additional Related Work	37
L	Compute Resources Used	38
M	Use of LLMs	38
N	Limitations	38
O	Broader Impact	38

A Output Coupling and Refinement Interference

This appendix provides the formal definitions and theorem statement for the output coupling problem discussed in §2.

Definition 2 (Output Coupling). *A generation task exhibits output coupling if the correctness of output c_i depends not only on its own specification d_i but also on the actual realizations of outputs it depends on:*

$$\phi_i(\mathcal{D}) = \phi_i(d_1, \dots, d_n, c_1, \dots, c_n) \quad \text{where } c_j = \mathcal{G}(\mathcal{D}, e_j). \quad (6)$$

Output coupling makes the following failure mode pervasive rather than exceptional:

Definition 3 (Refinement Interference). *Let $\mathcal{D}[d_i \mapsto d'_i]$ denote the specification bundle with d_i replaced by d'_i . A refinement $d_i \rightarrow d'_i$ exhibits interference if it degrades another entity:*

$$\exists j \neq i : \quad \phi_j(\mathcal{D}[d_i \mapsto d'_i]) < \phi_j(\mathcal{D}). \quad (7)$$

Self-refinement methods select entities to refine without reasoning over the dependency structure—typically at random or by prioritizing entities with the most errors—ignoring the call graph entirely.

B Theoretical Analysis

This section provides theoretical analysis and complete proofs for the theoretical results stated in the main paper.

B.1 Why Multi-Frontier Search Escapes Certain Frontier-Local Optima

A central role of the Pareto archive in SPECFORGE is to prevent the search from collapsing onto a single refinement trajectory too early. We formalize this benefit through the notion of *frontier-local optimality*: a state may be locally saturated with respect to its currently eligible refinements, while another non-dominated state still exposes a different eligibility frontier from which progress remains possible.

Definition 4 (Frontier-Local Optimum). *Let $v = (\mathcal{D}_v, \phi_v, \mathcal{S}_v)$ be a search state with eligibility frontier $\mathcal{E}_{\text{elig}}(v)$. We say that v is a frontier-local optimum if for every eligible entity $e \in \mathcal{E}_{\text{elig}}(v)$ and every admissible single-entity refinement $v \xrightarrow{e} v'$,*

$$|\mathcal{S}_{v'}| \leq |\mathcal{S}_v|. \quad (8)$$

That is, no refinement available from the current frontier increases the solved-set objective.

This notion captures the practical failure mode of single-path refinement: once the current state becomes frontier-locally optimal, all subsequent search effort is confined to a neighborhood with no objective improvement. The key observation is that frontier-local optimality is *state-dependent*: because eligibility depends on the solved set $\mathcal{S}_v$, two states with similar objective values may expose different feasible next refinements.

Definition 5 (Archive Dominance). *For two states u and v , we say that u dominates v if*

$$\phi_u \succeq \phi_v \iff \phi_i(\mathcal{D}_u) \geq \phi_i(\mathcal{D}_v) \text{ for all } i \in [n]. \quad (9)$$

A Pareto archive $\mathcal{A}$ contains only non-dominated states.

The importance of this criterion is that it preserves states with complementary strengths across entities rather than only the current best state under a scalar score. Such states can induce different solved sets and therefore different eligibility frontiers.

Assumption 1 (Complementary Frontier Reachability). *Suppose there exist two non-dominated states u and v in the search tree such that:*

1. *u is a frontier-local optimum;*
2. *there exists an entity $e^* \in \mathcal{E}_{\text{elig}}(v) \setminus \mathcal{E}_{\text{elig}}(u)$ and a refinement $v \xrightarrow{e^*} v^*$ such that $|\mathcal{S}_{v^*}| > |\mathcal{S}_v|$.*

This assumption states that one archived frontier may be saturated while another still contains a productive move. It is natural in our setting because eligibility is induced by the solved subset, and different trajectories typically solve different subsets of entities.

Proposition 1 (Multi-Frontier Advantage). *Under Assumption 1, any search strategy restricted to expanding only the single incumbent state u cannot improve the objective $|\mathcal{S}|$, whereas a multi-frontier strategy that retains both u and v in a Pareto archive has nonzero probability of reaching a state with strictly larger solved set.*

Proof. Since u is frontier-locally optimal, by Definition 4, every admissible refinement from u satisfies $|\mathcal{S}_{u'}| \leq |\mathcal{S}_u|$. Hence any single-path strategy confined to descendants of u cannot obtain an immediate objective improvement.

By Assumption 1, there exists another archived state v and an eligible entity $e^* \in \mathcal{E}_{\text{elig}}(v)$ such that refining e^* yields a child v^* with $|\mathcal{S}_{v^*}| > |\mathcal{S}_v|$. Because u and v are non-dominated, a Pareto archive retains both states. Therefore, any archive-sampling policy that assigns nonzero probability to every archived state has nonzero probability of selecting v and then refining e^* , reaching an improved state. Thus, retaining multiple non-dominated frontiers strictly enlarges the set of reachable improving moves compared with single-path search.

Proposition 1 formalizes the role of the Pareto archive in escaping local optima. Its benefit is not merely diversity for its own sake; rather, archive diversity induces *frontier diversity*. Because eligibility is determined by the solved subset, preserving multiple non-dominated states preserves multiple feasible continuation sets. When one frontier is exhausted, another may still expose an improving refinement.

The argument also clarifies why scalar best-first retention is brittle in our setting. A state with the highest current $|\mathcal{S}|$ or $\bar{\phi}$ need not be the most promising continuation state if its eligibility frontier is already saturated. By preserving non-dominated states with complementary per-entity strengths, the Pareto archive acts as a structural hedge against premature convergence.

Implication for SPECFORGE. In SPECFORGE, multi-frontier search is realized by maintaining a Pareto archive over per-entity pass-rate vectors and sampling from it during expansion. This design ensures that the search does not over-commit to a single locally attractive trajectory, but can continue from alternative archived frontiers whose dependency structure admits further progress.

B.2 Proof of Signal Corruption

We formalize the claim in §3.3 that noisy failures can reduce refinement quality when mixed with informative ones.

Let $\mathcal{F}_e(v) = \mathcal{F}_e^+ \cup \mathcal{F}_e^-$ denote the failing tests for entity e at state v , where $\mathcal{F}_e^+$ contains informative failures and $\mathcal{F}_e^-$ noisy ones. For any error batch $\mathbf{e} \subseteq \mathcal{F}_e(v)$, let

$$q(\mathbf{e}) = P(\text{successful refinement} \mid \mathbf{e}) \quad (10)$$

denote the success probability when conditioning on $\mathbf{e}$.

Assumption 2 (Signal–Noise Decomposition). *There exist functions $I, N : 2^{\mathcal{F}_e(v)} \rightarrow \mathbb{R}_{\geq 0}$ and a function $h : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \rightarrow [0, 1]$ such that for any $\mathbf{e} \subseteq \mathcal{F}_e(v)$,*

$$q(\mathbf{e}) = h(I(\mathbf{e}), N(\mathbf{e})), \quad (11)$$

where h is increasing in its first argument and decreasing in its second. Moreover,

$$I(\mathbf{e}) = I(\mathbf{e} \cap \mathcal{F}_e^+), \quad N(\mathbf{e}) = N(\mathbf{e} \cap \mathcal{F}_e^-). \quad (12)$$

Assumption 3 (Positive Signal and Strict Corruption). *For any nonempty $\mathbf{e}^+ \subseteq \mathcal{F}_e^+$ and nonempty $\mathbf{e}^- \subseteq \mathcal{F}_e^-$,*

$$I(\mathbf{e}^+) > 0, \quad N(\mathbf{e}^-) > 0. \quad (13)$$

Moreover,

$$h(I(\mathbf{e}^+), N(\mathbf{e}^-)) < h(I(\mathbf{e}^+), 0). \quad (14)$$

Theorem 1 (Signal Corruption). *Under Assumptions 2–3, for any nonempty $\mathbf{e}^+ \subseteq \mathcal{F}_e^+$ and $\mathbf{e}^- \subseteq \mathcal{F}_e^-$,*

$$q(\mathbf{e}^-) < q(\mathbf{e}^+), \quad (15)$$

and

$$q(\mathbf{e}^+ \cup \mathbf{e}^-) < q(\mathbf{e}^+). \quad (16)$$

In particular, if $\mathcal{F}_e^- \neq \emptyset$, then conditioning on the full failure set $\mathcal{F}_e(v)$ is suboptimal relative to conditioning on its informative part alone.

Proof. For any nonempty $\mathbf{e}^+ \subseteq \mathcal{F}_e^+$,

$$q(\mathbf{e}^+) = h(I(\mathbf{e}^+), 0), \quad (17)$$

since $\mathbf{e}^+ \cap \mathcal{F}_e^- = \emptyset$. For any nonempty $\mathbf{e}^- \subseteq \mathcal{F}_e^-$,

$$q(\mathbf{e}^-) = h(0, N(\mathbf{e}^-)), \quad (18)$$

since $\mathbf{e}^- \cap \mathcal{F}_e^+ = \emptyset$. Because h increases with signal and decreases with noise, and because $I(\mathbf{e}^+) > 0$ and $N(\mathbf{e}^-) > 0$ by Assumption 3, we obtain

$$h(0, N(\mathbf{e}^-)) < h(I(\mathbf{e}^+), 0), \quad (19)$$

which proves Eq. (15).

Next,

$$q(\mathbf{e}^+ \cup \mathbf{e}^-) = h(I(\mathbf{e}^+), N(\mathbf{e}^-)), \quad (20)$$

and Assumption 3 gives

$$h(I(\mathbf{e}^+), N(\mathbf{e}^-)) < h(I(\mathbf{e}^+), 0) = q(\mathbf{e}^+), \quad (21)$$

proving Eq. (16).

Finally, if $\mathcal{F}_e^- \neq \emptyset$, then taking $\mathbf{e}^+ = \mathcal{F}_e^+$ and $\mathbf{e}^- = \mathcal{F}_e^-$ yields

$$q(\mathcal{F}_e(v)) = q(\mathcal{F}_e^+ \cup \mathcal{F}_e^-) < q(\mathcal{F}_e^+). \quad (22)$$

B.3 Best-Branch Advantage of Diversified Expansion

We now formalize the advantage of diversified expansion in §3.3.

Definition 6 (Repeated Full-Conditioned Baseline). *Given width W , the repeated full-conditioned baseline constructs W branches, each conditioned on the same full failure set $\mathcal{F}_e(v)$. Its best one-step branch quality is*

$$Q_{\text{full}}^*(W) := \max_{1 \leq j \leq W} q(\mathcal{F}_e(v)) = q(\mathcal{F}_e(v)). \quad (23)$$

Definition 7 (Diversified Expansion). *Given width W , diversified expansion constructs W branches conditioned on batches*

$$\mathbf{e}_1, \dots, \mathbf{e}_W \subseteq \mathcal{F}_e(v).$$

Its best one-step branch quality is

$$Q_{\text{div}}^*(W) := \max_{1 \leq j \leq W} q(\mathbf{e}_j). \quad (24)$$

Theorem 2 (Best-Branch Advantage of Diversified Expansion). *If there exists $j^* \in \{1, \dots, W\}$ such that*

$$q(\mathbf{e}_{j^*}) > q(\mathcal{F}_e(v)), \quad (25)$$

then diversified expansion strictly improves the best one-step branch quality over the repeated full-conditioned baseline:

$$Q_{\text{div}}^*(W) > Q_{\text{full}}^*(W). \quad (26)$$

Equivalently,

$$\max_{1 \leq j \leq W} q(\mathbf{e}_j) > q(\mathcal{F}_e(v)). \quad (27)$$

Proof. By Definition 7,

$$Q_{\text{div}}^*(W) = \max_{1 \leq j \leq W} q(\mathbf{e}_j) \geq q(\mathbf{e}_{j^*}). \quad (28)$$

Combining this with Eq. (25) yields

$$Q_{\text{div}}^*(W) > q(\mathcal{F}_e(v)). \quad (29)$$

By Definition 6,

$$Q_{\text{full}}^*(W) = q(\mathcal{F}_e(v)). \quad (30)$$

Therefore,

$$Q_{\text{div}}^*(W) > Q_{\text{full}}^*(W). \quad (31)$$

Theorem 1 shows why full-set conditioning can be suboptimal: informative and noisy failures may be mixed in $\mathcal{F}_e(v)$, and the noisy part can depress refinement quality. Theorem 2 then shows that if diversified sampling produces even one cleaner batch than $\mathcal{F}_e(v)$, its best branch is strictly better than repeating full conditioning across all W branches.

Empirical Motivation. Our theoretical treatment is motivated by two recurring empirical observations from the software engineering literature.

1. **Failing tests differ substantially in diagnostic value.** Not all failures are equally informative for identifying the underlying defect. Some tests reveal the root cause directly, whereas others fail only through cascading effects or repeatedly exercise the same execution path. In the context of LLM-generated tests, Yuan et al. [18] observe that generated inputs often cluster around similar patterns, leaving critical edge cases underexplored and further increasing heterogeneity in diagnostic usefulness.
2. **Using more tests can degrade diagnosis rather than improve it.** Prior work on fault localization shows that adding tests beyond a certain point may reduce localization quality when additional tests contribute little new information. Chen et al. [19] show that excessive or low-value tests can dilute diagnostic signals, while Su et al. [20] demonstrate that carefully selected subsets can outperform the full test suite for fault localization.

In our setting, these observations imply that the full failure set may contain substantial noise. We attribute noisy errors $\mathcal{F}^-$ to three common sources: (i) **cascading failures**, where a test fails because a callee is incorrectly implemented rather than because the target entity’s specification is inadequate; (ii) **redundant signals**, where multiple tests expose essentially the same failure mode and therefore add little incremental information; and (iii) **misleading co-occurrence**, where presenting several errors together encourages the optimizer LLM to infer spurious patterns that are not causally related to the true knowledge gap.

Remark 1 (Connection to Fault Localization). *Our result formalizes a central intuition from fault localization: for diagnosis, test quality often matters more than test quantity [20]. Diversified error-batch sampling can be viewed as a hedging strategy: by exploring multiple subsets of the failure set, it increases the chance of encountering a batch dominated by informative signals, without requiring prior knowledge of the latent partition between informative and noisy errors.*

C Algorithm Details

Algorithm 1 presents the complete SPECFORGE procedure with all subroutines.

Algorithm 1 SPECFORGE: Complete Procedure

Require: Initial specification bundle $\mathcal{D}^{(0)}$, call graph G , test suite $\mathcal{T}$, width W , evaluation budget B

Ensure: Best specification bundle $\mathcal{D}_{v_{\text{best}}}$

```
1: // Initialization
2:  $\mathcal{C}^{(0)} \leftarrow \text{GENERATE}(\mathcal{D}^{(0)})$  ▷ Generate initial code
3:  $\phi^{(0)} \leftarrow \text{EVALUATE}(\mathcal{C}^{(0)}, \mathcal{T})$  ▷ Run all tests
4:  $\mathcal{S}^{(0)} \leftarrow \{e_i : \phi_i^{(0)} = 1\}$ 
5:  $v_0 \leftarrow \text{CREATENODE}(\mathcal{D}^{(0)}, \mathcal{C}^{(0)}, \phi^{(0)}, \mathcal{S}^{(0)})$ 
6:  $\mathcal{A} \leftarrow \{v_0\}; v_{\text{best}} \leftarrow v_0$ 
7:  $\text{BANDIT} \leftarrow \text{INITDUCB}(\gamma=0.95, C=2\sqrt{2})$ 
8:  $b \leftarrow 0$  ▷ Evaluations used: one per generated child
9:
10: // Main search loop
11: while  $b < B$  do
    // Pareto-guided state selection (§3.1)
12:    $v \leftarrow \text{PARETOSAMPLE}(\mathcal{A})$  ▷ Algorithm 2
    // Dependency-constrained entity selection (§3.2)
13:    $\mathcal{E}_{\text{elig}} \leftarrow \text{ELIGIBILITYFRONTIER}(v, G)$  ▷ Algorithm 3
14:    $e \leftarrow \text{BANDIT.SELECT}(\mathcal{E}_{\text{elig}}, \phi_v)$  ▷ Eq. (4)
15:   if  $e = \text{NULL}$  then continue
16:   end if
    // Error-diversified expansion (§3.3)
17:    $W_t \leftarrow \min(W, B - b)$  ▷ Siblings share the same budget
18:    $\{e_1, \dots, e_{W_t}\} \leftarrow \text{DIVERSIFIEDERRORBATCHES}(e, v, W_t)$ 
19:    $\{v'_1, \dots, v'_{W_t}\} \leftarrow \text{EXPANDPARALLEL}(v, e, \{e_1, \dots, e_{W_t}\})$  ▷ Algorithm 4
20:    $b \leftarrow b + W_t$  ▷ Each child is one specification → code → test evaluation
    // Archive update
21:   for all  $v' \in \{v'_1, \dots, v'_{W_t}\}$  do
22:     if  $\neg \exists u \in \mathcal{A} : \phi_u \succeq \phi_{v'}$  then ▷ Pareto dominance check
23:        $\mathcal{A} \leftarrow (\mathcal{A} \setminus \{u : \phi_{v'} \succeq \phi_u\}) \cup \{v'\}$ 
24:     end if
25:     if  $|\mathcal{S}_{v'}| > |\mathcal{S}_{v_{\text{best}}}|$  then  $v_{\text{best}} \leftarrow v'$ 
26:     end if
27:   end for
28:    $\text{BANDIT.UPDATE}(e, \Delta\phi_e = \max_j \phi_e(\mathcal{D}_{v'_j}) - \phi_e(\mathcal{D}_v))$ 
29: end while
30:
31: return  $\mathcal{D}_{v_{\text{best}}}$ 
```

Algorithm 2 PARETOSAMPLE: Frequency-Weighted Pareto Front Sampling

Require: Pareto archive $\mathcal{A}$, entity set $\mathcal{E} = \{e_1, \dots, e_n\}$

Ensure: Selected node $v \in \mathcal{A}$

```
1:  $w(v) \leftarrow 0$  for all  $v \in \mathcal{A}$  ▷ Initialize sampling weights
2:
3: // For each entity, find the archive node(s) with highest pass rate
4: for  $i = 1$  to  $n$  do
5:    $\phi_i^{\text{max}} \leftarrow \max_{v \in \mathcal{A}} \phi_i(\mathcal{D}_v)$ 
6:    $\mathcal{A}_i^* \leftarrow \{v \in \mathcal{A} : \phi_i(\mathcal{D}_v) = \phi_i^{\text{max}}\}$ 
7:   for all  $v \in \mathcal{A}_i^*$  do
8:      $w(v) \leftarrow w(v) + 1/|\mathcal{A}_i^*|$  ▷ Split credit among ties
9:   end for
10: end for
11:
12: // Sample proportional to weight
13: return  $v \sim \text{CATEGORICAL}(\{w(v)/\sum_u w(u)\}_{v \in \mathcal{A}})$ 
```

Algorithm 3 ELIGIBILITYFRONTIER: Dependency-Constrained Entity Selection

Require: Node v , call graph $G = (\mathcal{E}, E)$ **Ensure:** Eligible entity set $\mathcal{E}_{\text{elig}}$

```
1:  $\mathcal{E}_{\text{elig}} \leftarrow \emptyset$ 
2:
3: // Collect unsolved entities whose callees are all resolved
4: for  $e \in \mathcal{E}$  do
5:   if  $e \notin \mathcal{S}_v$  then
6:      $\text{deps\_resolved} \leftarrow \text{TRUE}$ 
7:     for  $(e, e_j) \in E$  do ▷ Iterate over callees of  $e$ 
8:       if  $e_j \notin \mathcal{S}_v$  then
9:          $\text{deps\_resolved} \leftarrow \text{FALSE}$ ; break
10:      end if
11:    end for
12:    if  $\text{deps\_resolved}$  then
13:       $\mathcal{E}_{\text{elig}} \leftarrow \mathcal{E}_{\text{elig}} \cup \{e\}$ 
14:    end if
15:  end if
16: end for
17:
18: return  $\mathcal{E}_{\text{elig}}$  ▷ Eq. (3)
```

Algorithm 4 EXPANDPARALLEL: Error-Diversified Parallel Expansion

Require: Parent node v , target entity e , error batches $\{e_1, \dots, e_W\}$ **Ensure:** Child nodes $\{v'_1, \dots, v'_W\}$

```
1:
2: for  $k = 1$  to  $W$  in parallel do
3:   // Error-driven refinement (two-stage diagnosis-prescription)
4:    $f_k \leftarrow \text{DIAGNOSE}(d_e^{(v)}, e_k)$  ▷ Stage 1: identify knowledge gap
5:    $d_e^{(k)} \leftarrow \text{PRESCRIBE}(d_e^{(v)}, f_k)$  ▷ Stage 2: targeted specification update
6:    $\mathcal{D}^{(k)} \leftarrow \mathcal{D}_v[e \mapsto d_e^{(k)}]$  ▷ Substitute refined specification
7:
8:   // Regenerate and evaluate
9:    $\mathcal{C}^{(k)} \leftarrow \mathcal{C}(\mathcal{D}^{(k)}) = (\mathcal{G}(\mathcal{D}^{(k)}, e_1), \dots, \mathcal{G}(\mathcal{D}^{(k)}, e_n))$  ▷ Regenerate the full module
10:   $\phi^{(k)} \leftarrow \text{EVALUATE}(\mathcal{C}^{(k)}, \mathcal{T})$  ▷ Re-run full test suite
11:   $\mathcal{S}^{(k)} \leftarrow \{e_i : \phi_i^{(k)} = 1\}$ 
12:
13:  // Create child node
14:   $v'_k \leftarrow \text{CREATENODE}(\mathcal{D}^{(k)}, \mathcal{C}^{(k)}, \phi^{(k)}, \mathcal{S}^{(k)})$ 
15:   $\text{SETPARENT}(v'_k, v)$ 
16: end for
17:
18: return  $\{v'_1, \dots, v'_W\}$ 
```

C.1 Recursive and Cyclic Call Graphs

The eligibility frontier (Eq. (3)) requires every callee of an entity to be resolved before the entity can be refined. On a call graph with cycles this condition can never hold for the entities on a cycle, so SPECFORGE removes cycles before search.

Self-recursion. A self-call $(e, e) \in E$ is ignored when evaluating eligibility: an entity cannot wait for itself to be resolved, and its recursive behavior is described within its own specification d_e .

Mutual recursion. We compute the strongly connected components (SCCs) of G and contract each SCC with more than one member into a single *meta-entity*. The search then operates on *refinement units*, each of which is either a single entity or a meta-entity. The condensation of G over these units is a DAG by construction, so the callee-before-caller order and the eligibility gate apply unchanged: a unit is eligible once it is unsolved and all units it calls are resolved, and a meta-entity counts as resolved once all of its members are solved.

Joint refinement of a meta-entity. When a meta-entity is selected, its members are refined together in a single expansion. The refinement prompt presents the source code and current specifications of all members, states that they are mutually recursive, and requires a complete replacement specification for every member, so that the specifications of the cycle remain consistent with one another. The resulting specifications are substituted into $\mathcal{D}_v$ together, and the child is evaluated as for any other expansion.

C.2 D-UCB Bandit Details

The entity selection bandit maintains a single set of statistics shared across all state nodes. We use Discounted UCB [17] with the following configuration:

Reward Normalization. After each expansion of entity e with observed improvement $\Delta\phi_e$, the normalized reward is computed as:

$$\hat{r}_e = \frac{\Delta\phi_e}{1 - \phi_e}, \quad (32)$$

where ϕ_e is the pass rate before refinement. This isolates the gap-closing rate θ_e from the remaining gap $(1 - \phi_e)$, preventing entities with large gaps from receiving inflated rewards.

Discounted Statistics. At each update step, all arms' statistics are discounted before incorporating the new observation:

$$n_\gamma(e) \leftarrow \gamma \cdot n_\gamma(e), \quad \bar{r}_\gamma(e) \leftarrow \gamma \cdot \bar{r}_\gamma(e), \quad N_\gamma \leftarrow \gamma \cdot N_\gamma \quad \forall e, \quad (33)$$

followed by $n_\gamma(e) \leftarrow n_\gamma(e) + 1$, $\bar{r}_\gamma(e) \leftarrow \bar{r}_\gamma(e) + \hat{r}_e$, and $N_\gamma \leftarrow N_\gamma + 1$ for the pulled arm. The discounted mean is $\hat{\theta}_e = \bar{r}_\gamma(e)/n_\gamma(e)$.

Hyperparameters.

- **Discount factor** $\gamma = 0.95$: yields an effective observation window of $\sim 1/(1-\gamma) = 20$ steps, allowing the bandit to adapt as callees stabilize and entity difficulty shifts.
- **Exploration constant** $C = 2\sqrt{2}$: following the standard UCB1 setting [59].
- **Initialization**: all arms start with $n_\gamma(e) = 0$, $\bar{r}_\gamma(e) = 0$. Untried entities receive score $\infty \times (1 - \phi_e)$, ensuring they are explored first with gap-proportional priority.

C.3 Expansion Implementation Details

Error Batch Construction. We use beam width $W = 3$ throughout all experiments, balancing the hedging benefit of diversified expansion (Theorem 2) against per-iteration cost. Given failing tests $\mathcal{F}_e(v)$ for entity e at state v , we sample W batches independently: each batch $\mathbf{e}_k$ is drawn by selecting $\lceil |\mathcal{F}_e(v)|/2 \rceil$ failures uniformly at random without replacement from $\mathcal{F}_e(v)$, with batches generated sequentially and any duplicate redrawn until W distinct batches are obtained. This ensures each batch covers roughly half the failure set while varying in composition across the W branches.

Small Failure Set Handling. When $|\mathcal{F}_e(v)|$ is too small for W distinct same-sized subsets to exist, we relax the batch-size constraint and draw W distinct nonempty subsets of $\mathcal{F}_e(v)$ covering a range of sizes—e.g., $|\mathcal{F}_e(v)| = 2$ admits the three subsets $\{f_1\}$, $\{f_2\}$, $\{f_1, f_2\}$. Theorem 2 applies in all such cases since it requires only $\mathbf{e}_j \subseteq \mathcal{F}_e(v)$.

Diagnosis-Prescription Process. Each refinement follows a two-stage process. In the *diagnosis* stage, the LLM analyzes the error batch to identify the knowledge gap: what information is missing or ambiguous in the current specification that caused the incorrect generation. In the *prescription* stage, the LLM produces a targeted specification update addressing the diagnosed gap. This decomposition prevents the LLM from making unfocused edits and ensures each refinement is traceable to a specific failure signal. Prompt templates for both stages are provided in Appendix F.

D Case Study: Module-Level Search Dynamics

The convergence analysis in §4.3 characterizes search behavior at the *aggregate* level—module-level $\bar{\phi}$ across budget steps. Here we zoom in to *entity-level* dynamics on a single module, stocktrends/indicators (6 entities, 17 test cases), to expose the concrete mechanisms behind each method’s success or failure. Figure 5 presents a side-by-side trace of Iterative Refinement, ACE, and SPECFORGE at budget checkpoints $B \in \{10, 20, 30, 40, 50\}$, showing the verbatim specification state (labeled Doc Before in the figure), test errors (if any), the LLM’s self-generated reflection, and the resulting refinement (Doc After), alongside per-entity pass rate radar charts.

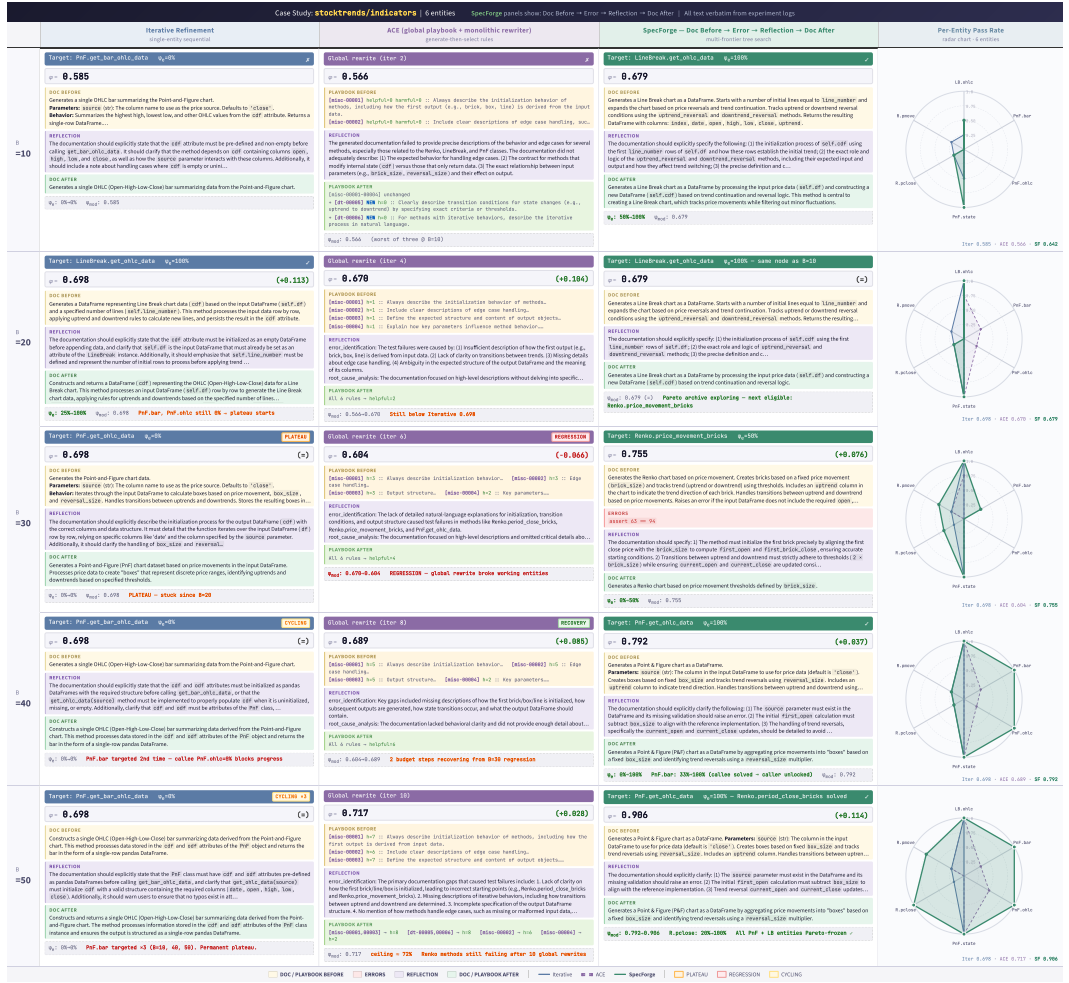


Figure 5: Module-level search dynamics on stocktrends/indicators (6 entities). Each row corresponds to a budget checkpoint; columns show the full refinement trace for Iterative Refinement (left), ACE (center-left), SPECFORGE (center-right), and per-entity pass rate radar charts (right). All text is verbatim from experiment logs, in which the specification is labeled “Doc”. Color coding: Doc Before (yellow), Errors (red), Reflection (purple), Doc After (green). Status tags indicate PLATEAU, REGRESSION, CYCLING, and RECOVERY. Radar charts overlay all three methods per checkpoint: Iterative (blue, solid), ACE (purple, dashed), SPECFORGE (green, bold).

D.1 Iterative Refinement: Diagnosis Without Action

The Iterative column reveals a striking pattern: the LLM *correctly diagnoses* the root cause of failure but *lacks the mechanism to act on its diagnosis*.

At $B=10$, the search targets `PnF.get_bar_ohlc_data` ($\phi_e=0\%$), a caller that depends on `PnF.get_ohlc_data` (the callee) to populate the `cdf` attribute. The reflection correctly identifies that “cdf must be pre-defined and non-empty” and that the method “depends on cdf containing columns open, high, low, and close.” Yet the refinement can only modify the *caller’s* specification—it cannot redirect budget to fix the callee whose broken implementation produces the malformed cdf in the first place. The entity remains at $\phi_e=0\%$.

By $B=20$, the search has shifted to `LineBreak.get_ohlc_data` and achieves a genuine success ($\phi_e: 25\% \rightarrow 100\%$), raising the module-level score to $\phi=0.698$. However, from this point onward, the search enters a **permanent plateau**: `PnF.get_ohlc_data` (the callee) never gets solved, so every subsequent attempt at the caller is doomed. The reflections at $B=40$ and $B=50$ both explicitly state that “`get_ohlc_data(source)` must be implemented to properly populate cdf”—diagnosing the callee as the bottleneck for the third time—but the single-entity sequential strategy has no way to redirect to the blocking dependency.

This produces the **cycling** pattern visible in the title bars: the same entity `PnF.get_bar_ohlc_data` is targeted at $B=10$, $B=40$, and $B=50$, with identical outcomes ($\phi_e=0\%$) each time. The radar charts confirm the stagnation: the Iterative polygon (blue) is identical from $B=20$ through $B=50$, with persistent zero-valued vertices at `PnF.bar`, `PnF.ohlc`, and `R.pmove`.

D.2 ACE: Regression and Recovery Tax

ACE’s global playbook mechanism exhibits a different but equally costly failure mode. Its rules accumulate helpful counts across iterations (e.g., `[misc-00001] helpful=0` at $B=10$ rising to `helpful=7` by $B=50$), and each iteration rewrites *all* entity specifications simultaneously from the updated playbook. This monolithic approach has no mechanism to protect previously solved entities.

The critical event occurs at $B=30$: a global rewrite causes **regression** from $\phi=0.670$ to $\phi=0.604$, a drop of -0.066 . The reflection diagnoses the failure accurately—“the documentation focused on high-level descriptions and omitted critical details about method behavior”—but this diagnosis applies equally to the entities that were *already working*. Because ACE rewrites all specifications from the playbook rather than preserving solved entities, the rewrite breaks previously correct descriptions of `PnF.get_state` and `LineBreak.get_ohlc_data`, which regress from $\phi_e=100\%$ to $\phi_e=50\%$ and $\phi_e=75\%$ respectively.

The next two budget steps ($B=40$, $B=50$) are spent **recovering** from this regression rather than making new progress. By $B=40$, ACE recovers to $\phi=0.689$, only marginally above its pre-regression score of 0.670 at $B=20$, and by $B=50$ it reaches $\phi=0.717$, a net gain of only $+0.047$ over three budget steps. Meanwhile, both Renko entities remain at or near zero throughout all 10 iterations, because the global rules are too coarse to capture the entity-specific algorithmic details (e.g., brick initialization alignment, reversal threshold logic) that these methods require. The `[misc-00004]` rule (“explain how key parameters influence method behavior”) stays frozen at `helpful=2` from $B=20$ onward, confirming that parameter-specific behavioral descriptions are the hardest to capture at the playbook level.

D.3 SpecForge: Callee-Before-Caller Unlocking

SPECFORGE’s trajectory illustrates how dependency-guided search converts the same budget into monotonic progress.

At $B=10$, the D-UCB bandit selects `LineBreak.get_ohlc_data`—a leaf entity with no unsolved callees—and solves it in a single step ($\phi_e: 50\% \rightarrow 100\%$). At $B=20$, the Pareto archive retains this state but finds no eligible entity that improves the frontier, so the module score holds at $\phi=0.679$. Crucially, *no budget is wasted*: the archive simply records that the current state is locally optimal and explores alternative branches.

The critical breakthrough comes at $B=30$ – 40 . At $B=30$, the search targets `Renko.price_movement_bricks`, whose test failure (`assert 63 == 94`) reveals a brick-

count mismatch. The reflection identifies the precise algorithmic gap: “the first brick must align the close price with `brick.size` to compute `first.open`.” This targeted, entity-specific diagnosis—impossible under ACE’s global playbook—raises ϕ_e from 0% to 50%.

At $B=40$, the dependency constraint makes `PnF.get_ohlc_data` eligible (its callee dependencies are resolved). The reflection pinpoints the initialization formula: “`first.open = roundit(first.value, box.size) - box.size`.” Solving this callee (ϕ_e : 0%→100%) triggers a **cascade**: its caller `PnF.get_bar_ohlc_data` simultaneously jumps from 33% to 100% *without any modification to its own specification*. This is the “callee solved → caller unlocked” effect that dependency-constrained search is designed to produce, and that Iterative refinement structurally cannot achieve.

By $B=50$, the Pareto archive allocates remaining budget to `Renko.period_close_bricks` (ϕ_e : 20%→100%), and the module reaches $\phi=0.906$. The radar chart at $B=50$ shows the SPECFORGE polygon (green) nearly filling the entire chart, while Iterative (blue) remains frozen at its $B=20$ shape.

D.4 Key Takeaways

This module-level trace exposes three failure modes and their structural causes:

1. **Dependency blindness** (Iterative). The LLM’s reflection is a powerful diagnostic tool—it correctly identifies cross-entity dependencies as the root cause of failure. But without a mechanism to act on dependency structure, this diagnostic power is wasted. The cycling pattern (same entity targeted 3 times, same diagnosis, same $\phi_e=0\%$) is a direct consequence of treating entities as independent.
2. **Collateral damage** (ACE). Global rewriting treats the specification bundle as a monolithic artifact, making it impossible to improve one entity without risking others. The regression at $B=30$ and subsequent two-step recovery illustrate the “two steps forward, one step back” dynamic that caps ACE’s effective progress rate. Even at convergence ($B=50$, $\phi=0.717$), entity-specific algorithmic details remain beyond the reach of global rules.
3. **Structural unlocking** (SPECFORGE). By constraining refinement to entities whose callees are already solved, SPECFORGE converts callee improvements into free caller improvements. The $B=40$ cascade—where solving `PnF.get_ohlc_data` simultaneously unlocks `PnF.get_bar_ohlc_data`—demonstrates that dependency-aware search does not merely avoid interference but actively *exploits* coupling as a source of compounding returns.

E Fidelity of the Self-Generated Improvement Signal

A recursive self-improvement loop is only as sound as the signal that drives it. In SPECFORGE the loop is fully self-supervised: the agent synthesizes the tests that score its own specifications, and no external label enters the loop. This raises a question that the main results cannot settle on their own, because every method in Table 1 is *evaluated* on hidden ground-truth tests but *optimized* against self-generated ones: does the self-generated signal track true behavioral correctness, or does the search merely learn to satisfy its own tests? In the latter case the measured gains would constitute reward hacking rather than self-improvement. Two properties are therefore required of the signal. *Fidelity*: optimizing against the self-generated signal must yield essentially the same outcome as optimizing against the ground truth. *Grounding*: the signal must be derived from execution rather than from the model’s own judgment, since Theorem 1 shows that a corrupted signal degrades refinement quality monotonically. This section verifies both properties and characterizes how performance degrades as the signal is weakened.

Construction of the signal. Test oracles are obtained by executing LLM-proposed inputs on the reference implementation to derive the expected outputs (§J); output correctness is therefore guaranteed by construction. The only residual imperfection lies in input quality: some generated inputs may be redundant, exercise overlapping code paths, or fail to cover critical edge cases. Error-diversified expansion (§3.3) is designed for precisely this kind of imperfection: by conditioning each child on a different subset of failing tests rather than on the full test set, it bounds the influence of any individual weak or redundant test.

Table 2: Sensitivity to feedback signal (GPT-4o, DevEval+).

Configuration	Feedback Signal	$\bar{\phi}$ (%)
SpecForge (default)	LLM tests	93.2
Ground-truth tests	GT tests	94.8
LLM-as-a-judge	Judge score	73.6
$\tau_{\text{cov}} = 0.7$	Reduced tests	93.4
$\tau_{\text{cov}} = 0.5$	Reduced tests	92.6
$\tau_{\text{cov}} = 0.3$	Reduced tests	87.5

Experimental design. We hold the search procedure, budget, and backbone (GPT-4o) fixed and vary only the improvement signal (Table 2). (i) *Signal source*: the LLM-generated tests are replaced by the hidden ground-truth tests, an oracle that is unavailable in practice and serves as an upper bound on signal quality. (ii) *Signal grounding*: execution feedback is replaced by an LLM-as-a-judge score of the generated code against the specification, i.e., the model evaluates itself without executing anything. (iii) *Signal coverage*: the coverage threshold τ_{cov} used during test generation is lowered from its default of 0.9 (§J) to 0.7, 0.5, and 0.3, so that the search is driven by progressively fewer tests.

Fidelity: self-generated vs. ground-truth tests. Replacing LLM-generated tests with ground-truth tests improves $\bar{\phi}$ only marginally, from 93.2 to 94.8, corresponding to a gain of just 1.6 percentage points. The self-generated signal is thus near-equivalent to the ground truth as an optimization target: the loop does not exploit its own tests, and the improvements reported in Table 1 reflect genuinely better behavioral specifications rather than reward hacking.

Grounding: execution feedback vs. LLM judgment. Replacing execution-based feedback with LLM-as-a-judge scoring causes a large performance drop from 93.2 to 73.6, i.e., a decrease of 19.6 percentage points. Self-assessment is exactly the corrupted-signal regime of Theorem 1: the robustness of the loop to imperfect tests stems from grounding in execution, not from tolerance to arbitrary noise, and the execution signal cannot be substituted by model judgment.

Coverage: sparser generated test suites. Lowering the coverage threshold τ_{cov} from its default of 0.9 to 0.7 or 0.5 has only a very small effect on performance, with $\bar{\phi}$ remaining above 92. At the more aggressive setting $\tau_{\text{cov}} = 0.3$, performance decreases to 87.5, corresponding to an overall drop of 5.7 percentage points relative to the default configuration. Degradation is graceful rather than brittle: the loop does not depend on a complete test suite, only on an execution-grounded one.

Implication for recursive self-improvement. Execution grounding matters more than signal density. Thinning the execution signal degrades performance gracefully: even the sparsest test suite ($\tau_{\text{cov}}=0.3$, 87.5) still outperforms the LLM judge (73.6) by 13.9 points. A sparse but execution-grounded signal is therefore a better optimization target than a dense judgment-based one. This points to a general principle: in recursive self-improvement, the agent may propose its own feedback signal, provided that signal is grounded in execution. It also explains why code is a privileged substrate for self-improvement. Code is executable, testable, and iterable by nature, so every attempt yields grounded feedback, such as failing assertions, type errors, and stack traces, which the agent can use as an improvement signal without any external supervision.

F Prompt Templates

All prompts in this appendix are reproduced verbatim from our implementation; they refer to the specification as “documentation”. The error-driven refinement process consists of two stages: diagnosis and prescription.

F.1 Stage 1: Diagnosis Prompt

Diagnosis Prompt Template

You are analyzing test failures to diagnose why an LLM failed to generate correct code.

Entity Information:

- Name: {entity_name}
- Type: {entity_type} (function/class/method)
- Current Documentation:

{current_doc}

Test Failures:

The generated code failed the following tests:

{error_messages}

Task:

Analyze the test failures and diagnose:

1. What went wrong? Identify the specific errors (type mismatches, incorrect outputs, missing logic, etc.)
2. Why did the LLM fail? What information is missing or ambiguous in the documentation that caused the LLM to misunderstand the requirements?
3. What knowledge gap exists? What does the LLM need to know to generate correct code?

Output:

Provide a concise diagnosis in the following format:

- Error Type: [brief description]
- Root Cause: [what the LLM misunderstood]
- Missing Information: [what needs to be clarified]

F.2 Stage 2: Prescription Prompt

Prescription Prompt Template

You are refining documentation for a code entity based on a diagnosis of test failures.

Entity Information:

- Name: {entity_name}
- Type: {entity_type}
- Current Documentation:

{current_doc}

Diagnosis:

{diagnosis_output}

Task:

Refine the documentation to address the diagnosed issues. The refined documentation should:

1. Directly address the identified knowledge gap
2. Clarify the ambiguous or missing information
3. Be precise enough for an LLM to generate correct code
4. Preserve all existing correct information

Output:

Provide only the refined documentation, without any explanation.

F.3 Code Generation Prompt

Code Generation Prompt Template

```
Generate Python code for all entities in this module based on their
documentation.
Module: {module_name}
Documentation:
{all_entity_documentation}
Requirements:
    • Implement each entity exactly according to its documentation
    • Ensure all cross-entity calls are consistent
    • Do not add functionality not specified in the documentation

Output:
Provide the complete Python implementation for all entities, without any
explanation or markdown.
```

F.4 Test Case Generation Prompt

Test Case Generation Prompt Template

```
You are generating test cases for a code entity. Each test case includes
inputs, expected outputs, and a description of what is being tested.
Entity Information:
- Name: {entity_name}
- Type: {entity_type} (function/class/method)
- Signature: {signature}
- Source Code:
{source_code}
Dependencies:
The following entities are called by this entity:
{dependency_signatures}
Task:
Generate diverse test cases that cover:
    1. Normal cases: typical usage scenarios
    2. Edge cases: boundary conditions, empty inputs, single elements
    3. Corner cases: special values, type boundaries

Requirements:
    • Each test case must include input, expected output, and description
    • The description should explain what aspect of the entity is being
      tested
    • Cover different code paths and branches
    • Include cases that test error handling if applicable

Output Format:
Provide test cases as a JSON list:
[
  {
    "input": {"args": [...], "kwargs": {...}},
    "expected_output": ...,
    "description": "Tests that ..."
  },
  ...
]
Provide only the JSON list, without any explanation.
```

F.5 Iterative Refinement Prompt (Baseline)

Iterative Refinement Prompt Template

```
You are refining documentation for a code entity based on test failures.
Entity Information:
- Name: {entity_name}
- Type: {entity_type}
- Current Documentation:
  {current_doc}
Test Failures:
The generated code failed the following tests:
{all_error_messages}
Task:
Refine the documentation to fix the issues revealed by the test failures.
The refined documentation should enable an LLM to generate correct code.
Output:
Provide only the refined documentation, without any explanation.
```

Note: Unlike SPECFORGE’s two-stage diagnosis-prescription process, the iterative baseline conditions on *all* error messages simultaneously in a single refinement step, which can lead to the signal corruption phenomenon described in §3.3.

F.6 Commercial Agent Prompts

We evaluate commercial coding agents (Cursor and Claude Code) on the task of generating specifications for code entities. Below are the prompt templates for each configuration.

F.6.1 Base Configuration Prompt

Commercial Agent Base Prompt

```
Generate documentation for all entities in this module. The documentation
should enable an LLM to correctly implement each entity.
Module: {module_name}
Entities (signatures and reference code):
{all_entity_signatures_and_code}
Requirements:
  • Describe the purpose, parameters, return values, and behavior of each
    entity
  • Include any edge cases or special conditions
  • Ensure documentation is precise enough for code generation
```

F.6.2 +Topo Configuration Prompt

Commercial Agent +Topo Prompt

```
Generate documentation for all entities in this module. The documentation
should enable an LLM to correctly implement each entity.
Module: {module_name}
Dependency Structure (for context):
The following shows the call dependencies between entities (callee ←
caller):
{dependency_graph}
Entities (signatures and reference code):
{all_entity_signatures_and_code}
Requirements:
```

- Describe the purpose, parameters, return values, and behavior of each entity
- Use the dependency information to ensure consistency across related entities
- Ensure documentation is precise enough for code generation

F.6.3 +Test Configuration Prompt

Commercial Agent +Test Prompt

Generate documentation for all entities in this module. Use a test-driven approach to verify documentation quality.

Module: {module_name}

Entities (signatures and reference code):

{all_entity_signatures_and_code}

Workflow:

1. Generate documentation for each entity
2. Generate test cases based on your documentation
3. Use the tests to verify that your documentation is unambiguous and complete
4. If tests reveal ambiguities, refine the documentation
5. Repeat until documentation passes self-verification

Requirements:

- Documentation should be testable and verifiable
- Generate comprehensive test cases covering normal and edge cases
- Refine documentation based on test feedback

F.6.4 +Topo+Test Configuration Prompt

Commercial Agent +Topo+Test Prompt

Generate documentation for all entities in this module. Use dependency information and test-driven verification.

Module: {module_name}

Dependency Structure (for context):

The following shows the call dependencies between entities (callee ← caller):

{dependency_graph}

Entities (signatures and reference code):

{all_entity_signatures_and_code}

Workflow:

1. Generate documentation for each entity, using dependency context for consistency
2. Generate test cases based on your documentation
3. Use the tests to verify documentation quality
4. Refine documentation if tests reveal ambiguities

Requirements:

- Use dependency information to ensure cross-entity consistency
- Documentation should be testable and verifiable
- Generate comprehensive test cases for self-verification

G Baseline Implementation Details

Static Methods. **RepoAgent** [10]: We use the official implementation with default hyperparameters. Documentation is generated in a single pass without test feedback. **DocAgent** [11]: We use the official implementation. Multi-agent collaboration generates documentation iteratively until convergence.

Commercial Agents. We evaluate **Cursor** and **Claude Code** under four configurations: (1) *Base*: generate specifications for all entities in a single session; (2) *+Topo*: provide the dependency graph as context for one-shot specification generation; (3) *+Test*: prompt the agent to generate test cases from its specifications, then iteratively refine code until tests pass, with a budget of 50 manual action approvals (agents typically terminate before exhausting the budget); (4) *+Topo+Test*: combine both strategies. Cursor supports Gemini-2.5-Flash, Gemini-3.1-Pro, and Claude-4.5-Sonnet but not Qwen3-235B-A22B or GPT-4o; Claude Code only supports Claude models. Prompts are provided in Appendix F.6.

Iterative Refinement. Algorithm 5 presents the iterative baseline. The **Iterative** variant selects entities randomly, while **Iterative+Topo** follows reverse topological order.

Algorithm 5 Iterative Refinement Baseline

Require: Specification bundle $\mathcal{D}$, max iterations T

```

1: for  $t = 1$  to  $T$  do
2:    $\mathcal{F} \leftarrow$  entities with failing tests
3:    $e \leftarrow \text{SELECTENTITY}(\mathcal{F})$  ▷ Random or Topo order
4:    $\mathbf{e} \leftarrow$  error messages for  $e$ 
5:    $d'_e \leftarrow \text{REFINE}(d_e, \mathbf{e})$ 
6:    $\mathcal{D} \leftarrow \mathcal{D}[e \mapsto d'_e]$ 
7: end for
```

Prompt Optimization Methods. We adapt two state-of-the-art prompt optimizers to our setting by treating the entire specification bundle $\mathcal{D} = (d_1, \dots, d_n)$ as a single optimizable prompt.

GEPA [13]. We implement a DSPy module where the input is the module’s source code and the output is the complete specification bundle for all entities. The metric function evaluates each candidate by generating code from the specifications, executing the same LLM-generated test suite that SpecForge receives during search (§J; ground-truth tests are withheld from all methods), and returning both a scalar score (solve rate) and textual feedback (formatted error messages from failing tests). We run GEPA with `auto="heavy"` and set `max_metric_calls=50` to match SpecForge’s budget $B = 50$. Each metric call corresponds to one full spec→code→test evaluation cycle.

ACE [14]. We implement ACE’s Generator-Reflector-Curator pipeline adapted for specification optimization. The Generator produces specifications from source code conditioned on the current playbook. The Reflector analyzes test failures to extract lessons (e.g., “entity X’s documentation lacks return type specification”). The Curator converts lessons into delta bullets appended to the playbook. Each iteration performs one full spec→code→test evaluation cycle against the same LLM-generated test suite used by SpecForge (§J); we run for 50 iterations to match SpecForge’s budget. Following the original implementation, we use execution feedback (test pass/fail + error messages) without ground-truth labels.

Critically, both methods optimize the specification bundle monolithically—they cannot attribute failures to individual entities or reason about inter-entity dependencies. This design choice is intentional: it evaluates whether state-of-the-art prompt optimization can succeed without the structural decomposition that SpecForge introduces.

DocSearch. We use the official implementation [24] with its default hyperparameters, guided by the same LLM-generated test suite that SpecForge receives (§J) and evaluated on the same hidden ground-truth tests. All results are obtained on the version of DEVEAL+ used in this paper, which is a deliberately more challenging setup than the one in the original DocSearch paper, curated to stress-test SpecForge: modules are larger and more densely tested (12.4 entities per module and 3.1 tests per entity, vs. 11.4 and 2.8).

H DevEval+ Benchmark

This appendix provides comprehensive details about the DEV EVAL+ benchmark used for evaluation.

H.1 Overview

DEV EVAL+ is a curated benchmark for module-level code generation, designed to evaluate methods under complex dependency structures. The benchmark comprises **20 Python modules** containing **247 entities** (functions, classes, and methods) validated by **760 test cases**. Modules are sourced from two origins: (1) DevEval [5], an existing repository-level code generation benchmark, and (2) additional open-source GitHub repositories selected to increase domain diversity and structural complexity.

Key statistics of the benchmark:

- **Average dependency depth:** 4.2 levels
- **Maximum dependency depth:** 10 levels
- **Average entities per module:** 12.4
- **Average tests per entity:** 3.1

H.2 Module Composition

Table 3 presents the complete list of modules in DEV EVAL+, including their source, domain, and structural statistics.

Table 3: Complete module list of DEV EVAL+. **Source:** D = DevEval, G = GitHub. Modules marked with † are included in the cross-language subset DEV EVAL+-JAVA.

Module	Source	Domain
arxiv_digest†	D	Academic/NLP
hybrid_images	D	Image Processing
textcnn	D	Deep Learning/NLP
chakin	D	NLP/Embeddings
geotext†	D	Geography/NLP
hone†	D	Data Processing
lice†	D	Utilities
particle-swarm-optimization†	D	Optimization
readtime†	D	Text Processing
stocktrends†	D	Finance
interpreter†	G	Programming Languages
croniter†	G	Scheduling
peakutils†	G	Signal Processing
pyfaidx	G	Bioinformatics
lexer†	G	Compilers
schedule†	G	Scheduling
patternpy†	G	Finance
schema†	G	Data Validation
parser†	G	Utilities
streaming-indicators†	G	Time Series

H.3 Data Collection and Curation

Source Selection Criteria. We selected modules based on the following criteria:

1. **Non-trivial dependencies:** Modules must contain at least 3 levels of call depth to exhibit meaningful output coupling effects.
2. **Testability:** All entities must have well-defined input-output behavior amenable to automated testing.

3. **Self-containment:** Modules should minimize external dependencies beyond the Python standard library to ensure reproducible evaluation.
4. **Domain diversity:** We balanced modules across application domains to avoid benchmark-specific overfitting.

From DevEval. We selected 10 modules from DevEval [5]: `arxiv.digest`, `hybrid.images`, `textcnn`, `chakin`, `geotext`, `hone`, `lice`, `particle-swarm-optimization`, `readtime`, and `stocktrends`. These modules span diverse domains including NLP, image processing, finance, and optimization. DevEval modules provide high-quality, human-verified test cases. We retained the original test suites without modification.

From GitHub. To increase structural complexity and domain coverage, we collected 10 additional modules from open-source GitHub repositories: `interpreter`, `croniter`, `peakutils`, `pyfaidx`, `lexer`, `schedule`, `patternpy`, `schema`, `parser`, and `streaming-indicators`. The data collection process consists of two phases: repository selection and quality assurance.

In the **repository selection** phase, we searched GitHub for Python repositories with permissive licenses (MIT, Apache 2.0, BSD) and non-trivial dependency structures (call graph depth ≥ 3). Crucially, we required candidates to include existing ground-truth test suites, as this ensures reliable evaluation without introducing annotation artifacts from manual test construction.

In the **quality assurance** phase, two authors independently set up the environment for each candidate repository and executed its test suite to verify functionality. Following this sanity check, annotators reviewed each test case for correctness and quality, verifying that tests accurately capture intended behavior rather than implementation quirks. Repositories were included only with unanimous agreement on test suite quality.

H.4 Dependency Structure

The benchmark exhibits significant structural complexity to induce meaningful output coupling effects. Across all 20 modules, the average dependency graph depth is 4.2 levels with a maximum of 10 levels. Approximately 39% of entities are leaf nodes (no callees), while 30% are root nodes (no callers). This structure ensures that refinements to callee specifications can propagate through multiple levels, creating the conditions necessary to observe and address the whack-a-mole phenomenon.

H.5 Test Suite Construction

All test cases in DEV-EVAL+ are sourced from ground-truth test suites: DevEval modules include human-verified tests from the original benchmark, while GitHub modules contain tests from their respective repositories. We performed verification on top of the upstream tests, with lightweight harness adjustments (import-path fixes and fixture rewiring) to integrate them into our evaluation pipeline.

Quality Assurance. All test cases were validated through:

1. **Execution verification:** Every test passes on the reference implementation.
2. **Correctness review:** Two authors independently verified that tests capture intended behavior rather than implementation quirks.
3. **Independence check:** Tests for entity e_i do not depend on implementation details of entities other than e_i 's direct callees.

H.6 Cross-Language Subset (DevEval+-Java)

For cross-language transfer experiments (§4.5.1), we constructed DEV-EVAL+-JAVA by translating test harnesses for a 16-module subset (marked with † in Table 3). This subset contains 208 entities validated by 680 test cases.

Translation Process.

1. **Test input preservation:** Input values were preserved exactly, with Python data structures mapped to Java equivalents (e.g., `list` $\rightarrow$ `ArrayList`, `dict` $\rightarrow$ `HashMap`).
2. **Expected output adaptation:** Outputs were adapted for Java type system while preserving semantic equivalence.
3. **Harness rewriting:** Test harness code was manually rewritten in JUnit 5 format.
4. **Verification:** Two authors independently verified semantic equivalence of translated tests.

Subset Selection. The 16 modules were selected by excluding 4 modules whose Python implementations depend on libraries without direct Java equivalents, with the remaining modules satisfying:

- Balanced source distribution (7 from DevEval, 9 from GitHub)
- Representative depth distribution (range: 3–10, mean: 4.3)
- Domain diversity (no more than 2 modules per domain)

I New Feature Implementation Setup (SWE-DEV)

This section details the new feature implementation experiment of §4.5.2: how SWE-DEV tasks and their PRDs are constructed, what each condition receives, and what the comparison tests.

Task construction. Each SWE-DEV instance is built from an open-source PyPI package with an executable, developer-written test file [9]. The test file is run under a tracer to obtain the call tree of the source functions it triggers; target functions are selected from this tree, and their implementations are removed from the repository. Difficulty follows call-tree depth: easy instances have depth 1–2, and hard instances have depth ≥ 3 and typically require implementing five or more interdependent functions. The test file is the evaluation oracle.

How the PRD is generated. The PRD shipped with each instance is written by GPT-4o in two parts [9]. A module-level description is synthesized from the test file. A function-level description of each target is obtained by augmenting the target’s original docstring with implementation-level details, with the prompt conditioned on the full code of its file, including the target function itself. The PRD concatenates the module description with, for each target, its signature, the augmented docstring, and a list of related functions (DEPENDENCIES). The PRD is therefore derived from the tests and the reference implementation, and it describes only the functions to be implemented: existing functions that the targets call appear in the dependency list by name only, without any description of their behavior.

Conditions. All conditions receive the same stubbed package source, in which the target functions are removed, and are scored by Pass@1 on the instance’s test file following the SWE-DEV protocol. They differ only in the accompanying text:

- **PRD+Source:** the full SWE-DEV PRD described above.
- **Spec+Source** (RepoAgent, Claude Code*, ACE, SPECFORGE): the signature and original docstring of each target function, which state *what* the new feature must provide, together with the specifications of the existing codebase produced by the method.

Specification generation and masking. Each method generates specifications on the complete package, so that the codebase is executable, as it is in real development before a feature is added, and every entity has a reference behavior from which SPECFORGE can derive test oracles. Deleting the target functions from a complete module beforehand would leave the module unable to run, since the remaining code still references them, which would rule out execution-based optimization methods such as SPECFORGE. Before the implementation step, we remove the specifications of the target functions. No method observes the instance’s evaluation test file.

Additional results analysis. The two conditions trade task-specific information against codebase-level information. The PRD describes the target functions in detail but says nothing about the behavior of the existing code they build on; the Spec+Source conditions describe the target functions only

minimally but describe the whole existing module. New features are rarely self-contained: SWE-DEV targets call into existing functions, constants, and exception types, and a correct implementation must use them exactly as they behave. For example, in the `vulture` configuration task, the PRD requires `_check_input_config` to validate values against `DEFAULTS` and to raise `InputError`, but never states which keys `DEFAULTS` contains or how `InputError` is constructed. We refer to this missing knowledge as the *semantic gap* of the PRD. The experiment therefore asks whether a better semantic understanding of the existing module lets an agent implement a high-quality feature from only the minimal information that defines it. Because the PRD condition receives strictly more target-specific information, derived from the reference implementation, the comparison is conservative for the Spec+Source conditions.

J Test Case Generation

This appendix describes the LLM-based test case generation process used to guide SPECFORGE’s search without access to ground-truth tests.

J.1 Overview

SPECFORGE operates as a self-improving framework where the search is guided by *LLM-generated* test cases rather than ground-truth tests. This design choice reflects realistic deployment scenarios where comprehensive test suites may not exist. The key insight is that even imperfect tests provide valuable diagnostic signals—test failures reveal knowledge gaps in the specifications, enabling targeted refinement.

The test generation process follows a generate-then-validate approach:

1. **Test generation:** The LLM generates complete test functions—including inputs, assertions, and error messages—based on entity signatures and reference source code.
2. **Validation:** Each generated test is executed on the reference implementation. Only tests that pass are retained; those that fail (due to incorrect assertions or invalid inputs) are discarded.

Two additional filters are applied post-validation: (i) tests that reach the target entity only via transitive calls to other functions are discarded, ensuring each retained test directly exercises its target; and (ii) tests with identical normalized inputs and assertions are deduplicated. This ensures test correctness by construction: every retained test is consistent with the reference implementation and contributes a distinct diagnostic signal.

J.2 Coverage-Driven Test Generation

For each entity e_i , we employ a coverage-driven generation strategy. The LLM generates test functions iteratively until branch coverage exceeds a target threshold τ_{cov} . Each iteration prompts the LLM to generate tests covering:

- **Normal cases:** Typical usage scenarios with common input values
- **Edge cases:** Boundary conditions (empty inputs, single elements, maximum values)
- **Corner cases:** Special values that exercise specific code paths
- **Uncovered branches:** When coverage feedback is available, tests targeting uncovered code regions

The generation loop terminates when (1) branch coverage reaches τ_{cov} (primary criterion), or (2) the per-iteration branch coverage gain stays below a positive threshold ε for P consecutive iterations (no-progress early-stop). The loop is bounded because (i) progress iterations (gain $\geq \varepsilon$) number at most $\lceil 1/\varepsilon \rceil$, since branch coverage is bounded above by 1.0 and each contributes at least ε ; and (ii) no-progress iterations (gain $< \varepsilon$) cannot accumulate indefinitely because the P -th consecutive one triggers the early-stop.

The generation prompt (provided in Appendix F) includes:

- Entity signature and type information
- Reference source code for understanding expected behavior

- Dependency signatures to ensure consistency with callees
- Coverage feedback as (line number, source snippet) pairs extracted from `coverage.py`'s branch-coverage output, indicating which lines remain uncovered

J.3 Quality Considerations

While the validation step ensures test correctness, LLM-generated tests may have quality limitations that motivate our diversified error batch sampling (§3.3).

Error Message Quality Variation. Since the LLM writes complete test functions, the diagnostic value of error messages depends entirely on how each assertion is written. Some tests include informative messages (e.g., `assert result == [1,2,3], f"Expected [1,2,3] got {result}"`) that directly reveal the failure mode, while others use bare assertions (e.g., `assert result == expected`) that produce only `AssertionError` with no actionable signal. This heterogeneity is inherent to LLM-generated tests and cannot be controlled a priori.

Input Quality Variation. Not all generated test inputs are equally informative. Some inputs may exercise redundant code paths, miss critical edge cases, or cluster around similar patterns. By conditioning refinements on different test subsets, diversified error batch sampling increases the probability of encountering informative failure signals.

External Dependencies. External dependencies (filesystem, network, time, randomness) are detected via static analysis of import statements and AST patterns; the LLM is then prompted to isolate them through `pytest` fixtures (mocks for filesystem and network calls, frozen-time wrappers for time-dependent code) so that test outcomes remain deterministic and reproducible across runs.

J.4 Hyperparameters

We use the following hyperparameters for test generation:

- **LLM:** GPT-4o (fixed across all experiments for consistency)
- **Temperature:** $\tau = 0.7$ for input diversity
- **Coverage threshold:** $\tau_{\text{cov}} = 90\%$ branch coverage
- **No-progress patience:** $P = 2$ consecutive iterations within $\varepsilon = 0.01$ coverage gain

K Additional Related Work

Behavioral specifications as vertical knowledge. Behavioral specifications can be viewed as a form of vertical knowledge: repository-specific facts about what each program entity does, which are absent from a model's weights and must be supplied to it at inference time. A broad line of work studies how such domain knowledge is acquired, structured, and used by LLMs. In cyber threat intelligence (CTI), LLMs have been used to construct knowledge graphs from unstructured threat reports [60], retrieval-augmented LLMs have been benchmarked over heterogeneous CTI sources [61], and CTI corpora have been organized into agent-native scaffolds for direct consumption by agents [62]. Related efforts parse hybrid system logs into structured templates [63] and translate natural language into first-order logic that symbolic reasoners can consume [64], and documentation optimized as repository-specific playbooks for coding agents [24] brings this view to software. These works share the premise that knowledge should be represented in the form its consumer can use. They differ from SPECFORGE in how the representation is judged and improved: knowledge graphs, retrieval corpora, and parsers are typically constructed once and evaluated by extraction or answer accuracy, whereas SPECFORGE treats the knowledge artifact itself as the object of recursive improvement, judges it by the behavior it induces in the consuming agent, and revises it until that behavior is verified correct.

L Compute Resources Used

All evaluations involving proprietary models (e.g., GPT-4o, Gemini-2.5-Flash, Gemini-3.1-Pro, Claude-4.5-Sonnet), were conducted through the APIs. These calls were issued from standard desktop machines using CPU-only execution, with no reliance on local GPU hardware, which makes the approach easy to reproduce in widely available computing environments.

For open-source models (e.g., Qwen), inference was run on a single node equipped with eight NVIDIA H100 GPUs (80 GB each). Experiments were executed sequentially on this node to maintain consistent and reproducible runtime conditions across different model families.

Overall, this setup highlights the modest computational demands of our method and shows that our approach remains practical and scalable even in resource-constrained settings when hosted APIs are used.

M Use of LLMs

In this work, large language models (LLMs) were primarily used for inference in the specification generation experiments. Specifically, they were employed to generate responses under different prompting and evaluation settings, which enabled us to assess the effectiveness of our method across a range of generation tasks and model families.

In addition to their role in experimental inference, LLMs were also used as a writing assistant during manuscript preparation. In particular, they helped improve the clarity, fluency, and overall readability of the paper by polishing sentence-level expressions and refining the presentation of technical descriptions. All substantive technical content, experimental design, analysis, and conclusions were determined and verified by the authors.

N Limitations

The recursion in SPECFORGE is over specifications, not over the improver: the search procedure itself is fixed, and the loop requires a reference implementation against which self-generated tests are validated. Our evaluation of the loop covers 20 Python modules; other languages appear only as a downstream target in the cross-language transfer experiment (§4.5.1).

O Broader Impact

This work studies how a coding agent can improve the behavioral specifications in its harness: natural-language descriptions of program behavior that are written for code-generation models and verified by execution, in contrast to human-facing documentation. Its main potential benefit is to improve the reliability, controllability, and practical utility of LLM-based coding agents in tasks such as code synthesis, migration, translation, and feature development, thereby reducing iteration cost and improving software quality. At the same time, more effective autonomous coding systems may lower the barrier to producing software at scale, including in harmful settings. The self-improvement studied here is bounded by design: it changes neither model weights nor the search procedure, and every candidate revision is evaluated by execution against tests validated on the reference implementation. Our method does not remove the need for verification or human oversight; it operates entirely through test-based feedback and does not introduce capabilities beyond improving the behavioral precision of specifications for downstream code generation. We therefore view its societal implications as broadly aligned with those of existing LLM-based software engineering systems, with the main positive contribution being better-structured, more evaluable interaction between codebases and generative models.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [\[Yes\]](#)

Justification: The main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope.

Guidelines:

- The answer [\[N/A\]](#) means that the abstract and introduction do not include the claims made in the paper.
- The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [\[No\]](#) or [\[N/A\]](#) answer to this question will not be perceived well by the reviewers.
- The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
- It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [\[Yes\]](#)

Justification: We discuss the limitations of the work in the Limitations section (Appendix [N](#)) and the sensitivity analysis.

Guidelines:

- The answer [\[N/A\]](#) means that the paper has no limitation while the answer [\[No\]](#) means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate “Limitations” section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren’t acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [\[Yes\]](#)

Justification: All theoretical results state assumptions explicitly and provide complete proofs in the appendix.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [\[Yes\]](#)

Justification: We provide detailed experimental information in the appendix and release source code that can reproduce the reported results.

Guidelines:

- The answer [\[N/A\]](#) means that the paper does not include experiments.
- If the paper includes experiments, a [\[No\]](#) answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general, releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
 - (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
 - (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
 - (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
 - (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We release code and data at <https://github.com/ccsnow127/SpecForge>.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (<https://neurips.cc/public/guides/CodeSubmissionPolicy>) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: We specify all training and test details in the experiments section and the appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We repeated experiments across multiple random seeds and report mean and standard deviation in the appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: We provide sufficient information on computational resources in the appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics <https://neurips.cc/public/EthicsGuidelines>?

Answer: [Yes]

Justification: The research conducted in the paper conforms, in every respect, with the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: We discuss potential positive and negative societal impacts in the Broader Impact section.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper poses no such risks.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: We properly credit the creators of all assets used in the paper and respect the licenses and terms of use.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, paperswithcode.com/datasets has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We document the new benchmark in the appendix and release it under permissive licenses.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: Our paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: Our paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [\[Yes\]](#)

Justification: We describe the LLM usage in the experiments section and the appendix.

Guidelines:

- The answer [\[N/A\]](#) means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.