"CODE IS CHEAP, SHOW ME THE SPECIFICATION": IMPROVING LLMS FOR END-TO-END VERIFIED CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Formal verification offers strong correctness guarantees for software, but writing precise *specifications* is often as difficult as proving them. Moreover, weak specifications can satisfy a verifier without capturing meaningful behavior, yielding formally verified yet empty guarantees. Contemporary LLMs lack the ability to generate verified code based on just natural language descriptions. Existing benchmarks primarily focus on evaluating specifications only indirectly, typically through testing or one-sided implication checks over simple pre-/post-conditions. We take a step toward closing both gaps for Rust paired with Verus, a verifier that checks Rust code against developer-written specifications. In this work, we develop techniques for *end-to-end training and evaluation of verified code generation in Rust*. We 1. introduce SpecBench, a specification-evaluation benchmark with SMT-based equivalence checking that definitively judges generated Verus specifications; 2. train and release VerusLM-4B-Codex , a 4B-parameter model trained from Qwen3-4B (Instruct), trained on end-to-end pipeline for Verus Prover to learn end-to-end verified code generation.

1 INTRODUCTION

A *verified program* comprises three mutually dependent artifacts: (a) Exec *executable* code that implements a task, (b) Spec a formal *specification* that precisely captures the intended behavior, and (c) Proof a correctness *proof* that the code satisfies the specification, discharged mechanically by a verifier. *End-to-end autoformalization* is the problem of generating all three from a developer's NL *natural-language* description. Figure 1 illustrates these components for a simple tree-maximum program: the Spec encodes the functional contract, the Proof provides the invariants and assertions that justify correctness, and the Exec implementation is the executable Rust code.

Large language models have dramatically accelerated code generation, yet they provide no correctness guarantees; LLM-generated code is as defect-prone as human-written code (Cotroneo et al., 2025). Testing covers only the inputs a developer enumerates. Formal verification provides a categorically stronger assurance: correctness for all valid inputs. This guarantee support high assurance systems like CompCert (verified C compiler) (Leroy et al., 2016), seL4 (verified microkernel) (Klein et al., 2009), and Anvil (verified Kubernetes controllers) (Sun et al., 2024), but at enormous cost. For every line of Rust code in Atmosphere Chen et al. (2023), roughly 7.5 lines of specification and proof were written. Scaling formal verification therefore demands automating not just code generation, but the full verified-program pipeline.

The three subproblems, implementation, specification, and proof, are tightly coupled and must be addressed together. Proof-generation systems (Wilf et al., 2025; Gloeckle et al., 2026; Li et al., 2026) synthesize proofs when code and specifications are given. Autoformalization systems (Wu et al., 2022; Weng et al., 2025; Li et al., 2024; Ma et al., 2025) translate informal descriptions into formal representations but do not produce machine-checked proofs for production programs. General-purpose LLMs can be prompted to attempt all three jointly, as evaluated in VerifyThisBench (Deng et al., 2025), but achieve very low end-to-end accuracy (Verified% ≤ 7.32 in Table 2). To our knowledge, VerusLM-4B-Codex is one of the first system trained to jointly generate code, specification, and proof from natural language for a mainstream programming language.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

```
verus! {
// Compute the maximum element of a non-empty vector.

spec fn max_seq(s:  Seq<i32>) -> i32
    decreases s.len(),
{ ... }

proof fn lemma_max_extend(s:  Seq<i32>, i:  int)
    requires 1 <= i < s.len(),
{ assert( s.subrange(0, i+1).drop_last() =~= s.subrange(0, i) ); }

fn max_vec(v:  &Vec<i32>) -> (result:  i32)
    requires v.len() > 0,
    ensures result == max_seq(v@),
{
    let (mut best, mut i) = (v[0], 1usize);
    while i < v.len()
        invariant 1 <= i <= v.len(), best == max_seq(v@.subrange(0, i as int))'
        decreases v.len() - i,
    {
        proof { lemma_max_extend(v@, i as int); }
        if v[i] > best { best = v[i]; }
        i += 1;
    }
    best
}} // verus!
```

Figure 1: An example Verus program computing the maximum over a vector. Colored annotations distinguish `natural language intent`, `formal specifications`, `proof hints`, and `executable code`.

Mainstream languages like Rust are being used in production systems software, including operating-system kernels, browsers, and safety-critical embedded software. *Verus* (Lattuada et al., 2023) is a SMT-based tool that verifies Rust programs with annotations: specifications and proofs are written directly in Rust syntax and checked by Z3, with no translation required. However, generating verified Rust with Verus remains hard, with even frontier general-purpose models achieving at most 7.32% Verified% on end-to-end generation in VerifyThisBench (Table 2). Most prior work on end-to-end generation or specification evaluation instead targets formal languages such as Lean (de Moura et al., 2015) or Dafny (Leino, 2010) that are separate from mainstream software development (Gloeckle et al., 2026; Thakur et al., 2025; Ye et al., 2025). This leaves a gap for verified-code generation in languages whose artifacts can be directly compiled and deployed. In this work we attempt to fill this gap and target end-to-end verified code generation in Rust.

Prior work has observed that writing a specification is a crucial and time-intensive process for ensuring the reliability of any system (Jain et al., 2025; Stoica et al., 2024). This is because specifications need to precisely formalize the intended behavior of a system and thus need much careful reasoning. They are also difficult to evaluate as the only real oracle to check their correctness is the intent in the developer's mind.

In this paper, we construct a benchmark, called SpecBench, for evaluating specifications by starting from real-world, ground-truth specifications in complex verified code repositories, and reconstructing the natural language intent. We then evaluate a generated specification by comparing it to the ground-truth, producing, if possible, a proof (checked by Verus) that guarantees their equivalence. We also show (Section 5.2.2) that the absence of a proof does not necessarily imply incorrectness of the specification. Therefore, we add an additional LLM-as-a-judge stage to judge whether the generated specification is coherent and semantically aligned with the natural language intent. SpecBench is (i) grounded in real-world, complex codebases, (ii) is hard for current strong models to solve (Table 2), and yet (iii) reveals a substantial gap between the open and closed frontier models (Table 3).

We also propose a training pipeline to train LLMs to jointly generate Exec, Spec, Proof from NL. Our two-stage pipeline uses a combination of SFT and RL to teach Verus syntax and to improve the semantic alignment of its responses, respectively. The SFT stage uses selectively masked configurations of (NL, Exec, Spec, Proof), which teach the model to write each in isolation and in combination. Then, online RL runs in a multi-turn verifier environment where we apply a series of checks: (a) anti-cheat checks, (b) the Verus verifier, and where specifications have to be written,

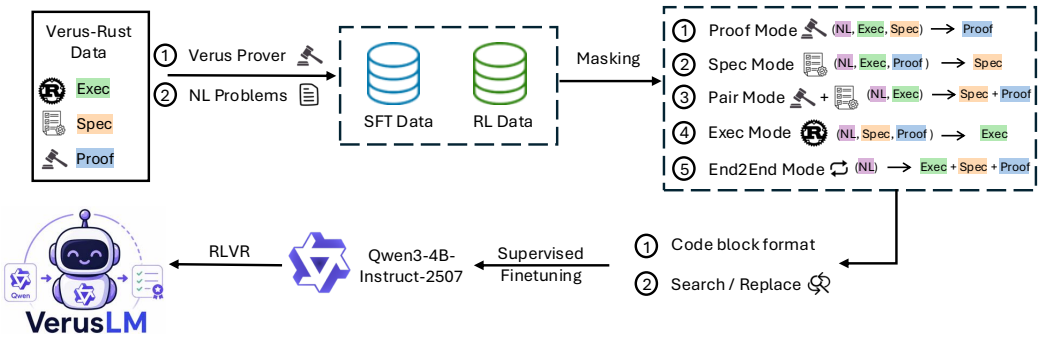

Figure 2: **VerusLM-4B-Codex training pipeline.** (1) NL intent is generated from ground-truth verified code. (2) Five masking modes decompose each program into sub-tasks of increasing difficulty. (3) SFT targets are encoded as SEARCH/REPLACE edit blocks for partial-mask modes and full Rust code blocks for END2END-MODE. (4) RLVR rollouts pass a gated reward pipeline (anti-cheat gates, Verus score, equivalence harness) with structured verifier feedback across up to $T = 3$ turns.

(c) an SMT-based equivalence harness with respect to the ground-truth. We use this pipeline to train VerusLM-4B-Codex showcases improvements on benchmarks ranging from proof writing, spec generation and end to end autoformalization and verified code generation. In summary, we make the following contributions:

(1) We introduce SpecBench, a benchmark for specification generation derived from real-world verified Rust repositories, together with an evaluation methodology based on formal equivalence checking and LLM-assisted semantic judgment. We show that SpecBench is hard for current frontier models, and also reveals a substantial gap between open and closed frontier models.

(2) We introduce VerusLM-4B-Codex , the first language model trained for end-to-end verified code generation in Rust and Verus. We show that it achieves strong performance on a variety of benchmarks covering proof generation, specification generation, and end-to-end verified code generation.

2 RELATED WORK

Autoformalization and theorem proving in mathematics. LLM-based autoformalization has made strong progress on theorem formalization (Wu et al., 2022; Weng et al., 2025), proof search (Polu & Sutskever, 2020; Han et al., 2022), and competition mathematics (Google DeepMind, 2024), aided by Mathlib (Mathlib community, 2020), a shared library of thousands of machine-verified lemmas. Verified software has no analogous resource: program specifications encode the behavioral contract of a specific implementation and must be authored from scratch, tightly coupled to both the code structure and the invariants required by downstream proofs. This coupling makes autoformalization for software a harder and qualitatively different problem.

Verified code generation with LLMs. Verina (Ye et al., 2025) and CLEVER (Thakur et al., 2025) benchmark verified code generation in Lean 4; WybeCoder (Gloeckle et al., 2026) and Goedel-Code (Li et al., 2026) generate or search for Lean (de Moura et al., 2015) proofs using frontier models or trained provers. All face the same practical limitation: Lean is a proof assistant, not a system programming language, and verified Lean artifacts cannot be compiled or deployed. In parallel, a line of work focuses on Dafny (Leino, 2010), which focus on code generation in Dafny. Laurel (Mugnier et al., 2025), DafnyPro (Banerjee et al., 2026) and Dafny-annotator (Poesia et al., 2024). However these approaches exclusively focus on inference style setups to improve proof writing. Our work targets Rust, which is already in production use in operating-system kernels, browsers, and safety-critical systems. Secondly, Dafny does not compile directly to native binaries, making Verus a secure tool for verified code generation.

Verified Rust code generation with LLMs. Verus (Lattuada et al., 2023) verifies Rust programs via SMT-based reasoning, with specifications and proofs written directly in Rust syntax. Prior systems targeting it are uniformly proof-only: SAFE (Chen et al., 2025) bootstraps proof generation; PSV (Wilf et al., 2025) trains spec-to-code-and-proof through rejection fine-tuning while explicitly avoiding online RLVR; VeruSyn (Di et al., 2026) fine-tunes a 32B model on agent-collected proof trajectories; and AutoVerus (Yang et al., 2025) and VeruSAGE (Yang et al., 2026) are inference-time

multi-agent systems. In contrast our work focuses on training for end-to-end formalization from NL to Exec, Spec, and Proof.

Specification Generation and Evaluation. Specifications have no direct correctness oracle: a specification can type-check, pass the verifier, and still be semantically vacuous (`ensures(true)` is the canonical example). Testing-based evaluation (Endres et al., 2024; Lahiri, 2024; Chen et al., 2026) is limited to boolean predicates and is limited by test coverage. Verina (Ye et al., 2025) checks implication between generated and reference specifications, but implication is not well-defined for spec functions with non-boolean return types, and it cannot evaluate shared helper spec functions used across multiple pre- and post-conditions. We evaluate semantic equivalence, which is strictly stronger than either implication direction and handles arbitrary return types and helper functions.

3 Evaluating Autoformalization: SpecBench

We propose SpecBench benchmark that evaluates specification generation in isolation. Given a Verus program with all contents of the Spec masked, a model generate the specifications; SpecBench then judges whether the result is semantically equivalent to the ground truth rather than solely verifying.

3.1 Benchmark Construction

SpecBench contains 97 tasks drawn from real-world verified Rust projects: 84 from VeruSAGE (Yang et al., 2026) and 13 from VerifyThisBench (Deng et al., 2025). Each task is a Verus source file with all `spec fn` and `spec_checked_fn` bodies masked, and all `requires`, `ensures`, `recommends`, and `decreases` clauses stripped. We mask all the specifications present in the Verus code and relevant proof blocks. This is an important step as the model is capable of using the `requires` and `ensures` annotations in place to reconstruct the specification by copying annotations that were written after the spec was known. Our task span a range of difficulty, from simple array-bound invariants to recursive data-structure contracts and system-level controller proofs drawn from real Verus-verified projects.

3.2 Evaluation Pipeline

Stage 1 – Verification. The model receives the masked file and a natural-language problem description generated using Qwen3-30B-Thinking , and must return a SEARCH/REPLACE patch. The patch is applied and passed through five gates in sequence (the same gates are reused as RL training reward checks; Section **??**): 1. **(C1) Cheat Patterns.** The model may not introduce new statements that bypass the prover (e.g., `admit()`), nor increase the number of axiom declarations, relative to the masked input. The full forbidden set is listed in Appendix B.4. 2. **(C2) Scope and coverage.** Edits are confined to the masked region (enforced by byte-level equality via Lynette's `compare`), and every $(\textit{kind}, \textit{name})$ entry recorded for that region must reappear in the candidate, preventing silent drops, renames, or edits to surrounding context. The model receives up to five attempts, with structured Verus error feedback after each failure.

Stage 2 – Equivalence. A Stage 1 pass confirms only that the generated specification is verification-sufficient: the existing proof compiles against it. It does not confirm semantic equivalence, since proofs written with headroom may verify against a weaker-than-intended specification. Stage 2 constructs an *equivalence harness*: for each paired ground-truth specification S^* and model-generated specification $\hat{S}$, we emit a Verus lemma requiring $\forall \bar{x}.\ S^*(\bar{x}) \equiv \hat{S}(\bar{x})$. If all lemmas discharge with empty proof bodies, Stage 2 passes automatically. Otherwise the model is asked to fill the proof bodies, subject to the same anti-cheat gates as Stage 1.

Stage 3 – Coherency (metadata only). When Stage 1 passes but Stage 2 fails, an LLM judge Gemini-3-Flash-Preview receives the natural-language description, the ground-truth file, and the model-generated file, and returns an x/n ratio counting how many of the n relevant specification elements the model captured correctly. This score is diagnostic metadata and does not contribute to the benchmark total; it is reported separately as *Total Coherent Score* in Table 2.

Table 1: Human validation of SpecBench. Left: adequacy of the 97 task descriptions. Right: agreement between annotators and the Gemini-3-Flash-Preview judge on all 54 Stage-3 cases.

Task descriptions		**Stage-3 judge**	
Adequate (majority vote)	97/97	Pairs scored	54 (all)
Pairwise agreement	95.9–96.9%	Human–judge ρ	0.91
Leakage	0%	Vacuous specs penalized	4/4

3.3 Correctness and Validation of SpecBench

The three stages differ in how much they can be trusted. A Stage 2 pass is a machine-checked proof of equivalence and needs no further validation. Two components, however, depend on an LLM: the NL task descriptions, which models condition on, and the Stage-3 judge, which scores every output that verifies but cannot be proved equivalent. We validate both with three annotators who have intermediate Verus experience and who worked independently of each other and of the judge.

Task descriptions are adequate and leak-free. Annotators rated all 97 descriptions on whether each contains enough information to derive every `requires`/`ensures` clause without seeing the ground-truth code, and flagged any *leakage*, meaning proof-specific content such as a named invariant that would give the answer away. Individual annotators judged 95, 96, and 95 of the 97 descriptions adequate; by majority vote all 97 are adequate, pairwise agreement is 95.9–96.9%, and no task received more than one dissenting vote. No annotator found leakage in any description. SpecBench's difficulty therefore reflects the tasks themselves rather than under-specified or hint-laden descriptions.

The Stage-3 judge agrees with human annotators. Annotators scored all 54 (task, output) pairs in which any evaluated model passed Stage 1 and failed Stage 2. This is the complete set of cases where Stage 3 is invoked, not a sample. They used the same x/n rubric given to Gemini-3-Flash-Preview, without seeing the judge's scores. The Spearman rank correlation between the annotators' and the judge's scores is $\rho = 0.91$, so the judge ranks outputs by coherence much as humans do. The set includes four vacuous or near-vacuous specifications (e.g., `ensures true`); the annotators and the judge penalized all four. Gemini's judgments on exactly the outputs that verification alone cannot distinguish thus match human judgment, which is what Stage 3 requires.

Train/evaluation separation. The VeruSAGE repositories are excluded from the VerusLM-4B-Codex training corpus by construction to avoid data contamination in SpecBench.

4 Training a Model for End-to-End Verified Code Generation

We train VerusLM-4B-Codex from Qwen3-4B-Instruct-2507 (Qwen Team, 2025) in two stages: supervised finetuning (SFT) to teach Verus syntax and output formats, followed by multi-turn reinforcement learning with verifiable rewards (RLVR). The pipeline figure, data-construction details, and full hyperparameters are in Appendix B.

4.1 Data Preparation

We start from the VeruSyn training corpus (Di et al., 2026) (∼7M Verus-annotated Rust programs), keep only files that verify, and subsample 864K programs covering a range of program lengths and Verus keywords. For each program we synthesize a natural-language description NL with Qwen3-30B-Thinking-2507 that states the task and its pre-/post-conditions without referring to the proof. This yields quadruples (NL, Exec, Spec, Proof), which we split 70:30 into SFT and RL pools.

Using Lynette (Yang et al., 2026), we mask different subsets of each quadruple to produce five sub-tasks of increasing difficulty: Proof-Mode (→ Proof), Spec-Mode (→ Spec), Pair-Mode (→ Spec + Proof), Exec-Mode (→ Exec), and End2End-Mode (NL → all three). Each mode conditions on NL and the unmasked artifacts.

4.2 SUPERVISED FINETUNING

We finetune on all five sub-tasks with TRL (von Werra et al., 2020), encoding targets as SEARCH/REPLACE patches for partial-mask modes and as full Rust files for END2END-MODE. We train for three epochs at a context length of 16,384 tokens with AdamW (peak learning rate 2×10^{-5}, cosine decay to 2×10^{-6}).

4.3 REINFORCEMENT LEARNING WITH VERIFIER REWARDS

Data and curriculum. RL uses only PROOF-MODE, SPEC-MODE, and END2END-MODE, sampled 1:2:2 (14,254 instances in total) and interleaved round-robin in that dependency order so every gradient step sees a mix of modes. **Multi-turn environment.** Each rollout is an episode of up to $T = 3$ turns. A turn ends when a gate fails or the Verus check completes, and the environment returns structured verifier feedback for the next attempt. Only the terminating turn is rewarded: $R = \gamma^t S_t$ with $\gamma = 0.9$, where $t \in \{0, 1, 2\}$ is the terminating turn and S_t its score. This rewards later-turn recoveries while still favoring one-shot solutions.

Gated reward. The per-turn score reuses the SpecBench gate cascade (Section 3.2), ordered cheap-to-expensive. The anti-cheat gates carry asymmetric penalties: a cheat-pattern violation costs -2.0, while an honest attempt can earn up to $+1.5$ before verification, so no cheating strategy out-scores an honest partial attempt. A candidate that passes all gates receives the dense Verus score $\mu = V/(V + E)$, where V and E count discharged verification conditions and errors. PROOF-MODE stops there. SPEC-MODE and END2END-MODE add two phases after a full Verus pass. In Phase B the environment builds the SpecBench equivalence harness between the generated and ground-truth specifications and rewards automatic equivalence. If Verus cannot discharge it, Phase C asks the model to write the equivalence proof in the next turn. A spec that merely verifies therefore never earns the full reward.

Optimization. Rewards are sparse and high-variance, and rollouts come from a vLLM snapshot that lags the trainer by one update. We therefore combine a critic-free GRPO baseline (Shao et al., 2024) over $G = 16$ rollouts per prompt with three stabilizers: a binary trust-region mask on the trainer–inference probability gap, a quadratic mismatch-KL penalty on the importance ratio, and a small stop-gradient anchor toward the frozen SFT model. Without the last two, training quickly lost stability and policy entropy collapsed. We also drop groups with zero reward variance and resample them, following DAPO (Yu et al., 2025). RL runs for 4,500 steps on 8 B200 GPUs; the full objective is given in Appendix B.3.

5 EXPERIMENTS AND ANALYSIS

5.1 SETUP

Models We compare our VerusLM-4B-Codex with several baselines, we benchmark against Qwen family of models (Qwen Team, 2025), Kimi-K2.5 (Team et al., 2026), GPT-OSS (OpenAI et al., 2025), MiniMax-M2.5 (MiniMax, 2026), DeepSeek-V3.2 (Guo et al., 2025). We also baseline against other open source models that are trained on verus data like PSV (Wilf et al., 2025) and VeruSyn (Di et al., 2026). For evaluation hyperparameters, please refer to Appendix F.3.

Benchmarks We evaluate VerusLM-4B-Codex on four benchmarks targeting different stages of end-to-end verified code generation. **VeruSage-Bench** (Yang et al., 2026) is a system-verification suite of 849 proof tasks extracted from eight open-source Verus-verified Rust systems such as key-value stores and operating systems, It uses*hands-on* agentic harness that surfaces customized error feedback, and to isolate model capability, we additionally evaluate under a stripped-down *plain harness* that only supports a ReAct-style loop over raw Verus errors. **SpecBench** is the specification generation benchmark from Section 3 where Stage 1 is a Verus verification check, Stage 2 tries to prove equivalence to ground-truth, and Stage 3 runs an LLM judge to check coherence and semantic alignment. **VerifyThisBench** (Deng et al., 2025) consists of 41 natural-language competition problems for end-to-end generation, where the model must jointly produce code, specification, and proof. We report the Verus pass rate, and because this setting admits several reward-hacking failure modes (Appendix F.2), we additionally enforce that the response contains a valid `verus!` macro and use Gemini-3-Flash-Preview to judge whether the model's output is coherent and semantically

Table 2: Results on all 4 benchmarks for different models. We report successful proof attempt % in both the Hands-On harness and the Plain Harness for VeruSage-Bench, Stage 1 pass %, Stage 2 pass % and the total Stage 3 score for SpecBench, Verified% and Coherent% for VerifyThisBench and VerifyThisBench-XS. Inst./Think correspond to Instruct/Thinking variants. We highlight our base checkpoint in green and our model (**VerusLM-4B-Codex**) in purple.

	VeruSage-Bench (849)		SpecBench (97)			VTB-XS (99)		VerifyThisBench (41)	
Model	Hands-On%	Plain-Harness%	S1%	S2%	S3 Score	Verified%	Coherent%	Verified%	Coherent%
Qwen3-Coder-30B-A3B	12.13	5.65	3.09	1.03	1.88	13.13	11.11	7.32	0.00
Qwen3-Coder-480B-A35B	27.92	25.91	16.49	6.19	9.21	26.26	23.23	0.00	0.00
Qwen3-4B (Inst.)	3.30	1.30	2.06	1.03	0.80	3.03	3.03	0.00	0.00
Qwen3-30B-A3B (Inst.)	8.24	3.89	2.06	1.03	1.00	9.09	9.09	0.00	0.00
Qwen3-235B-A22B (Inst.)	23.91	19.67	4.12	1.03	3.00	12.12	8.08	0.00	0.00
Qwen3-4B (Think)	3.06	1.06	1.03	1.03	—	9.09	8.08	0.00	0.00
Qwen3-30B-A3B (Think)	11.78	3.65	3.09	2.06	1.00	16.16	15.15	0.00	0.00
Qwen3-235B-A22B (Think)	23.44	19.55	6.19	4.12	2.00	27.27	25.25	0.00	0.00
Kimi-K2.5	**41.58**	**41.22**	**19.59**	**9.28**	**9.68**	43.43	**42.42**	2.44	**2.44**
GPT-OSS (120B)	24.85	6.24	10.31	3.09	6.75	37.37	25.25	0.00	0.00
MiniMax-M2.5	20.14	11.31	8.25	3.09	4.75	23.23	20.20	0.00	0.00
Qwen-VeruSyn (32B-Inst.)	14.96	13.55	2.06	2.06	—	14.14	11.11	7.32	0.00
PSV-Verus (3B-Inst.)	0.24	0.12	0.00	0.00	—	8.08	2.02	0.00	0.00
VerusLM-4B-Codex-SFT	1.30	9.07	0.00	0.00	—	6.06	2.02	65.85	**2.44**
VerusLM-4B-Codex	7.66	7.42	4.12	0.00	3.00	**46.46**	5.05	**75.61**	0.00

aligned with the natural language intent or not. **VerifyThisBench-XS** (Deng et al., 2025) is a relaxed 99-task variant in which the model generates one of {code, specification, proof} given the other two, better isolating each sub-skill; we compare zero-shot and multi-turn settings in Appendix G.1.

5.2 Evaluation

5.2.1 RQ1: How well do different models perform at verified code generation?

Table 2 reports results across all four benchmarks. We compare VerusLM-4B-Codex against 12 open-source general-purpose models (Qwen3-Coder, Qwen3 Instruct/Think variants, Kimi-K2.5, GPT-OSS, MiniMax-M2.5, and DeepSeek-V3.2) and two prior verification-specific approaches, VeruSyn (Di et al., 2026) and PSV-Verus (Wilf et al., 2025).

Performance relative to model size. Figure 3 plots each model's average verification pass rate across the four benchmarks against its parameter count. Among baseline models, verification success grows roughly log-linearly with scale, gaining about 7.7 points per $10\times$ increase in parameters (dashed line). VerusLM-4B-Codex lies far above this trend: at 4B parameters it reaches an average of 33.43%, compared with 1.34% predicted by the trend at its size and 3.04% for the strongest baseline of equal size (Qwen3-4B Thinking). It improves on its own base checkpoint by $18\times$ (1.85% $\rightarrow$ 33.43%) and exceeds Kimi-K2.5 (26.71%), a model roughly $250\times$ larger. The gain is not due to SFT alone: RL raises the average from 19.27% to 33.43%, consistent with the multi-turn training of Section B.2 teaching the model to act on verifier feedback (Appendix G.1).

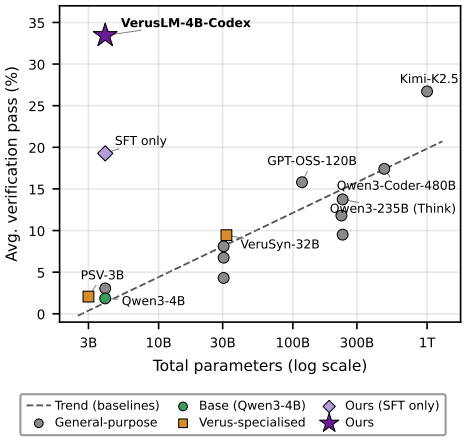

Figure 3: Average verification pass rate across the four benchmarks vs. model size. Dashed line: log-linear fit to baseline models.

VeruSage-Bench. Kimi-K2.5 is the strongest general-purpose model (41.58% Hands-On, 41.22% Plain-Harness), followed by DeepSeek-V3.2 (37.57 / 32.27) and Qwen3-Coder-480B (27.92 / 25.91).

Plain-harness scores are uniformly lower than hands-on scores, confirming that customized error feedback contributes a non-trivial fraction of agentic-harness performance.

SpecBench. Several general-purpose models obtain moderate Stage 1 pass rates but degrade sharply at Stage 2 (e.g. Kimi-K2.5: 19.59% $\rightarrow$ 9.28%), indicating that many generated specifications compile but are semantically weak. This validates Stage 2 and Stage 3 as the necessary training signal. The prior verification-specific approaches VeruSyn and PSV-Verus score near zero across all stages. VerusLM-4B-Codex attains a Stage 1 pass rate competitive with frontier models while remaining more robust under stricter checks.

VerifyThisBench. Most models fail outright at end-to-end generation (Verified% $\leq$ 7.32, Coherent% $\leq$ 2.44). VerusLM-4B-Codex reaches **75.61% Verified**, roughly $10\times$ the next best baseline, but its Coherent% collapses to 0%, indicating successful reward hacking enabled by the benchmark's weak verification signal. We dissect these failure modes in Appendix F.2. In **VerifyThisBench-XS.** With two of three artifacts provided, the feedback channel is strong enough that general-purpose models iterate effectively on Verus errors: Kimi-K2.5 reaches 43.43% Verified. VerusLM-4B-Codex leads on Verified% (46.46%) but again at low coherency (5.05%), showing that the reward-hacking pattern persists whenever the verification signal dominates supervision.

5.2.2 RQ2: How important is Stage 3 for SpecBench?

In SpecBench, a model-generated specification that passes Stage 1 (Verus verification + cheat checks) but fails Stage 2 (equivalence proof) goes to Stage 3 where an LLM judge determines what portion of the relevant specifications it was able to fill in coherently and correctly. Through a qualitative analysis, we identify three distinct categories of Stage 2 failure and show why they create a need for Stage 3. We illustrate each with a representative example.

Case 1: The specification is semantically equivalent but the equivalence proof is too hard. On task `temporal_logic__always_implies_to_leads_to` from the Anvil (Sun et al., 2024) repository, five models (inlcuding GPT-5.3, Kimi-K2.5) produce specifications with *zero* textual difference from the ground truth. Stage 2 still fails because the equivalence harness emits one proof obligation per specification function, and these obligations are interdependent: since `leads_to` calls `always`, proving `leads_to == leads_to_model` first requires `always == always_model` and `eventually == eventually_model`. Verus does not auto-discharge these equalities for higher-order specification functions, and the model cannot construct the chain in three attempts. This accounts for the majority of Stage 2 failures among strong models, and the LLM judge consistently marks them fully correct in Stage 3.

Case 2: The specification is strictly stronger than the ground truth. On `Finding_Two_Duplets_in_an_Array` from the VerifyThisBench (2011) challenge, 7/7 models that pass Stage 1 independently add the same postcondition that the ground truth omits:

```
fn max(a: &[i32]) -> (max_idx: usize) // Ground truth:
ensures forall|i: int| 0 <= i < a.len() ==> a[max_idx as int] >= a[i],
fn max(a: &[i32]) -> (max_idx: usize) // Model (GPT-5.5, and 6 others):
ensures max_idx < a.len(), forall|i: int| 0 <= i < a.len() ==> a[i] <= a[max_idx as int],
```

The ground truth does not guarantee `max_idx < a.len()`, so callers cannot derive it from the specification alone. The model's specification is strictly more useful as it provides a bound that any caller indexing into `a` at `max_idx` would need. The equivalence proof fails in the GT$\Rightarrow$Model direction because the missing the model's specification is stronger.

Case 3: The specification is strictly weaker than the ground truth. On `tentatively_append_wrapping` from the `verified-storage` repository (Microsoft, 2026), GPT-5.5 replaces a 15-line postcondition specifying crash-consistency properties of a circular persistent-memory log with `ensures true`. The lemma verifies trivially but is useless: callers learn nothing about crash safety. Stage 3 catches this error and deducts points for the incorrect specification.

```
fn lemma_tentatively_append_wrapping(...) // Ground truth
ensures //...15 lines: no outstanding writes, offsets unreachable during recovery, state
↪  consistency
fn lemma_tentatively_append_wrapping(...) ensures true, // Model (GPT-5.5)
```

Implications. Taken together, these cases show that while an equivalence proof provides an irrefutably strong guarantee on the correctness of the generated specification, the absence of a proof does not imply incorrectness. Since the proof can fail for varied reasons, it is important and beneficial to have a flexible judge decide whether the specification is coherent and semantically aligns with the natural language intent. This is the purpose that Stage 3 serves for SpecBench.

5.2.3 RQ3: How does specification complexity affect LLM Verification capabilities?

We observe that most LLMs struggle to perform well on SpecBench, this makes it harder to understand the ceiling performance of the automated systems. To evaluate it's robustness we also evaluate closed sources frontier models on SpecBench. Table 3 discuss the performance of 3 frontier models, where we can observe GPT-5.5 outperfroms all other SLMs significantly and yet is only able to solve 51% of the benchmark in the Stage 1, and can only solve 21% of the problems in Stage 2.

Table 3: Spec-Gen (97) results for frontier GPT models. S1 / S2 are Stage 1 / Stage 2 verification rates; **S3 Score** is the Stage 3 total coherence score. From GPT-5.5's results, we can see that SpecBench reveals a substantial gap between open and closed frontier model performance.

Model	S1%	S2%	S3 Score
GPT-5.3-codex	24.74	8.25	15.13
GPT-5.4-mini	13.40	5.15	7.63
GPT-5.5	**51.55**	**23.71**	**22.41**

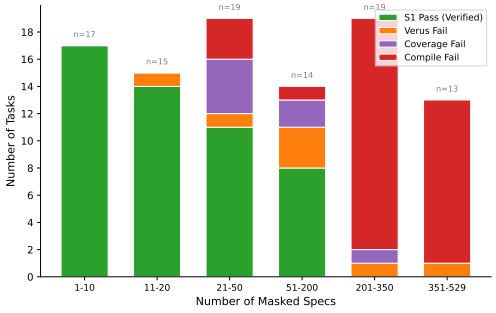

Figure 4: Effect of specification complexity on verification performance across tasks grouped by the number of masked specifications.

We define specification complexity as the number of specifications masked within a code file. Figure 4 reveals a sharp phase transition: GPT-5.5 achieves near-perfect verification success below 20 masked specifications, degrades substantially at intermediate complexity, and fails entirely beyond 200. Failure modes also shift with complexity. Small tasks are dominated by verifier failures, indicating semantically incorrect but syntactically valid programs. Intermediate tasks introduce more coverage and compilation failures, reflecting challenges in jointly maintaining specifications, proofs, and code. At the highest complexity, failures are almost entirely compilation errors, suggesting syntactic breakdown across interdependent verification artifacts. Overall, increasing complexity shifts failures from semantic reasoning to syntactic incoherence, revealing a solvability frontier for current LLM-based verification systems.

6 Conclusion and Limitations

We present a verifier-grounded training pipeline for end-to-end verified Rust code generation and train VerusLM-4B-Codex , demonstrating promising results across proof, specification, and code generation tasks. We introduce SpecBench, a benchmark highlighting challenges in specification generation for complex Rust repositories and gaps between open and closed frontier models. Our evaluation relies on reference specifications, leaving robust assessment of semantic equivalence and alignment with developer intent an important direction for future work.

REFERENCES

Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. Dafnypro: Llm-assisted automated verification for dafny programs, 2026. URL https://arxiv.org/abs/2601.05385.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, et al. Automated proof generation for Rust code via self-evolution. 2025.

Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. Atmosphere: Towards practical verified kernels in rust. In *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, KISV '23, pp. 9–17, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400704116. doi: 10.1145/3625275.3625401. URL https://doi.org/10.1145/3625275.3625401.

Zaoyu Chen, Jianbo Dai, Boyu Zhu, Jingdong Wang, Huiming Wang, Xin Xu, Haoyang Yuan, Zhijiang Guo, and Xiao-Ming Wu. Codespecbench: Benchmarking llms for executable behavioral specification generation, 2026. URL https://arxiv.org/abs/2604.12268.

Domenico Cotroneo, Cristina Improta, and Pietro Liguori. Human-written vs. ai-generated code: A large-scale study of defects, vulnerabilities, and complexity, 2025. URL https://arxiv.org/abs/2508.21634.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). 2015.

Xun Deng, Sicheng Zhong, Barış Bayazıt, Andreas Veneris, Fan Long, and Xujie Si. Verifythisbench: Generating code, specifications, and proofs all at once, 2025. URL https://arxiv.org/abs/2505.19271.

Nongyu Di, Tianyu Chen, Shan Lu, Shuai Lu, Yeyun Gong, Peng Cheng, Jacob R. Lorch, Yuan Yao, and Xiaoxing Ma. Reducing the costs of proof synthesis on rust systems by scaling up a seed training set, 2026. URL https://arxiv.org/abs/2602.04910.

DPPO Authors. Decoupled proximal policy optimization with binary total-variation trust region. Technical report, 2025.

Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 2024.

Fabian Gloeckle, Mantas Bakšys, Darius Feher, Kunhao Zheng, Amaury Hayat, Sean B. Holden, Gabriel Synnaeve, and Peter O'Hearn. Wybecoder: Verified imperative code generation. *Preprint*, 2026.

Google DeepMind. AI achieves silver-medal standard solving international mathematical olympiad problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. 2022.

Rijul Jain, Shraddha Barke, Gabriel Ebner, Md Rakib Hossain Misu, Shan Lu, and Sarah Fakhoury. What's in a proof? analyzing expert proof-writing processes in f* and verus, 2025. URL https://arxiv.org/abs/2508.02733.

Kimi Team. Kimi K2.5: Scaling reinforcement learning with mismatch-KL regularization. Technical report, 2025.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating systems principles (SOSP)*, 2009.

Shuvendu K. Lahiri. Evaluating llm-driven user-intent formalization for verification-aware languages, 2024. URL https://repositum.tuwien.at/handle/20.500.12708/200786.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 2023.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. 2010.

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert-a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*, 2016.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Xian Zhang, Fan Yang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency. 2024.

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-code-prover: Hierarchical proof search for open state-of-the-art code verification, 2026. URL https://arxiv.org/abs/2603.19329.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models, 2025. URL https://arxiv.org/abs/2401.08807.

Mathlib community. The Lean mathematical library. 2020.

Microsoft. Verified storage. https://github.com/microsoft/verified-storage, 2026. GitHub repository, accessed 2026-05-07.

MiniMax. MiniMax M2.5: Built for Real-World Productivity. https://www.minimax.io/news/minimax-m25, February 2026. Accessed: 2026-05-07.

Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM on Programming Languages*, 2025.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. gpt-oss-120b gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Gabriel Poesia, Chloe Loughridge, and Nada Amin. dafny-annotator: Ai-assisted verification of dafny programs, 2024. URL https://arxiv.org/abs/2411.15143.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

Prime Intellect. prime-rl: Decentralized RL training at scale. https://github.com/PrimeIntellect-ai/prime-rl, 2024.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Yu Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G Patil, Lingjiao Chen, Wei-Lin Chiang, et al. Specifications: The missing link to making the development of llm systems an engineering discipline. *arXiv preprint arXiv:2412.05299*, 2024.

Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 649–666, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/sun-xudong.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, Yanru Chen, Yanxu Chen, Yicun Chen, Yimin Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Ziwei Chen, Dazhi Cheng, Yean Cheng, Minghan Chu, Jialei Cui, Jiaqi Deng, Muxi Diao, Hao Ding, Mengfan Dong, Mengnan Dong, Yuxin Dong, Yuhao Dong, Angang Du, Chenzhuang Du, Dikang Du, Lingxiao Du, Yulun Du, Yu Fan, Shengjun Fang, Qiulin Feng, Yichen Feng, Garimugai Fu, Kelin Fu, Hongcheng Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Chengyang Gong, Xiaochen Gong, Zhuoma Gongque, Qizheng Gu, Xinran Gu, Yicheng Gu, Longyu Guan, Shuhao Guan, Yuanying Guo, Xiaoru Hao, Dailan He, Tianhong He, Weiran He, Wenyang He, Yibo He, Yunjia He, Chao Hong, Hao Hu, Jiaxi Hu, Yangyang Hu, Zhenxing Hu, Ke Huang, Ruiyuan Huang, Weixiao Huang, Zhiqi Huang, Chaobo Jia, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yu Jing, Guokun Lai, Aidi Li, C. Li, Cheng Li, Fang Li, Guanghe Li, Guanyu Li, Haitao Li, Haoyang Li, Jia Li, Jingwei Li, Junxiong Li, Lincan Li, Mo Li, Weihong Li, Wentao Li, Xinhang Li, Xinhao Li, Yang Li, Yanhao Li, Yiwei Li, Yuxiao Li, Zhaowei Li, Zhaoxi Li, Zheming Li, Weilong Liao, Jiawei Lin, Xiaohan Lin, Yibo Lin, Zhishan Lin, Zichao Lin, Cheng Liu, Chenyu Liu, Hongzhang Liu, Liang Liu, Shaowei Liu, Shudong Liu, Shuran Liu, Tianwei Liu, Tianyu Liu, Weizhou Liu, Xiangyan Liu, Yangyang Liu, Yanming Liu, Yibo Liu, Yuanxin Liu, Zhengying Liu, Zhongnuo Liu, Enzhe Lu, Haoyu Lu, Zhiyuan Lu, G. Luo, Junyu Luo, Tongxu Luo, Yashuo Luo, Long Ma, Shaoguang Mao, Yuan Mei, Xin Men, Fanqing Meng, Zhiyong Meng, Yibo Miao, Minqing Ni, Kun Ouyang, Siyuan Pan, Bo Pang, Yuchao Qian, Ruoyu Qin, Zeyu Qin, Jiezhong Qiu, Bowen Qu, Zeyu Shang, Youbo Shao, Tianxiao Shen, Zhennan Shen, Juanfeng Shi, Lidong Shi, Shengyuan Shi, Feifan Song, Pengwei Song, Tianhui Song, Xiaoxi Song, Hongjin Su, Jianlin Su, Zhaochen Su, Lin Sui, Jinsong Sun, Junyao Sun, Tongyu Sun, Flood Sung, Yunpeng Tai, Chuning Tang, Heyi Tang, Xiaojuan Tang, Zhengyang Tang, Jiawen Tao, Shiyuan Teng, Chaoran Tian, Pengfei Tian, Bowen Wang, Chensi Wang, Chuang Wang, Congcong Wang, Dingkun Wang, Dinglu Wang, Dongliang Wang, Feng Wang, Hailong Wang, Haiming Wang, Hao Wang, Hengzhi Wang, Huaqing Wang, Hui Wang, Jiahao Wang, Jinhong Wang, Jiuzheng Wang, Kaixin Wang, Linian Wang, Qibin Wang, Shengjie Wang, Shuyi Wang, Si Wang, Wei Wang, Xiaochen Wang, Xinyuan Wang, Yao Wang, Yejie Wang, Yipu Wang, Yiqin Wang, Yucheng Wang, Yuzhi Wang, Zhaoji Wang, Zhaowei Wang, Zhengtao Wang, Zhexu Wang, Zifan Wang, Zihan Wang, Zizhe Wang, Chu Wei, Ming Wei, Chuan Wen, Zichen Wen, Chengjie Wu, Haoning Wu, Junyan Wu, Rucong Wu, Wenhao Wu, Yuefeng Wu, Yuhao Wu, Yuxin Wu, Zijian Wu, Chenjun Xiao, Jin Xie, Xiaotong Xie, Yuchong Xie, Bowei Xing, Boyu Xu, Jianfan Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu,

Weixin Xu, Xinbo Xu, Xinran Xu, Yangchuan Xu, Yichang Xu, Yuemeng Xu, Zelai Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Guangyao Yang, Hao Yang, Junwei Yang, Kai Yang, Ningyuan Yang, Xiaofei Yang, Xinlong Yang, Xinyu Yang, Ying Yang, Yi Yang, Yi Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Dan Ye, Haoran Ye, Wenjie Ye, Zhuorui Ye, Peng Yebo, Bohong Yin, Chengzhen Yu, Longhui Yu, Tao Yu, Tianxiang Yu, Enming Yuan, Mengjie Yuan, Xiaokun Yuan, Yang Yue, Weihao Zeng, Dunyuan Zha, Haobing Zhan, Dehao Zhang, Hao Zhang, Jin Zhang, Puqi Zhang, Qiao Zhang, Rui Zhang, Xiaobin Zhang, Xiaoyun Zhang, Y. Zhang, Yadong Zhang, Yangkun Zhang, Yichi Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yushun Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Chenguang Zhao, Feifan Zhao, Jinxiang Zhao, Shuai Zhao, Xiangyu Zhao, Xuanle Zhao, Yikai Zhao, Zijia Zhao, Huabin Zheng, Ruihan Zheng, Shaojie Zheng, Tengyang Zheng, Junfeng Zhong, Longguang Zhong, Weiming Zhong, M. Zhou, Runjie Zhou, Xinyu Zhou, Zaida Zhou, Jinguo Zhu, Liya Zhu, Xinhao Zhu, Yuxuan Zhu, Zhen Zhu, Jingze Zhuang, Weiyu Zhuang, Ying Zou, and Xinxing Zu. Kimi k2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally verified code generation, 2025. URL https://arxiv.org/abs/2505.13938.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouedec. TRL: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

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, 2023. URL https://arxiv.org/abs/2212.10560.

Ke Weng, Lun Du, Sirui Li, Wangyue Lu, Haozhe Sun, Hengyu Liu, and Tiancheng Zhang. Autoformalization in the era of large language models: A survey. *arXiv preprint arXiv:2505.23486*, 2025.

Alex Wilf, Pranjal Aggarwal, Bryan Parno, Daniel Fried, Louis-Philippe Morency, Paul Pu Liang, and Sean Welleck. Propose, solve, verify: Self-play through formal verification, 2025. URL https://arxiv.org/abs/2512.18160.

Yuhuai Wu, Albert Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. 2022.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. Autoverus: Automated proof generation for rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2): 3454–3482, 2025.

Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R. Lorch, and Shan Lu. Verusage: A study of agent-based verification for rust systems, 2026. URL https://arxiv.org/abs/2512.18436.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. *arXiv preprint arXiv:2505.23135*, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

A Static Analysis and the Anti-Cheat Cascade

This appendix describes the structural-analysis foundation that every gate, masking operation, and equivalence check in the paper relies on: the Lynette static analyzer (§A.1) and the five anti-cheat gates that compose into the cascade used by both SpecBench evaluation and RL reward shaping (§B.4).

A.1 LYNETTE: STATIC ANALYZER FOR VERUS

LYNETTE (Yang et al., 2026) parses the Verus abstract syntax tree and assigns every syntactic element to a typed *segment*, enabling programmatic extraction, masking, and structural comparison of specification, proof, and executable regions.

Segment taxonomy. Every segment carries a *kind* drawn from one of three categories:

Category	Function kinds	Clause kinds
Specification	`spec_fn`, `spec_checked_fn`	`requires`, `ensures`, `recommends`, `decreases`
Proof	`proof_fn`, `proof_axiom_fn`	`invariant`, `invariant_ensures`, `invariant_except_break`, `loop_ensures`, `assert`, `assert_forall`, `assume`, `proof_block`, `decreases`
Executable	`exec_fn`, `fn`	—

Masking and comparison. Masking a set of segment kinds removes the bodies of all function-kind segments in that set and drops all clause-kind segments belonging to surviving functions; this produces the five training modes (PROOF-MODE, SPEC-MODE, EXEC-MODE, PAIR-MODE, END2END-MODE). When deciding whether a function's `requires`/`ensures` clauses are masking targets, functions explicitly tagged as external (`#[verifier::external_body]`, `#[verifier::external_fn_specification]`, `#[verifier::external]`) are excluded, since they carry no genuine specification; a function whose body merely contains `admit()` or `assume(false)` still has a meaningful specification and remains eligible. The `lynette compare` command takes two Verus files and a mode flag (`--proof-mode` or `--spec-mode`, optionally with `--allow-helpers`) and verifies that all segments outside the designated mask region are byte-equivalent between the two files; this is the implementation of the C3 structural-integrity gate.

Extensions for VerusLM-4B-Codex . We extend the original LYNETTE with three capabilities used throughout this paper: (i) mode-specific masking for all five training modes (Section 4.1), (ii) segment-coverage comparison for the C4 coverage gate, and (iii) spec-function body extraction used by the equivalence harness (§D.3).

B TRAINING PIPELINE DETAILS

B.1 DATA PREPARATION

Figure 2 gives an overview. Following Self-Instruct (Wang et al., 2023), we prompt Qwen3-30B-Thinking-2507 (Qwen Team, 2025) to write, for each verified program, a self-contained problem statement NL that describes the algorithmic task and all pre-/post-conditions without referencing the proof body. The five masking modes are: 1. PROOF-MODE (NL, Exec, Spec) → Proof 2. SPEC-MODE (NL, Exec, Proof) → Spec 3. PAIR-MODE (NL, Exec) → Spec + Proof 4. EXEC-MODE (NL, Spec, Proof) → Exec 5. END2END-MODE (NL) → Exec + Spec + Proof. Decomposition into structural units uses LYNETTE (Yang et al., 2026), which we extend with mode-specific masking, segment-coverage comparison, and spec-function body extraction (Appendix A). RL instances are drawn with the two-tier diversity sampler of Appendix C, and a sequential rollout buffer preserves the round-robin mode interleaving across epochs (Appendix E). SFT and RL hyperparameters are in Table **??**.

B.2 MULTI-PHASE ROLLOUTS

In SPEC-MODE and END2END-MODE, the multi-turn mechanism covers both plain retries and phase transitions. After a full Verus pass (Phase A), the environment constructs the equivalence harness (Phase B). If some equivalence lemmas remain undischarged, the partially verified harness itself

becomes the next-turn task (Phase C). If harness generation must drop a model-defined helper that lives in a scope the harness cannot mirror (e.g. a trait impl), the environment surfaces the helper to the model and returns the rollout to Phase A for a spec rewrite. Per-phase reward values and the harness construction are in Appendix D.

B.3 Full Optimization Objective

The trainer maximizes

$$\mathcal{J}(\theta) = \mathbb{E}_{(n,\langle M\rangle)\sim\mathcal{D},\,\{o_i\}_{i=1}^{G}\sim\pi_{\text{infer}}(\cdot|n,\langle M\rangle)}\left[\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\left(m_{i,t}\,\hat{A}_{i,t}\,\rho_{i,t} - \kappa\,(\log\rho_{i,t})^2\right)\right], \tag{1}$$

where $\rho_{i,t} = \pi_\theta(o_{i,t} \mid s_{i,t})/\pi_{\text{infer}}(o_{i,t} \mid s_{i,t})$ is the per-token trainer–inference ratio, $m_{i,t}$ is the binary trust-region mask, $\hat{A}_{i,t}$ is the teacher-anchored advantage built on the group baseline $\hat{A}_i = R_i - \frac{1}{G}\sum_{j=1}^{G} R_j$, and the normalizer sums over all generated tokens in the group. Dynamic sampling drops groups with $\hat{A}_i = 0$ for every rollout and resamples (up to $4\times$ over-generation across 8 attempts per step) until the batch contains only groups with non-trivial reward variance. Definitions of $m_{i,t}$, the mismatch-KL coefficient κ, the teacher anchor, and per-component motivation are in Appendix E.

B.4 Anti-Cheat Gate Definitions (C1–C5)

The same five gates are reused as the gate cascade in both SpecBench evaluation (Section 3.2) and RL reward shaping (Section **??**); the only difference between their two uses is the asymmetric reward/penalty values applied during RL (§D). The main text presents the cascade compactly as two checks: its C1 (cheat patterns) corresponds to gates C1–C2 below, its C2 (scope and coverage) to gates C3–C4, and the Stage-2 equivalence check of Section 3.2 is gate C5.

Let $\langle M\rangle$ denote the masked input program presented to the model and $\langle O\rangle$ the candidate program reconstructed from the model's output.

- **(C1) Cheat-pattern check.** For a fixed forbidden set $\mathcal{P}$, the candidate may not introduce any new occurrence of $\pi \in \mathcal{P}$:

  $$\text{count}(\langle O\rangle,\,\pi) \;\le\; \text{count}(\langle M\rangle,\,\pi) \quad \forall \pi \in \mathcal{P}.$$

  The full forbidden set is $\mathcal{P}$ = {`unimplemented!()`, `admit()`, `assume(false)`, `#[verifier::external_body]`, `#[verifier::external_fn_specification]`, `#[verifier::external]`}. Counting occurrences (rather than checking presence) makes the gate robust to reformatting and renaming.
- **(C2) Axiom check.** The number of `proof_axiom_fn` declarations may not increase: $\text{count}_{\texttt{proof_axiom_fn}}(\langle O\rangle) \le \text{count}_{\texttt{proof_axiom_fn}}(\langle M\rangle)$. This is the `spec_fn`-level analogue of C1 against axiom injection.
- **(C3) Structural integrity.** Lynette's `compare` command enforces byte-level equality outside the masked region, ruling out edits to sibling lemmas or preconditions that would trivialize the target obligation. The exact configuration is mode-dependent: Proof-Mode uses `--proof-mode`; Spec-Mode uses `--spec-mode`; Phase C of the equivalence harness (§D.3) uses `--spec-mode --allow-helpers` to permit new helper proof functions while still rejecting edits to spec bodies or equivalence-lemma clauses.
- **(C4) Coverage.** Every (*kind*, *name*) entry in the masked input segment list must reappear in the candidate, preventing silent drops or renames of required `spec_fn`, `proof_fn`, `exec_fn`, `requires`, or `ensures` entries. End2End-Mode additionally requires that the candidate expose a non-empty `verus!{}` top-level macro, ruling out the trivial cheat of emitting a Verus-free Rust file.
- **(C5) Equivalence harness** (Spec-Mode and End2End-Mode only). After the candidate passes the verifier, the harness pairs each candidate spec function with its ground-truth counterpart and requires SMT-discharged semantic equivalence (§D.3). This rules out vacuous

but verification-sufficient specifications such as `ensures(true)` or strict weakenings of the intended postcondition.

The gates are ordered cheap-to-expensive (parsing → pattern → structural → verification → equivalence), and a failure on any earlier gate short-circuits the cascade. They are also *composable but not redundant*: a candidate that inserts `assume` outside the masked region fails C1 but not C3; one that rewrites a sibling lemma's postcondition without touching any cheat pattern fails C3 but not C1. Reporting per-gate failure rates therefore localizes reward hacking to a specific mechanism; §F.2 illustrates the failure modes in prior benchmarks that motivated this design.

C RL DATA SAMPLING

The RL pool drawn in Section **??** (14,254 instances across PROOF-MODE, SPEC-MODE, and END2END-MODE in a 1:2:2 ratio) is constructed by a deterministic two-tier sampler that targets diversity along program length, mask composition, and Verus feature usage. Uniform sampling would be dominated by short programs that exercise only the most common Verus constructs.

Tier 1: rare-construct guarantee. A curated set of high-signal but uncommon Verus keywords – `nonlinear_arith`, `bit_vector`, `broadcast`, `reveal`, `tracked`, `Map`, `Set`, `proof_fn` – is identified upfront. Every candidate sample whose masked region contains any of these keywords is included unconditionally, ensuring rare constructs survive downstream filtering.

Tier 2: stratified fill. The remaining quota is allocated over strata defined by (lines-of-code bucket) $\times$ (mask combination). LOC allocations are levelled to a flat marginal distribution; within each bucket, allocations across mask combinations are sub-linear in stratum size, up-weighting rare combinations relative to common ones.

Within-cell quality bias. Each sample is scored by the IDF-weighted sum of Verus keywords in its masked region, so rarer keywords contribute more. Within every stratum, half the slots are filled deterministically with the highest-scoring samples and the other half are drawn uniformly, combining a quality bias with enough stochasticity to avoid overfitting to a few high-scoring outliers. A final rounding pass fills any unfilled buckets and trims over-allocations by removing the lowest-scoring Tier 2 samples first.

D MODE-SPECIFIC REWARD PIPELINES

This appendix gives the full reward shape that Section **??** summarizes. We describe the single-phase PROOF-MODE pipeline first (§D.1), then the three-phase SPEC-MODE and END2END-MODE pipelines built on top of it (§D.2), and finally the equivalence-harness construction that the latter two depend on (§D.3). All gates referenced below are defined in §B.4.

D.1 PROOF MODE (SINGLE PHASE)

The model receives (n, r, s) and emits SEARCH/REPLACE patches that fill the masked proof region. The candidate is driven through the gate cascade in increasing order of cost: patch parse → `verus --no-verify` compile → C1+C2 → C3 → Verus. Per-gate reward values are listed in Table 4; the asymmetric C1+C2 penalty (-2.0) is calibrated so that the maximum honest pre-verification score ($+1.5$) strictly exceeds the maximum score obtainable by any pattern of cheating (-0.5).

Conditional on passing every cheat gate, the verifier yields the dense score

$$\mu = \frac{V}{V+E}, \tag{2}$$

where V and E are the discharged verification conditions and errors. A fully verified proof ($\mu = 1$) earns an additional terminal bonus and ends the rollout; a partial pass returns Verus stderr to the policy as next-turn feedback.

Gate	Pass reward	Fail penalty
Patch credit	$+0.5 \times {}^{\lvert\text{applied}\rvert}/_{\lvert\text{total}\rvert}$	0
Compilation	$+0.5$	-0.5
C1+C2 (cheat patterns, axiom fns)	$+0.5$	-2.0
C3 (structural integrity)	$+0.5$	-1.0
Verus dense score	$\mu + \mathbf{1}[\mu{=}1]$	0

Table 4: Per-gate reward values for PROOF-MODE. Asymmetric penalties make cheating dominated by honest partial credit.

D.2 SPEC AND END-TO-END MODES (THREE PHASES)

SPEC-MODE and END2END-MODE share a three-phase structure that distributes across the (up to three) turns of a rollout and reuses the proof-mode pipeline as Phase A.

Phase A – internal consistency. Mirrors proof mode but with the masked region targeting specifications. The cheat cascade is re-targeted: C3 is configured for spec edits (`lynette compare --spec-mode`), and C4 is added with reward $+0.1$ on pass and $-\min(0.25 \cdot n_{\text{missing}}, 2.0)$ on fail. A partial Verus pass is fed back as next-turn feedback exactly as in proof mode, but a *full* Verus pass does not terminate the rollout: the model has only proved its spec is internally consistent, not that it captures the intended postcondition. The environment therefore transitions to Phase B.

Phase B – automatic equivalence. The environment synthesizes the equivalence harness (§D.3) and attempts a no-model verification pass. Full equivalence earns $+1.0$ (partial credit) $+1.0$ (full bonus) and terminates the rollout. Partial equivalence transitions to Phase C; harness-generation failure carries no penalty and retries Phase A. A subtle case arises when harness generation must drop a model-defined helper because it lives in a scope (e.g. a trait impl) the harness cannot mirror: the environment surfaces the offending helper to the model and routes the rollout back to Phase A for a spec rewrite rather than continuing fruitlessly in Phase C.

Phase C – model-supplied equivalence proof. The partially-verified harness is presented to the model, which fills proof-body regions via SEARCH/REPLACE patches. Gates are: patch credit, compilation, cheat check (no spec-function tampering, `lynette compare --spec-mode --allow-helpers`), and Verus. A full pass earns $+1.0$ bonus and terminates.

End-to-end-mode differences. END2END-MODE differs from SPEC-MODE only in Phase A. First, alongside the natural-language description we provide an empty skeleton of the program: ground-truth function signatures and `requires`/`ensures` clause headers with all bodies blanked. The skeleton fixes the structural shape so that the equivalence machinery in Phase B can pair the model's spec functions with their ground-truth counterparts by name; without it, the model would be free to invent unrelated signatures that no harness could check. The action space is correspondingly the full Verus program: the model emits a single fenced ```` ```rust ... ``` ```` block containing the complete file, rather than SEARCH/REPLACE patches. Second, since the skeleton is intentionally non-verifying, C3 is replaced by C4 anchored on the ground-truth segment list. Phases B and C are unchanged.

D.3 EQUIVALENCE HARNESS CONSTRUCTION

The equivalence harness is the machinery behind Stage 2 of SpecBench (Section 3.2) and Phases B/C of the RL pipeline above; it pins reward to semantic correctness rather than verifier exploitation.

For each spec function f shared by the ground-truth and candidate programs, LYNETTE extracts both bodies and the harness emits an SMT obligation that pairs them:

(1) Synthesize a companion `f_model` carrying the candidate body alongside the original.

(2) Lift `requires` and `ensures` clauses into standalone spec functions (`f_requires`, `f_ensures`, `f_model_requires`, `f_model_ensures`) so that clause-level equivalence can be checked in isolation.

(3) Emit, for each pair, a Verus proof obligation:

```
proof fn equivalence_lemma_f<T>(args)
    ensures f(args) =~= f_model(args)
{ //%% PROOF_BODY_START ... //%% PROOF_BODY_END }
```

Special cases.

- *Trait-impl spec functions.* A `_model` companion cannot be added directly inside a trait impl; we instead introduce a subtrait that hosts both the original and the candidate definitions.
- *Self-referential spec functions.* A recursive call site in the `_model` variant would create a trivially-discharged equivalence (the candidate calling itself in the same scope as the ground truth). We rename recursive call sites in the `_model` variant via `_rename_spec_calls()` to break the loop and force the equivalence check to compare bodies rather than identities.
- *Reserved keywords.* Equivalence-lemma generation is skipped for spec functions whose names are Verus reserved words (`forall`, `exists`, `invariant`, etc.).
- *Where-clause bounds.* Generic where-clause bounds are inlined into generic parameter lists for all synthesized items.

The harness returns `(harness_code, coverage_dict)`, where `coverage_dict` tracks emitted versus expected spec functions, `requires` clauses, and `ensures` clauses. The dict is consumed by the C4 coverage gate (§B.4) to detect silent drops.

E Optimization, Infrastructure, and Hyperparameters

This appendix expands the optimization objective $\mathcal{J}(\theta)$ of Eq. equation 1 (Section B.3): §E.1 details each loss-side component together with the intuition that motivates it, §E.2 describes the orchestrator-side rollout machinery, and §E.3 lists the infrastructure and hyperparameters used during SFT and RL.

E.1 Loss-Side Components

Group-relative advantage (Dr. GRPO). The advantage is the critic-free GRPO (Shao et al., 2024) group baseline $\hat{A}_i = R_i - \frac{1}{G}\sum_{j=1}^{G} R_j$ over $G = 16$ rollouts. Following the Dr. GRPO variant (Prime Intellect, 2024), we omit within-group standard-deviation normalisation: most of our groups are nearly all-correct or all-incorrect, and dividing by a near-zero standard deviation produces unstable, occasionally exploding advantages.

Trust-region mask (DPPO-Binary). We compute the policy-gradient surrogate of DPPO Authors (2025),

$$J_{\text{DPPO}}(\theta) \;=\; \mathbb{E}_{o\sim\pi_{\text{infer}}}\left[\sum_t m_{i,t}(\theta)\,\rho_{i,t}(\theta)\,\hat{A}_i\right], \qquad \rho_{i,t}(\theta) \;=\; \frac{\pi_\theta(o_{i,t}\mid s_{i,t})}{\pi_{\text{infer}}(o_{i,t}\mid s_{i,t})},$$

where $m_{i,t} \in \{0,1\}$ is a per-token trust-region mask. We use the binary instantiation of DPPO Authors (2025), thresholded on the per-token probability gap, with asymmetric thresholds following the DAPO *clip-higher* principle (Yu et al., 2025):

$$m_{i,t}(\theta) \;=\; \begin{cases} 0 & \text{if } \hat{A}_i > 0 \text{ and } \pi_\theta(o_{i,t}\mid s_{i,t}) - \pi_{\text{infer}}(o_{i,t}\mid s_{i,t}) > \tau_{\text{hi}}, \\ 0 & \text{if } \hat{A}_i < 0 \text{ and } \pi_\theta(o_{i,t}\mid s_{i,t}) - \pi_{\text{infer}}(o_{i,t}\mid s_{i,t}) < -\tau_{\text{lo}}, \\ 1 & \text{otherwise,} \end{cases} \tag{3}$$

with $\tau_{\text{lo}} = 0.20$ and $\tau_{\text{hi}} = 0.28$. Because $\rho_{i,t}$ enters the surrogate raw, the policy gradient flows through the un-clipped importance weight on retained tokens and is zeroed on rejected ones. The gating signal is a probability difference bounded in $[-1, 1]$ regardless of how small π_{infer} becomes – exactly the regime where PPO's $|\rho - 1|$ proxy is most unreliable – and the clip-higher asymmetry ($\tau_{\text{hi}} > \tau_{\text{lo}}$) lets verified rollouts be reinforced past the conservative bound a symmetric trust region would impose, while the lower threshold prevents the policy from running away from rollout-time mistakes faster than the next inference-engine sync can correct.

Mismatch-KL regularizer. The DPPO mask is paired with a Kimi-K2.5 mismatch-KL term (Kimi Team, 2025),

$$\mathcal{L}_{\mathrm{KL}}(\theta) \;=\; \kappa \left(\log \pi_\theta(o_{i,t} \mid s_{i,t}) - \log \pi_{\text{infer}}(o_{i,t} \mid s_{i,t}) \right)^2 \;=\; \kappa \, (\log \rho_{i,t})^2, \qquad \kappa = 2{\times}10^{-3}, \tag{4}$$

applied to every trainable token, including those zeroed by the DPPO mask. The two mechanisms are complementary: the binary mask defines the trust region, the quadratic mismatch-KL tightens the variance of the importance-ratio estimator inside it. On tokens where the mask fires ($m_{i,t} = 0$), the KL term is the only source of gradient, supplying a smooth anchor where the binary mask has hard zeros.

Frozen-SFT teacher anchor. A pure RL update on the verifier reward tends, at small model scales, to drift toward short, low-coverage completions whose only virtue is satisfying the most lenient gates. Following Kimi Team (2025) we mediate the bootstrap anchor through the per-token advantage:

$$\hat{A}_{i,t}(\theta) \;=\; \alpha \, \hat{A}_i \;+\; \beta \left(\log \pi_{\text{teacher}}(o_{i,t} \mid s_{i,t}) - \log \pi_\theta(o_{i,t} \mid s_{i,t}) \right)_{\perp}, \tag{5}$$

with $\alpha = 1$, $\beta = 5{\times}10^{-3}$, $(\cdot)_\perp$ denoting a stop-gradient, and the teacher's per-token log-probabilities served by a separate frozen-SFT vLLM replica. Unlike a global KL penalty, the anchor vanishes whenever the policy already agrees with the teacher; on tokens where the policy has drifted, it injects a small gradient back toward the SFT distribution without restricting how much the advantage itself can move the policy on its own.

E.2 Orchestrator-Side Components

Online difficulty filtering and dynamic sampling. A substantial fraction of groups end up uniformly successful or uniformly failed even with shaped rewards, so $\hat{A}_i = 0$ for every rollout and the group contributes zero gradient. We enable DAPO-style dynamic sampling at the orchestrator: groups filtered by zero-advantage, repetition, or gibberish detectors are dropped before reaching the trainer, and the orchestrator over-generates (up to $4\times$ the target batch, at most 8 resampling attempts per step) until the gradient batch is filled with groups exhibiting non-trivial within-group reward variance.

Curriculum-preserving sequential buffer. The default rollout buffer in `prime-rl` (Prime Intellect, 2024) samples examples with replacement, which destroys the round-robin interleaving of Proof-Mode, Spec-Mode, and End2End-Mode (Section **??**). We replace it with a sequential buffer that walks the dataset in JSON order cyclically, so the interleaved schedule survives across epochs and every gradient step contains a coherent slice of modes.

Rollout environment. The environment is built on the `verifiers` library. Per-turn completions are capped at $16{,}384$ tokens with a 64-token minimum (the floor prevents a degenerate first-token-EOS exploit observed in pilot runs); summed across three turns, this fits inside the $40{,}960$-token training context after accounting for the prompt and feedback messages.

E.3 Infrastructure and Hyperparameters

Hardware. RL training runs on a single node of $8\times$ NVIDIA B200 GPUs. Trainer-to-inference weight syncs are NCCL-broadcast directly over IB, which forces `max_async_level=1` and yields the single-step lag relied on by the trust-region analysis above. Verus invocations during reward computation run on 96 CPU workers, with checkpoints mirrored to Blob storage every 100 steps for fault tolerance.

SFT and RL hyperparameters. Table 5 summarizes the two stages.

F Benchmarks and Evaluation Details

This appendix complements the evaluation setup of Section 5: §F.1 details the construction of the SpecBench benchmark, §F.2 presents failure-mode case studies from prior benchmarks that motivated the gate cascade of §B.4, and §F.3 lists the evaluation hyperparameters used across all benchmarks.

Table 5: SFT and RL hyperparameters.

Setting	SFT	RL
Base model	Qwen3-4B-Instruct-2507	Qwen3-4B-Instruct-2507 (SFT ckpt)
Optimizer	AdamW	AdamW
Peak learning rate	2×10^{-5}	10^{-6}
LR schedule	Cosine to 2×10^{-6} floor	Cosine, 100-step warmup
Epochs / steps	3 epochs	4500 steps ($\approx$ 1 epoch)
Per-device batch / Group size G	24	16 rollouts per prompt
Effective batch	—	64 rollouts per step
Context length	16,384 tokens	16,384 (training) / 40,960 (rollout)
Max completion tokens	—	16,384 per assistant turn
Max model turns T	—	3
Discount γ	—	0.9
DPPO τ_{hi} / τ_{lo}	—	0.28 / 0.20
Mismatch-KL κ	—	2×10^{-3}
Teacher anchor β	—	5×10^{-3}
Sampling temperature	—	1.0
Parallelism	DeepSpeed ZeRO-2	FSDP + vLLM (DP=6, TP=1)
Frozen-SFT teacher replica	—	FSDP + vLLM (DP=2, TP=1)
Sequence packing	Enabled	—
Attention	—	FlashAttention-2
Data Shuffling	True	False

F.1 SpecBench Construction

Task sources. Of the 97 tasks (Section 3.1), 84 are drawn from VeruSAGE (Yang et al., 2026) – covering nine real-world Verus-verified systems: Anvil Library, Anvil Controller, IronKV, Memory Allocator, Node Replication, NRKernel, Atmosphere OS, Storage, and Vest – and 13 from VerifyThisBench (Deng et al., 2025) (competition-style verified programming problems).

Selection criteria. A file is retained if it (a) verifies in its original form, (b) fails verification when all spec function bodies are replaced with opaque uninterpreted stubs (ensuring the specifications are non-trivially load-bearing), and (c) admits a complete equivalence harness covering all masked spec functions.

F.2 Failure Modes in VerifyThisBench

We illustrate three concrete failure modes we observed in VerifyThisBench, the only prior specification-generation benchmark for Verus. Each failure mode maps to a gate in §B.4.

Missing `verus!` macro. On "Mergesort with Runs" (2022), PSV-Verus repeatedly hit compile errors on the first five subtasks and finally succeeded on its second attempt at subtask 6 by emitting a Rust file that omitted the `verus!` macro entirely:

```
use std::vec::Vec;
use vstd::prelude::*;

struct Sr { runs: Vec<usize>, data: Vec<i64> }

fn merge(r1: Sr, r2: Sr) -> Sr {
    /* ... merge body ... */
}

fn msort(data: Vec<i64>, l: usize, h: usize) -> Sr {
```

Table 6: Benchmark hyperparameters and evaluation settings used across all benchmarks and harnesses. "Attempts" denotes the maximum number of generation attempts allowed per sample.

Benchmark / Harness	# Samples	Attempts	API Timeout	Verifier Timeout
VerifyThisBench				
Verified + `verus!` present	41	5	10 min	300 s
Coherent / Anti-Cheat (`Gemini-3-Flash-Preview`, Temp. 1)	41	5	10 min	300 s
VerifyThisBench-XS				
Verified + `verus!` present	99	5	10 min	300 s
Coherent / Anti-Cheat (`Gemini-3-Flash-Preview`, Temp. 1)	99	5	10 min	300 s
VeruSage-Bench				
Hands-On Harness	849	5	5 min	20 min
Plain Harness	849	5	5 min	20 min
SpecBench				
Stage-1	97	3	5 min	20 min
Stage-2	97	3	5 min	20 min
Stage-3 Coherent Avg. Score	97	3	5 min	20 min
Stage-3 Coherent Total Score	97	3	5 min	20 min

```
    /* ... recursive split + merge ... */
}

fn main () {}
```

The output contained no specifications and Verus succeeded trivially because there was nothing to verify. We found this pattern in 16 out of 41 PSV-Verus responses, meaning the model could obtain a nominal score of 39% on the benchmark without performing meaningful verification. Once we enforce a non-empty `verus!{}` top-level macro (gate C4), PSV-Verus's score on this benchmark drops to 0%.

Trusted escape hatches. On "Smart Array Copy by Shuffled Subsegments" (2024), Qwen-VeruSyn discovered that inserting `admit()` into proofs allowed arbitrary specifications to verify, and that declaring `spec fn`s as `uninterp` avoided writing meaningful specifications entirely. Both tricks are rejected by gates C1–C2.

Vacuous postconditions. On "Work-Stealing for Task Trees" (2024), Qwen3-Coder-30B-A3B-Instruct learned that `ensures true` verifies vacuously, and emitted functions of the form

```
fn verify_task_execution_once(...) ensures true {
    proof { assert(true); }
}
```

This is caught by the coherency judge.

Shared judge. Unlike the self-coherence mechanism of the original VerifyThisBench paper, our coherency judge is a single LLM (Gemini-3-Flash-Preview) shared across all evaluated models, making scores directly comparable.

F.3 Evaluation Hyperparameters

Table 6 summarizes the evaluation hyperparameters used across all benchmarks and harnesses. Unless otherwise specified, all evaluations use bounded generation attempts, fixed API timeouts, and capped verifier execution budgets to ensure reproducibility and fair comparison across models and evaluation settings. We used a temperature of 0.7 (except for reasoning models that only support a temperature of 1) for VerifyThisBench, and a temperature of 1.0 for both VeruSage-Bench and SpecBench.

G Extended Results and Ablations

G.1 VerifyThisBench-XS: Zero-shot vs. Multi-turn

Section 5 reports our headline VerifyThisBench-XS scores under the three-turn multi-turn rollout we use throughout the paper. Figure 5 compares those final scores against a single-shot zero-shot baseline – the same prompt, but with verifier feedback disabled and no retries – across every model in Table 2.

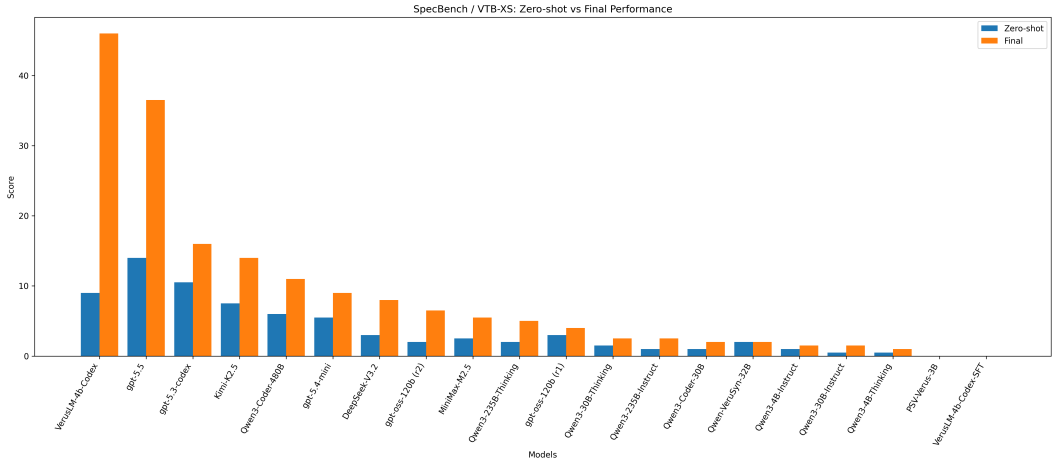

Figure 5: VerifyThisBench-XS pass rate under zero-shot (blue) and multi-turn (orange) evaluation across all baselines and VerusLM-4B-Codex . Models are sorted by final score. Multi-turn lifts every model that already had a non-trivial zero-shot signal, but the lift is largest for VerusLM-4B-Codex ($9 \rightarrow 46$, a $5\times$ gain), reflecting that the model was trained explicitly to act on verifier feedback; weaker baselines that fail the structural checks at turn 1 generally also fail at turn 3 because they cannot recover from the feedback they receive.

Three observations stand out. First, VerusLM-4B-Codex is the largest beneficiary of the multi-turn protocol despite starting near the middle of the zero-shot column; the absolute gain ($+37$ points) exceeds that of any other model, including frontier baselines. This matches our training setup, in which the policy is explicitly trained to consume structured Verus feedback across $T = 3$ turns (§D.2). Second, models with strong zero-shot scores – GPT-5.5, GPT-5.3-codex, Kimi-K2.5, Qwen3-Coder-480B-A35B – gain a smaller but consistent multiplier (1.5–$2.6\times$), suggesting their gains come from refining nearly-correct attempts rather than recovering from structural failures. Third, the bottom half of the ranking (Qwen3 base/Thinking variants under 30B, the prior verification-specific baselines, and the SFT-only checkpoint) sees almost no benefit from extra turns: zero-shot and final scores collapse to within a point of each other, confirming that multi-turn verifier feedback is only useful once the model has the latent capability to act on it.

This pattern is the empirical justification for our reward design: granting partial credit and feeding back diagnostics across turns is what lets a 4B model close the gap to substantially larger baselines on this benchmark.

H Licenses

All datasets, models, and tools used in this work, including VerifyThisBench, VeruSAGE, VeruSyn, Lynette, Verus, and Qwen3, are used in compliance with their respective licenses. These resources are distributed under permissive open-source or Creative Commons licenses, including MIT and CC-BY variants. We provide attribution and relevant usage details in Section 4.1, Section 3.1, Appendix A, and the bibliography.