SELF-SPEC VERIFIABLE CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) may generate unreliable code on corner cases missed by testing, while formal verification can provide machine-checkable guarantees. Recently, researchers have proposed several benchmarks to evaluate the capabilities of LLMs in generating formally verifiable code, where LLMs need to formulate formal specifications, generate the corresponding code, and verify its correctness. However, existing benchmarks have two key limitations: (I) They primarily evaluate specification and code generation stage-wise, with code generation typically conditioned on an oracle specification. This setup overlooks whether strong stage-wise performance translates into end-to-end success. (II) They mainly focus on a single proof-oriented language and mathematically structured tasks, offering limited coverage of tasks common in software development. In this paper, we introduce VERICODEBENCH, a benchmark for self-spec verifiable code generation, where the LLM relies solely on its own generated specification and code throughout the entire process. VERICODEBENCH contains 400 language-native problems across C, Java, Rust, and Python, covering practical concerns in software development. We evaluate specification coverage, code validity, and joint problem-level success across four representative LLMs. We further introduce CODENOVA to enhance the capabilities of LLMs in self-spec verifiable code generation. CODENOVA makes requirements explicit through constraint-guided specification and uses verifier feedback to guide targeted implementation repairs. Experimental results reveal that self-generated specifications remain a major bottleneck, while providing more sophisticated specifications may not necessarily lead to higher verification success rates. CODENOVA substantially improves performance across all evaluation metrics, enabling Claude Sonnet 5 to achieve the strongest results under the self-spec protocol. Our code is anonymously available at `https://anonymous.4open.science/r/ICLR27-VeriCodeBench/`.

1 INTRODUCTION

Large language models (LLMs) are widely used for code generation (Wang & Chen, 2023; Dong et al., 2025; Wang et al., 2025a). To assess the correctness of LLM-generated code, existing evaluation methods primarily rely on test cases (Ryan et al., 2024; Nunez et al., 2024). However, test cases can cover only a limited portion of the input space and therefore cannot guarantee the correctness of code under all possible circumstances, such as rare inputs, unsafe memory states, or unusual exceptional behavior. Formal verification (Hasan & Tahar, 2015; Wang et al., 2025b) provides a potential solution by checking programs against formal specifications, thereby offering machine-checkable guarantees for all states that satisfy the specifications. With the increasing capabilities of LLMs, generating code that is formally verified by construction has emerged as a promising possibility (Aggarwal et al., 2024; Dougherty & Mehta, 2025). Starting from user requirements, an LLM needs to formulate formal specifications, generate the corresponding code, and verify its correctness:

$$\text{requirement} \longrightarrow \text{specification} \longrightarrow \text{code} \longrightarrow \text{verification}.$$

We refer to this process as *verifiable code generation*. Heretofore, several benchmarks (Thakur et al., 2026; Ye et al., 2026; Le-Cong et al., 2025) have been proposed to evaluate the capabilities of LLMs in verifiable code generation. However, existing evaluations leave two complementary gaps.

First, prevailing benchmarks decouple specification generation from downstream code generation and verification. They then simply combine results from these stages to assess the model's capability for "end-to-end" verifiable code generation. Such stage-wise protocols are useful for diagnosing

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.

Table 1: Comparison with representative methods and benchmarks. *Joint Generation* indicates that a method or benchmark covers both specification and code generation. *Self-Spec* means that LLM relies solely on its own generated specification rather than an oracle throughout the entire process.

Method / Benchmark	Joint Generation	Self-Spec	Multilingual	Language	Size
nl2spec (Cosler et al., 2023)	×	×	×	LTL	36
AutoSpec (Wen et al., 2024)	×	×	×	C	251
SpecGen (Ma et al., 2025)	×	×	×	Java	385
ClassInvGen (Sun et al., 2025)	×	×	×	C++	9
PropertyGPT (Liu et al., 2024)	×	×	×	Solidity	23
SLD-Spec (Chen et al., 2025)	×	×	×	C	62
WybeCoder (Gloeckle et al., 2026)	×	×	×	Lean	360
Dafny-Synthesis (Misu et al., 2024)	✓	×	×	Dafny	153
AlgoVeri (Zhao et al., 2026)	×	×	✓	Dafny, Verus, Lean	77
CLEVER (Thakur et al., 2026)	✓	×	×	Lean	161
VERINA (Ye et al., 2026)	✓	×	×	Lean	189
VERICODEBENCH (ours)	✓	✓	✓	**C, Java, Rust, Python**	**400**

individual capabilities, but their combination does **not** indicate the end-to-end capability. Crucially, these stages are **interdependent**. Errors or design choices in a generated specification can alter the difficulty and even the objective of downstream code generation. Conversely, a more complete specification may impose stronger proof obligations and reduce verification success. Consequently, strong specification generation and oracle-spec code generation do not necessarily imply strong end-to-end performance. Evaluating this interaction requires propagating the model-generated specification through the remainder of the pipeline and measuring the resulting joint success.

Second, existing methods and benchmarks are concentrated in a single formal ecosystem, often using proof-oriented languages and mathematically structured tasks. Retrofitting an existing benchmark can instantiate our self-spec protocol, but doing so would still inherit its task distribution, formal language, and verification ecosystem. Such benchmarks enable careful study of proof synthesis, but provide limited exposure to the programming abstractions and failure modes encountered in software development (e.g., pointer validity and frame conditions in C, object mutation and exceptions in Java). We need a common evaluation shape that is end-to-end by construction while remaining faithful to each language and verification toolchain.

To address the aforementioned gaps, we introduce VERICODEBENCH, a multilingual benchmark centered on self-spec verifiable code generation where the LLM relies solely on its own generated artifacts throughout the entire process. VERICODEBENCH comprises 400 language-native problems across four developer-facing languages (C, Java, Rust, and Python) paired with their native verification ecosystems. We separately evaluate whether the generated specification covers requirement-level obligations and whether the resulting code verifies against that exact specification. Their conjunction defines problem-level success. The evaluation process is fully automated and deterministic. VERICODEBENCH reports specification coverage, code validity, and joint problem-level success. We additionally evaluate stage-wise settings to quantify the performance gap between oracle-guided and self-spec generation. Moreover, the multilingual design of VERICODEBENCH serves to broaden coverage to developer-facing programming models and practical software concerns that arise in real-world software development. The four language tracks preserve language-specific syntax, semantics, and proof obligations, exposing models to distinct challenges.

The benchmark also motivates a practical approach to improving verifiable code generation. We introduce CODENOVA[1], an end-to-end approach to verifiable code generation from natural-language requirements. CODENOVA combines requirement formalization, code generation, and verifier-guided repair in a unified generation pipeline. First, CODENOVA extracts atomic behavioral and safety constraints from the requirement, translates them into the target specification language, and self-checks the resulting specification for requirement coverage. The pipeline then generates an implementation conditioned on this specification. Then, CODENOVA uses verifier feedback to analyze unsatisfied proof obligations, generate focused repair candidates, and check them with the native verifier. Overall, CODENOVA addresses requirement formalization and iterative implementation refinement to improve verifiable code generation under the self-spec protocol.

[1]NOVA stands for Natural-language Objectives to Verified Artifacts.

Our main contributions are three-fold:

- We present VERICODEBENCH, a 400-problem benchmark covering programming abstractions and verification challenges common in software development. Its multilingual tracks use native verification toolchains and capture language-specific concerns.
- We define self-spec verifiable code generation, preserving the generated specification through code generation, repair, and verification. We evaluate the whole pipeline with separate specification-adequacy and code-validity measurements.
- We introduce CODENOVA, an end-to-end generation approach combining constraint-guided specification with verifier-guided candidate repair. Experimental results demonstrate the effectiveness of CODENOVA.

2 RELATED WORK

2.1 BENCHMARKS FOR FORMAL REASONING AND VERIFICATION

Benchmarks for formal reasoning address different parts of the path from informal intent to machine-checked correctness. MiniF2F (Zheng et al., 2021) evaluates theorem proving on competition-level mathematics across multiple formal systems, while ProofNet (Azerbayev et al., 2023) pairs undergraduate mathematics problems with informal proofs and Lean statements to support both autoformalization and formal proving. These benchmarks test mathematical reasoning but do not center on synthesizing executable programs with state and memory obligations.

FormalBench (Le-Cong et al., 2025) evaluates specification inference for existing programs, while AlgoVeri (Zhao et al., 2026) aligns classical algorithms across Dafny, Verus, and Lean under equivalent functional contracts. CLEVER (Thakur et al., 2026) separates specification generation, equivalence checking, implementation synthesis, and correctness proofs on Lean-adapted HumanEval problems. VERINA (Ye et al., 2026) supports generation of code, specifications, and proofs on curated Lean tasks. However, these benchmarks primarily evaluate individual stages of the verifiable code generation pipeline rather than the model's ability to autonomously complete the full process. Their application scenarios are also relatively narrow and do not fully reflect the abstractions and failure modes of real-world software development. VERICODEBENCH addresses these gaps with a self-spec setting, where the model relies solely on its own generated specification and code throughout the full verifiable code generation pipeline. Moreover, VERICODEBENCH contains 400 language-native tasks across C, Java, Rust, and Python, covering common programming scenarios and language-specific verification challenges.

2.2 LLM-ASSISTED FORMAL VERIFICATION

LLMs have been applied to several tasks in formal verification, including specification generation, program synthesis, and proof generation. Work in Dafny, Why3, Lean, and related systems (Baksys et al., 2025; Zheng et al., 2021; Azerbayev et al., 2023; Yang et al., 2023) has explored LLM prompting, proof search, and verifier-guided repair to produce machine-checked programs or proofs. Recent efforts (Wen et al., 2024; Ma et al., 2025; Le-Cong et al., 2025; Wu et al., 2025) also target developer-facing verification ecosystems, including ACSL/Frama-C, JML/OpenJML, and Verus, using verifier feedback to refine specifications, implementations, or proof annotations. However, most prevailing systems only focus on individual stages of the verifiable code generation pipeline, without integrating these stages into a unified workflow. Our CODENOVA combines requirement formalization, code generation, and verifier-guided repair in a unified generation pipeline, improving performance across the full verifiable code generation process.

3 SELF-SPEC VERIFIABLE CODE GENERATION

3.1 PROBLEM FORMULATION

Let ℓ denote a programming-language and verification-toolchain track (e.g., C with ACSL and Frama-C/WP). A benchmark instance is a tuple

$$x_i^\ell = (r_i^\ell, h_i^\ell, G_i^\ell, c_i^{\star,\ell}, V^\ell), \tag{1}$$

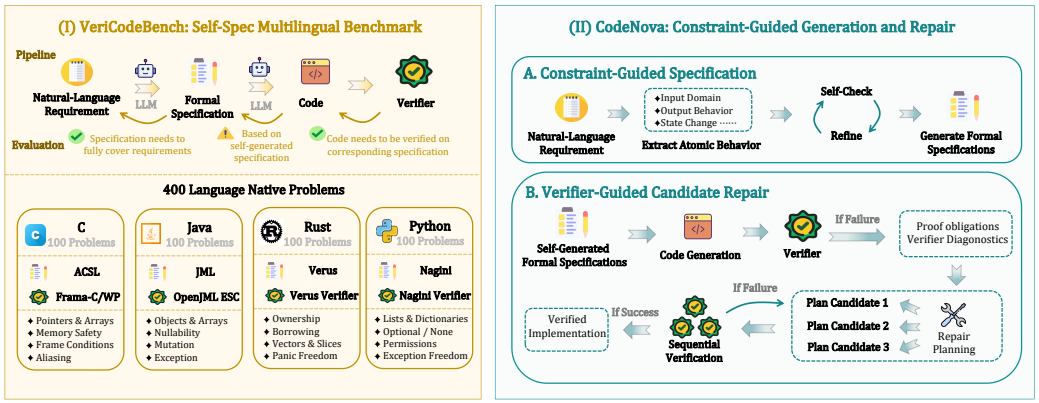


Figure 1: Overview of VERICODEBENCH and CODENOVA. **Left:** VERICODEBENCH evaluates the self-spec pipeline by jointly measuring whether the generated specification covers the requirement and whether the generated code verifies against that same specification, across 400 language-native problems. **Right:** CODENOVA combines Constraint-Guided Specification with Verifier-Guided Candidate Repair to generate verified code under a self-generated specification.

where r_i^ℓ is a natural-language requirement, h_i^ℓ is the provided function signature and language context, G_i^ℓ is a curated set of target specification obligations, $c_i^{\star,\ell}$ is a verifier-accepted reference implementation, and V^ℓ is the language-native verifier. The reference specification and implementation define and validate the benchmark instance; **neither** is exposed to the model.

Given only (r_i^ℓ, h_i^ℓ), a system first generates a language-specific specification

$$\hat{s}_i^\ell = F_{\text{spec}}^\ell(r_i^\ell, h_i^\ell), \tag{2}$$

and then generates an implementation conditioned on that specification,

$$\hat{c}_i^\ell = F_{\text{code}}^\ell(r_i^\ell, h_i^\ell, \hat{s}_i^\ell). \tag{3}$$

The resulting task is therefore not two independent predictions, but the coupled computation

$$(r_i^\ell, h_i^\ell) \longrightarrow \hat{s}_i^\ell \longrightarrow \hat{c}_i^\ell \longrightarrow V^\ell(\hat{c}_i^\ell, \hat{s}_i^\ell). \tag{4}$$

This dependency is essential: the specification is both a prediction whose semantic adequacy must be assessed and an intermediate artifact that constrains downstream code generation.

3.2 VERICODEBENCH

VERICODEBENCH operationalizes the task defined above through three complementary design choices. First, its self-spec evaluation protocol requires the model to rely exclusively on its own generated specifications throughout code generation, repair, and verification, without access to oracle specifications. Second, it assesses specification adequacy and code validity separately. Specification adequacy is scored by a fully automated, deterministic procedure, without LLM-based judging or subjective human assessment. Joint success requires both full coverage of curated requirement-level obligations and verifier acceptance under the generated specification for the same problem. Third, it comprises four language-native tracks with 400 problems, evenly distributed across C, Java, Rust, and Python. The tracks share this evaluation protocol while retaining distinct programming abstractions, specification languages, and verification toolchains. Together, these choices enable end-to-end evaluation of whether models can faithfully formalize requirements and produce implementations that verify against their own specifications across diverse programming settings.

3.2.1 SELF-SPEC EVALUATION

We call a pipeline self-spec when the specification generated in Equation 2 is the mandatory specification supplied in Equation 3. The pipeline may add implementation-level proof annotations or

repair the function body. This is the primary evaluation setting in VERICODEBENCH. It reflects the information boundary of autonomous use: after receiving a natural-language request, the system cannot assume access to a human-written formalization that corrects its own interpretation. It also prevents an oracle specification from masking error propagation. For example, code may verify against a generated specification that omits an important behavior, while a specification that excludes valid inputs may make verification artificially easy. Both outcomes are failures of the end-to-end task even when the verifier accepts the implementation. Moreover, we also conduct stage-wise diagnostic evaluation as a secondary analysis. We replace $\hat{s}_i^\ell$ in Equation 3 with the curated oracle specification $s_i^{\star,\ell}$. This controlled setting estimates code-generation ability when specification error is removed and helps attribute the gap between oracle and self-spec performance.

3.2.2 TWO INDEPENDENT CORRECTNESS CONDITION

Verifier acceptance establishes that code satisfies a specification, not that the specification faithfully captures the requirement. We therefore evaluate two conditions separately. First, specification adequacy compares the generated specification $\hat{s}_i^\ell$ with the manually curated, requirement-level target set $G_i^\ell = \{g_{i,1}^\ell, \ldots, g_{i,M_i}^\ell\}$. This comparison uses the Constraint Entailment Framework (CEF), which combines fixed, language-specific normalization and matching rules with SMT-based entailment checks. Given the curated targets and a generated specification, adequacy scoring is fully automated and deterministic: it requires neither LLM calls nor human adjudication.

A generated specification may split, merge, or restructure target clauses, so CEF tests each target against conjunctions of compatible generated clauses. For a nonempty subset U of normalized clauses, write $\Phi(U) = \bigwedge_{\hat{s}\in U} \hat{s}$. Coverage uses the entailment direction below:

$$\mathrm{Covered}(g_{i,m}^\ell) = \begin{cases} 1, & \exists U:\ g_{i,m}^\ell \models \Phi(U), \quad g_{i,m}^\ell \text{ is a precondition,} \\ 1, & \exists U:\ \Phi(U) \models g_{i,m}^\ell, \quad g_{i,m}^\ell \text{ is a post-, frame-, or exceptional condition,} \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

The reversed direction for preconditions prevents credit for an unnecessarily restrictive domain; guarantees must imply the target behavior. CEF returns the fraction of curated targets with a witness,

$$A_i^\ell = \frac{1}{M_i} \sum_{m=1}^{M_i} \mathrm{Covered}(g_{i,m}^\ell). \tag{6}$$

The denominator M_i is fixed, so missing, unparsable, or unsupported specifications receive zero credit for affected targets. Search and entailment implementation details appear in Appendix A.3.

Second, code validity asks whether the language-native verifier accepts the generated implementation against the exact generated specification ($\mathbb{1}$ denotes the indicator function):

$$C_i^\ell = \mathbb{1}\left[V^\ell(\hat{c}_i^\ell, \hat{s}_i^\ell) = \mathrm{valid}\right]. \tag{7}$$

Syntax errors, failed proof obligations, verifier errors, and timeouts are not counted as valid. The primary problem-level measure requires both conditions to hold for the same benchmark instance:

$$J_i^\ell = \mathbb{1}\left[C_i^\ell = 1 \ \wedge \ A_i^\ell = 1\right]. \tag{8}$$

Thus, *joint success* requires a verifier-accepted implementation and full coverage of the target requirement on the same problem. VERICODEBENCH reports mean specification coverage, code-validity rate, and joint-success. This metric suite distinguishes an implementation failure from a specification failure and rejects verification success obtained under a vacuous specification.

3.2.3 FOUR LANGUAGE TRACKS

C/ACSL/Frama-C. The C track covers 13 categories, including pointer and pointer-block manipulation, mutable and immutable arrays, array slices, loops, byte and string buffers, records, and scalar arithmetic. Specifications use ACSL `requires`, `ensures`, and `assigns` clauses (Baudin et al., 2008) , and implementations are checked with Frama-C's weakest-precondition plugin . The track makes memory-side conditions explicit: functional behavior interacts with pointer validity, separation, buffer bounds, frame conditions, loop invariants, and machine-integer safety. These obligations make C substantially different from a purely algorithmic synthesis benchmark.

Table 2: The four language-native tracks in VERICODEBENCH. Each track contains 100 problems.

Language	Specification	Verifier	Size	Native coverage
C	ACSL	Frama-C/WP	100	Pointers and aliasing, arrays and slices, loops, byte/string buffers, structs, frame conditions, and bounded C arithmetic
Java	JML	OpenJML ESC	100	Arrays, scalar and Boolean APIs, strings, nullability, object invariants, object/field frames, mutation, and exceptional behavior
Rust	Verus	Verus	100	Scalar arithmetic, `Option`/`Result`, vectors and slices, ownership and borrowing, unique mutation, bounds, and panic freedom
Python	Nagini specifications	Nagini	100	Typed scalar functions, `Optional`/`None`, lists and dictionaries, predicate permissions, mutation, and exception freedom

Java/JML/OpenJML. The Java track spans 15 categories centered on arrays, scalar arithmetic, Boolean logic, strings and characters, mutation, nullability, object state, and exceptions. JML specifications (Burdy et al., 2005) are checked by OpenJML (Cok, 2011) in extended static checking mode. Problems involving helper classes additionally provide a fixed *type context* that declares available fields, visibility, and class invariants. This context prevents the model from inventing a different object representation without revealing the curated JML specification. The track therefore tests both functional postconditions and Java-specific behavioral specifications such as `assignable`, object invariants, and exceptional outcomes.

Rust/Verus. The Rust track contains scalar-arithmetic, `Option`/`Result`, immutable-vector, vector-mutation, and ownership/slice families, together with extended variants of each family. Verus specifications (Lattuada et al., 2023) are attached to Rust function signatures through `requires` and `ensures` clauses. The problems emphasize safe-Rust functional correctness while retaining verification challenges not present in ordinary compilation: view-based reasoning about vectors and slices, pre/post-state relations for mutable borrows, bounds and panic freedom, and proof annotations for loops. In contrast to the C track, ownership and borrowing rule out broad classes of aliasing behavior but introduce their own specification vocabulary and proof obligations.

Python/Nagini. The Python track targets the typed subset supported by Nagini (Eilers & Müller, 2018). Its 12 categories cover scalar functions, `Optional`/`None`, mutable list operations, dictionary APIs, and exception freedom, including extended variants. specifications are expressed as executable-looking `Requires(...)` and `Ensures(...)` statements, while Nagini translates verification conditions to its underlying permission logic. Container problems therefore include predicate permissions such as list or dictionary access, and use pre-state expressions when postconditions refer to values before mutation. This track tests whether models can produce statically verifiable specifications and implementations despite Python's otherwise dynamic surface syntax.

3.3 CODENOVA

CODENOVA combines Constraint-Guided Specification (CGS) with Verifier-Guided Candidate Repair (VGCR) to generate verified code under a self-generated specification. Given a requirement r and fixed interface h, CGS produces a specification $\hat{s}$; VGCR then repairs its implementation using the native verifier V. Appendix B provides implementation details.

3.3.1 CONSTRAINT-GUIDED SPECIFICATION

CGS separates requirement interpretation from formalization. It first extracts atomic behavioral and safety constraints $\mathcal{Q} = \{q_1, \ldots, q_m\}$, covering admissible inputs, outputs, state changes, and language-specific obligations, then translates them into a specification:

$$\mathcal{Q} = F_{\text{extract}}(r, h), \qquad s^{(0)} = F_{\text{translate}}(r, h, \mathcal{Q}). \tag{9}$$

The model reviews the specification against the requirement and extracted constraints, identifying omissions and inconsistencies. Its assessment $u^{(j)}$ guides refinement when needed:

$$\begin{aligned} u^{(j)} &= F_{\text{check}}(r, h, \mathcal{Q}, s^{(j)}), \\ s^{(j+1)} &= F_{\text{refine}}(r, h, \mathcal{Q}, s^{(j)}, u^{(j)}). \end{aligned} \tag{10}$$

Review stops when the model reports alignment or no remaining issues, or the review budget is exhausted; the selected specification is saved as $\hat{s}$. This self-check is a generation heuristic: $\mathcal{Q}$ is model-generated, and CEF independently evaluates $\hat{s}$ without external hints. Auxiliary proof hints, such as loop invariants, are kept separate from the function specification and remain mutable.

3.3.2 VERIFIER-GUIDED CANDIDATE REPAIR

The pipeline generates $c_0 = F_{\text{code}}(r, h, \hat{s})$ and checks $V(c_0, \hat{s})$. On failure, VGCR uses verifier diagnostics d_t for the current implementation c_t to plan repairs and propose up to K candidates:

$$P_t = F_{\text{plan}}(r, h, \hat{s}, c_t, d_t), \tag{11}$$

$$\tilde{c}_{t,k} = F_{\text{repair}}(r, h, \hat{s}, c_t, P_t, f_{t,k}), \quad k = 1, \ldots, K. \tag{12}$$

The plan P_t decomposes failures into subgoals with diagnostic evidence and repair hints, such as correcting bounds checks or strengthening loop invariants. The focus $f_{t,k}$ directs a candidate toward combined subgoals, an individual issue, or an alternative implementation. Candidates share the same starting code and plan within a round, and are checked sequentially as complete programs by V. VGCR returns the first verifier-accepted candidate. If none passes, the last candidate submitted to V and its diagnostics seed the next round; candidates rejected by language-subset prechecks do not replace the current code. If the repair budget is exhausted without verifier acceptance, the instance is recorded as a failure. By translating verifier diagnostics into explicit repair subgoals, VGCR guides changes to executable code and local proof annotations. Exploring candidates with different repair focuses provides alternative ways to resolve verification failures, while checking each candidate as a complete program validates whether the proposed changes satisfy the proof obligations.

4 EXPERIMENTS

In this section, we conduct thorough evaluations of frontier LLMs and CODENOVA on VERICODEBENCH. We organize the evaluation around four research questions. **RQ1 (Section 4.2):** How well do current LLMs and CODENOVA perform on end-to-end verified code generation under the self-spec protocol? **RQ2 (Section 4.3):** How does the source of the specification affect downstream code generation? **RQ3 (Appendix C):** How does joint success scale with the sampling budget, as measured by pass@k? **RQ4 (Appendix D):** What do representative case studies reveal about the strengths and limitations in end-to-end verified code generation?

4.1 EXPERIMENTAL SETUP

Models and Inference Configuration. We evaluate DeepSeek V3.2, Kimi-K2.7-Code, Qwen3.6-plus, and Claude-Sonnet-5 on all 400 problems. Each pipeline uses the same base LLM for specification generation, code generation, and any subsequent review or repair. We also evaluate the performance with different sampling budgets on DeepSeek V4.1 Flash.

Implementation Details. We verify each implementation using the native verification toolchain for its respective language: Frama-C/WP for C, OpenJML for Java, Verus for Rust, and Nagini for Python. Code validity requires verifier acceptance under the specification supplied to generation. Syntax or type errors, unproved obligations, verifier errors, and timeouts count as failures. The primary protocol is self-spec. We compare four configurations: **Direct** generates a specification and one implementation without repair; **CGS** replaces direct specification generation with Constraint-Guided Specification; **VGCR** adds Verifier-Guided Candidate Repair to Direct; and **CODENOVA** combines CGS and VGCR. Implementation details are described in Appendix B.3. We report mean per-problem requirement coverage (Req. cov.), the number of verifier-accepted implementations (Valid), and the number that additionally achieve complete requirement coverage (Joint), following Section 3.1. Each track contains 100 problems, so Valid and Joint counts also equal percentage rates.

4.2 END-TO-END SELF-SPEC PERFORMANCE

We evaluate four LLMs under the four configurations in Section 4.1 on all 400 problems using the self-spec protocol. Results are shown in Table 3. Under direct setting, code validity averages 71.0% across the 16 model–language settings, but joint success reaches only 44.1%. Under Direct

Table 3: Results on VERICODEBENCH, with 100 problems per language. Joint denotes joint-success counts; Req. cov. represents mean requirement coverage; Valid means code-valid counts. Direct and VGCR share the same generated specifications, as do CGS and CODENOVA. Joint success requires both complete requirement coverage and verifier acceptance on the same problem. **Bold** values denote the best result. Second best results are underlined.

Base LLM	Method	C (100)			Java (100)			Rust (100)			Python (100)		
		Joint	Req. cov.	Valid	Joint	Req. cov.	Valid	Joint	Req. cov.	Valid	Joint	Req. cov.	Valid
DeepSeek V3.2	Direct	8	0.3247	34	30	0.7113	76	33	0.5522	44	46	0.8267	68
	+VGCR	11	0.3247	60	31	0.7113	82	45	0.5522	68	46	0.8267	71
	+CGS	12	0.5215	30	36	0.7810	56	30	0.5572	49	51	0.8593	65
	CODENOVA	17	0.5215	54	38	0.7810	66	35	0.5572	58	52	0.8593	68
Kimi-K2.7-Code	Direct	16	0.5856	68	56	0.8250	77	57	0.6965	91	76	0.9327	86
	+VGCR	17	0.5856	**86**	59	0.8250	86	57	0.6965	92	76	0.9327	86
	+CGS	21	0.6529	71	55	0.8265	68	43	0.7562	60	73	0.9380	82
	CODENOVA	22	0.6529	**86**	61	0.8265	79	43	0.7562	60	73	0.9380	82
Qwen3.6-plus	Direct	13	0.4480	50	37	0.6522	83	38	0.6667	56	54	0.8940	65
	+VGCR	17	0.4480	64	37	0.6522	91	50	0.6667	76	55	0.8940	66
	+CGS	18	0.6552	39	46	0.7958	82	42	0.7164	54	61	0.9148	71
	CODENOVA	24	0.6552	66	46	0.7958	88	49	0.7164	69	64	0.9148	74
Claude-Sonnet-5	Direct	30	0.6878	73	65	0.9060	90	68	0.8060	96	76	0.9430	79
	+VGCR	**33**	0.6878	80	71	0.9060	**99**	68	0.8060	**98**	76	0.9430	79
	+CGS	23	**0.7157**	56	63	**0.9097**	82	71	**0.8507**	94	**85**	**0.9673**	**90**
	CODENOVA	31	**0.7157**	77	**73**	**0.9097**	97	**73**	**0.8507**	97	**85**	**0.9673**	**90**

setting, Claude-Sonnet-5 performs the best, achievint 60.3% joint success in average across the four tracks. Thus, producing implementations that both verify and fully cover the requirements remains challenging even for the strongest evaluated model.

CGS improves coverage but may increase downstream difficulty. CGS improves requirement coverage in all 16 settings, raising the mean from 0.7162 to 0.7761 and indicating more complete formalization of the target obligations. Yet code validity declines in several settings. For Kimi-K2.7-Code on Rust, coverage rises from 0.6965 to 0.7562 while valid implementations drop from 91 to 60. This pattern is consistent with more demanding specifications making implementation and proof generation harder. Unnecessary complexity or overly restrictive conditions may further compound the difficulty. Coverage alone does not establish specification complexity or rule out over-constraint. Despite this trade-off, CGS improves aggregate joint success from 44.1% to 45.6% in average.

VGCR improves verification under both direct and CGS settings. VGCR uses verifier feedback to repair code and implementation-level proof annotations, including loop invariants. Added to Direct, it raises code validity and joint success significantly. VGCR also helps implement CGS-generated specifications: combining two stages raises validity from 65.6% to 75.7% and joint success from 45.6% to 49.1%. Thus, CODENOVA achieves the best aggregate joint success among the four configurations, with Claude-Sonnet-5 reaching 65.5% across tracks. These gains support the complementarity of specification guidance and verifier-guided repair.

Specification adequacy limits the end-to-end pipeline. Downstream improvements remain bounded by the frozen specification. For DeepSeek V3.2 on C, VGCR increases valid implementations from 34 to 60, but joint successes rise from 8 to only 11. Because generation, verification, and repair cannot restore requirement coverage when the specification omits an obligation. Even verifier-accepted code cannot achieve joint success without an adequate initial specification.

4.3 ERROR PROPAGATION AND ORACLE-SPEC DIAGNOSTIC

We compare code generation conditioned on the model's own specification (*Self*) with code generation conditioned on a curated oracle function specification (*Oracle*). The oracle supplies interface-level obligations, including applicable preconditions, postconditions, and frame conditions; the model must still generate the implementation and its implementation-level proof annotations, such as loop invariants. We evaluate Direct and CODENOVA in both settings. The reported metric is code validity under the respective supplied specification, not self-spec joint success.

Oracle specifications reveal substantial hidden difficulty in self-spec generation. Table 4 shows that oracle specifications improve code validity in nearly every comparison. With oracle specifica-

Table 4: Code validity with self-generated versus oracle specifications. *Self* uses the Direct or CGS specification from Table 3; *Oracle* supplies a curated function specification. Both settings require the model to generate code and implementation-level proof annotations.

Base LLM	Method	C (100)		Java (100)		Rust (100)		Python (100)	
		Self	Oracle	Self	Oracle	Self	Oracle	Self	Oracle
DeepSeek V3.2	Direct	34	69$_{\uparrow 25}$	76	94$_{\uparrow 18}$	44	70$_{\uparrow 26}$	68	90$_{\uparrow 22}$
	CODENOVA	54	69$_{\uparrow 15}$	66	99$_{\uparrow 33}$	58	91$_{\uparrow 33}$	68	93$_{\uparrow 25}$
Kimi-K2.7-Code	Direct	68	70$_{\uparrow 2}$	77	96$_{\uparrow 19}$	91	91$_{-}$	86	92$_{\uparrow 6}$
	CODENOVA	86	89$_{\uparrow 3}$	79	100$_{\uparrow 21}$	60	93$_{\uparrow 33}$	82	95$_{\uparrow 13}$
Qwen3.6-plus	Direct	50	86$_{\uparrow 36}$	83	93$_{\uparrow 10}$	56	74$_{\uparrow 18}$	65	90$_{\uparrow 25}$
	CODENOVA	66	92$_{\uparrow 26}$	88	99$_{\uparrow 11}$	69	80$_{\uparrow 11}$	74	92$_{\uparrow 18}$
Claude-Sonnet-5	Direct	73	87$_{\uparrow 14}$	90	96$_{\uparrow 6}$	96	98$_{\uparrow 2}$	79	95$_{\uparrow 16}$
	CODENOVA	77	91$_{\uparrow 14}$	97	100$_{\uparrow 3}$	97	100$_{\uparrow 3}$	90	100$_{\uparrow 10}$

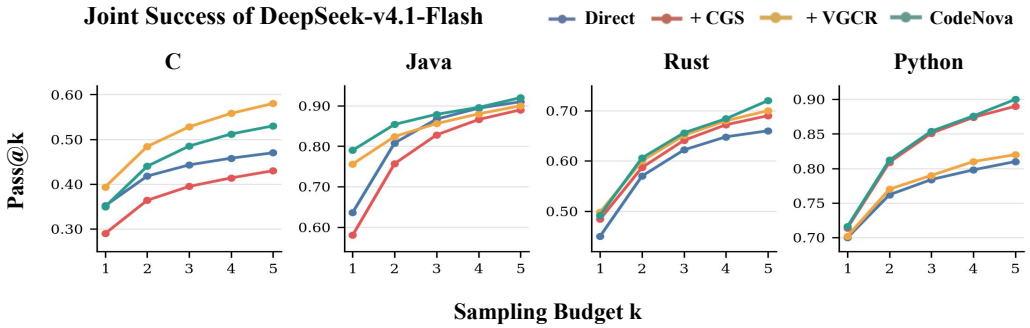


Figure 2: Pass@k joint-success curves across the four language tracks. The panels use language-specific y-axis ranges to improve visual resolution. Detailed analysis are provided in Appendix C.

tions, strong models such as Claude-Sonnet-5 achieve consistently high verification success, indicating that downstream code generation and proof synthesis are considerably more capable when the specification is reliable. This explains why existing benchmarks that evaluate oracle-spec code generation separately and combine it with specification generation performance may provide overly optimistic estimates of end-to-end verified code generation.

Code generation performance is sensitive to which specification it receives. Even with VGCR, CODENOVA using self-generated specifications remains 12.4 percentage points lower than the oracle setting in aggregate validity. This gap demonstrates that improving downstream repair alone cannot fully eliminate the challenges introduced by self-spec generation. However, the gap should not be interpreted as a direct measurement of specification errors, because replacing a self-generated specification with an oracle specification changes both the information available to the model and the verification obligations. Conversely, a small oracle gap does not necessarily imply adequate specifications. For example, Kimi-K2.7-Code on Rust achieves identical Direct validity under Self and Oracle specifications, while its self-spec joint success remains substantially lower. These results highlight that specification quality and verifier acceptance capture different failure modes, and both are necessary for reliable end-to-end verified code generation. Cases can be found in Appendix D.4.

5 CONCLUSION

We introduced VERICODEBENCH, a self-spec multilingual benchmark of 400 problems across C, Java, Rust, and Python. By jointly assessing requirement coverage and native verifier acceptance, the benchmark exposes failures that verification under a generated specification alone can miss. Our evaluation shows that specification quality strongly affects downstream verification and remains a bottleneck for end-to-end success. CODENOVA combines constraint-guided specification with verifier-guided candidate repair to improve aggregate joint success, while its remaining failures highlight the limits of repair under an inadequate specification. These findings motivate methods that advance faithful requirement formalization alongside implementation and proof generation, and evaluation that preserves their dependencies throughout the pipeline.

AI USE STATEMENT

In this work, we used generative AI tools solely to polish the writing, including improving grammar, wording, clarity, and overall readability. We have not used generative AI tools to design or conduct experiments, collect, process, or analyze data, generate research findings, or produce scientific claims. All experiments were conducted by the authors, and all data used in this work were obtained and handled manually by the authors. The remaining required disclosure tasks are not applicable to this work. We have reviewed all AI-assisted writing and verified that it accurately reflects the authors' intended meaning and the results of the work. We take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We release an anonymous implementation of VERICODEBENCH and CODENOVA at `https://anonymous.4open.science/r/ICLR27-VeriCodeBench/`. The main paper describes the self-spec benchmark protocol, the four language–toolchain tracks, and the CODENOVA method, which combines Constraint-Guided Specification (CGS) and Verifier-Guided Candidate Repair (VGCR). The appendix provides the dataset construction and quality-control procedures, implementation details for CGS and VGCR, prompt specifications, and the controlled component evaluation. The anonymous repository contains the benchmark problems and associated data under `benchmarks/`, while the commands for running the requirement-to-code pipelines and evaluations are documented in `docs/requirement_to_code_pipeline.md`. Together, these materials provide the code, benchmark data, configuration, and procedural information needed to reproduce the reported experiments.

REFERENCES

Pranjal Aggarwal, Bryan Parno, and Sean Welleck. Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement. *arXiv preprint arXiv:2412.06176*, 2024.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W Ayers, Dragomir Radev, and Jeremy Avigad. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *arXiv preprint arXiv:2302.12433*, 2023.

Mantas Baksys, Stefan Zetzsche, Olivier Bouissou, and Sean B Holden. Minif2f-dafny: Llm-guided mathematical theorem proving via auto-active verification. *arXiv preprint arXiv:2512.10187*, 2025.

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi c specification language. *CEA-LIST, Saclay, France, Tech. Rep. v1*, 2:79, 2008.

Lilian Burdy, Yoonsik Cheon, David R Cok, Michael D Ernst, Joseph R Kiniry, Gary T Leavens, K Rustan M Leino, and Erik Poll. An overview of jml tools and applications. *International journal on software tools for technology transfer*, 7(3):212–232, 2005.

Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. Sld-spec: enhancement llm-assisted specification generation for complex loop functions via program slicing and logical deletion. *arXiv e-prints*, pp. arXiv–2509, 2025.

David R Cok. Openjml: Jml for java 7 by extending openjdk. In *NASA Formal Methods Symposium*, pp. 472–479. Springer, 2011.

Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In *International Conference on Computer Aided Verification*, pp. 383–396. Springer, 2023.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*, 2025.

Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pp. 72–79. IEEE, 2025.

Marco Eilers and Peter Müller. Nagini: a static verifier for python. In *International Conference on Computer Aided Verification*, pp. 596–603. Springer, 2018.

Fabian Gloeckle, Mantas Baksys, Darius Feher, Kunhao Zheng, Amaury Hayat, Sean B Holden, Gabriel Synnaeve, and Peter O'Hearn. Wybecoder: Verified imperative code generation. *arXiv preprint arXiv:2603.29088*, 2026.

Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pp. 7162–7170. IGI Global Scientific Publishing, 2015.

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*, 7(OOPSLA1):286–315, 2023.

Thanh Le-Cong, Bach Le, and Toby Murray. Can llms reason about program semantics? a comprehensive evaluation of llms on formal specification inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21991–22014, 2025.

Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*, 2024.

Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 16–28. IEEE, 2025.

Md Rakib Hossain Misu, Cristina V Lopes, Iris Ma, and James Noble. Towards ai-assisted synthesis of verified dafny methods. *Proceedings of the ACM on Software Engineering*, 1(FSE):812–835, 2024.

Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. *arXiv preprint arXiv:2409.10737*, 2024.

Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1 (FSE):951–971, 2024.

Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark Barrett, David Dill, Xiaokang Qiu, and Shuvendu K Lahiri. Classinvgen: Class invariant synthesis using large language models. In *International Symposium on AI Verification*, pp. 64–96. Springer, 2025.

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. *Advances in Neural Information Processing Systems*, 38, 2026.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. In *International Conference on Learning Representations*, volume 2025, pp. 2432–2478, 2025a.

Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pp. 284–289. IEEE, 2023.

Weiqi Wang, Marie Farrell, Lucas C Cordeiro, and Liping Zhao. Supporting software formal verification with large language models: An experimental study. In *2025 IEEE 33rd International Requirements Engineering Conference (RE)*, pp. 423–431. IEEE, 2025b.

Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*, pp. 302–328. Springer, 2024.

Valentina Wu, Alexandra Mendes, and Alexandre Abreu. Specification-guided repair of arithmetic errors in dafny programs using llms. In *International Conference on Software Engineering and Formal Methods*, pp. 261–278. Springer, 2025.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. Verina: Benchmarking verifiable code generation. In *International Conference on Learning Representations*, volume 2026, pp. 38933–38972, 2026.

Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V Veeravalli, Aarti Gupta, and Sanjeev Arora. Algoveri: An aligned benchmark for verified code generation on classical algorithms. *arXiv preprint arXiv:2602.09464*, 2026.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

A Implementation Details of VeriCodeBench

A.1 Dataset Construction and Quality Control

We construct every track in four stages. First, each problem is assigned a concise natural-language requirement and a fixed language-level signature. The signature removes irrelevant interface search while leaving behavioral formalization to the model. When a task depends on auxiliary declarations, such as a Java record-like class, the input also includes the minimal type context needed to compile and verify the target method. The primary protocol exposes only the requirement, signature, and such public context; target clauses and reference code remain hidden.

Second, we manually curate a set of atomic target obligations from the intended behavior. Each target record stores a stable identifier, clause type, formal expression, natural-language gloss, and provenance. Depending on the track, clause types include preconditions, postconditions, frame or assignability conditions, and exceptional behavior. The target set is not obtained by blindly extracting every annotation from a reference file. Reference implementations often contain proof-oriented conditions—for example pointer-validity facts, permission predicates, overflow guards, or helper invariants—that may be necessary for a particular proof but are not always part of the requirement's semantic target. Curators include such conditions only when they are part of the benchmark's intended specification. Conversely, when a concise requirement leaves a verification-relevant detail implicit, the reference specification is used to resolve the instance's operational meaning. The released target clauses make these decisions explicit and auditable.

Third, every instance is paired with a manually constructed implementation and full native specification. Reference programs may contain loop invariants, frame annotations, ownership/view facts, or permission assertions required by their respective verifier. They are checked with the same verifier family used to score generated artifacts. This serves two purposes: it establishes that the task is realizable under the fixed interface, and it validates the interaction between the curated behavioral targets and the language-specific verification environment. The reference implementation is never used as a candidate answer in model evaluation.

Finally, automated dataset checks validate identifier alignment, paths, signatures, clause schemas, and target counts between the requirement and ground-truth manifests. The current release contains exactly 100 requirements and 100 corresponding target records in each track. Reference suites are run in toolchain-specific containers to control verifier and solver dependencies. We retain older subsets separately for reproducibility, but all results in this paper use the 100-problem manifests.

A.2 Design Principles

The benchmark is organized around three principles.

Dependency fidelity. The benchmark preserves the causal dependency of autonomous generation. A model first produces a specification and must then implement the program under that same specification. The specification is treated as a frozen intermediate artifact during code generation and verifier-guided repair. This prevents a system from obtaining an easier proof by silently deleting a postcondition, strengthening a precondition, or substituting the benchmark's target specification. Consequently, the primary result captures error propagation from requirement interpretation to specification and finally to verified implementation.

Language-native verification. Single-ecosystem benchmarks provide focused measurements within one formal setting; our multilingual design adds complementary breadth across programming models and verifier semantics. Rather than translating one abstract task set into several syntaxes, we evaluate the abstractions developers actually encounter in each ecosystem. The four tracks share the same high-level interface—requirement, specification, implementation, verifier, and coverage targets—but use language-specific problems and native specification idioms. This choice exposes difficulties such as pointer validity in C, object frames in Java, ownership-aware mutation in Rust, and permission-based container reasoning in Python. Cross-language differences should therefore be interpreted as differences between complete task–toolchain combinations, not as controlled measurements on parallel translations.

Joint evaluation with diagnostic controls. Verifier acceptance and requirement alignment are recorded independently, then combined only at the problem level as defined in Equation 8. The self-spec path is the benchmark's primary protocol. At the same time, the curated targets and reference specifications support oracle-spec and stage-wise controls that localize failures without changing the primary task. All variants use the same fixed target denominator, so a generation failure or unparsable specification cannot improve coverage by removing difficult obligations from evaluation.

A.3 Constraint Entailment Details

CEF searches compatible generated clauses in increasing conjunction size, from single clauses through all nonempty subsets, and stops at the first witness. Thus one target requires at most $2^N - 1$ entailment checks. In our implementation, we set $N = 5$ and limits the maximum search attempts to $2,000$. Each query $L \models R$ is reduced to unsatisfiability of $L \wedge \neg R$ in an SMT solver after conservative, language-specific canonicalization. Canonicalization removes superficial differences such as parameter names and supported Boolean rewrites while preserving pre/post-state distinctions. Exceptional constructs outside the shared Boolean fragment use conservative type-aware matching. The LLM is never asked to prove equivalence; coverage depends on the specification logic itself. Missing, unparsable, or unsupported specifications receive zero credit for affected targets.

A.4 Benchmark Artifacts

VeriCodeBench releases both static dataset artifacts and the machinery needed to reproduce an end-to-end run. For each problem, the static release contains the requirement and category metadata, fixed signature and optional type context, atomic ground-truth obligations, a complete reference specification, and an annotated reference implementation. Track-level scripts verify reference programs and invoke the appropriate requirement-to-code pipeline.

For every model run, the pipeline stores four layers of evidence: (i) the canonical generated specification, including structured clauses and the exact specification text; (ii) the generated source file; (iii) the normalized verifier verdict and raw diagnostics; and (iv) the CEF and problem-level summary reports. The canonical representation is language aware: ACSL and JML use source-level specification blocks, Verus stores structured signature clauses, and Nagini stores both specification statements and normalized clauses. This representation is the source of truth shared by code generation, verification, and coverage scoring.

The pipeline operationalizes the self-spec boundary at the artifact level. The generated specification is supplied to code generation and remains frozen during repair; only method bodies, helper proof code, and implementation-level annotations such as loop invariants may change. The verifier-facing form is language specific: ACSL and JML blocks appear above the target function, Verus clauses are rebuilt into the function signature, and Nagini statements appear at the beginning of the function body. The Java and Python adapters reinsert the canonical specification after generation and repair, while the Rust adapter derives the signature from its stored structured clauses; the C pipeline passes the same stored ACSL block to every downstream prompt. Specification-placement and enforcement actions are retained in run metadata, and an explicit consistency check rejects Java source/specification mismatches. Together with raw diagnostics and fixed manifests, these artifacts make each reported failure traceable to specification generation, code generation, specification preservation, or the native verifier.

B Implementation Details of CodeNova

This section supplements the CGS and VGCR procedures in Section 3.3, including their language-specific representations, candidate scheduling, and controlled evaluation.

B.1 Constraint Representation and Specification Refinement

Atomic constraints. Each extracted constraint expresses a single condition in concise mathematical or logical language. The structured representation groups constraints by role, including admissible inputs, return-value behavior, state changes, and invariants. Track-specific fields capture frame conditions and exceptional behavior in Java, mutation and panic freedom in Rust, and permissions and exception freedom in Python. The fixed signature and public context constrain extraction to the supplied interface and object representation. For example, updating one array element while preserving the others requires both an updated-value relation and an unchanged-region condition. Distinct return cases and boundary inputs should likewise remain explicit. The extracted set $\mathcal{Q}$ can nevertheless be incomplete; it is not the curated target set G used by CEF.

Native translation and proof hints. Translation retains access to the original requirement and interface, distinguishes input assumptions from output guarantees, and preserves pre/post-state references for mutation. Language-specific instructions guide ACSL pointer validity and frames, JML assignability, Verus sequence views and mutable borrows, and Nagini container permissions. The specification artifact may also contain auxiliary hints for code generation, including candidate loop invariants, loop frames, and termination measures where supported. These are stored separately from the function specification because they depend on a future implementation and may change during repair. They are neither assumptions on every function call nor additional requirement-level evaluation targets.

Review and fallback. The self-check returns a structured alignment judgment, missing constraints, inconsistencies, and refinement hints. As in Equation 10, refinement uses those findings to revise the specification. Review ends when the model reports alignment or lists no missing or inconsistent items, or when its configured budget is reached. The adapters validate the specification representation and apply their supported-syntax checks and fallback rules before saving $\hat{s}$. Budget exhaustion does not establish alignment. Neither CEF coverage nor its entailment witnesses are available to this process.

B.2 Repair Planning and Candidate Scheduling

Plan structure. The verifier-derived plan contains a failure summary, subgoals, an overall repair strategy, and potential semantic risks. Each subgoal records an issue type, supporting diagnostic evidence, and a proposed implementation or annotation change. Issues can concern syntax and specification placement, postconditions, memory or permission safety, bounds and overflow, loop invariant establishment or preservation, frames, or termination. Timeouts and unclassified failures can also trigger planning. These subgoals are LLM interpretations of diagnostics, not independently proved lemmas; a failed or timed-out proof attempt is not treated as a concrete counterexample.

Focus schedule and selection. For each plan, the focus schedule first integrates all subgoal hints, then prioritizes individual subgoals in order, and finally requests an alternative implementation if the candidate budget reaches that focus. The schedule repeats if necessary. Each candidate is generated from the same c_t and P_t within a round and undergoes language-specific processing before native verification. Checking the complete program preserves interactions between subgoals: a postcondition repair must still satisfy memory safety, framing, and all other verifier obligations. The first accepted candidate is returned; otherwise, the last candidate submitted to the verifier becomes c_{t+1}. Precheck-rejected candidates do not replace the current implementation; if all candidates are rejected, the round records a failure. Selection uses verifier acceptance, without an LLM score or an assumption that unproved obligations decrease monotonically. On budget exhaustion, the final implementation is retained for analysis.

B.3 Controlled Component Evaluation

Direct generates a specification and one implementation, then verifies it once. CGS changes specification generation while retaining that single code attempt; VGCR adds repair under Direct's saved specification; full CodeNova combines both stages. Each repair run reuses the exact specification and initial code from its corresponding run without repair. Neither artifact is resampled, so specification coverage is identical within each pair. This isolates repair gains from specification-generation effects. Extracted constraints, self-check findings, repair plans, candidate focuses, and verifier outcomes are recorded for analysis.

C Pass@k Analysis

To address **RQ3**, we conduct evaluation on the performance with different sampling budgets. We adopt DeepSeek v4.1 Flash as the base model, measuring pass@k performance under four configurations. Figure 2 highlights a central distinction between improving the specification and improving the implementation. CGS can lower the curve in some language tracks, especially at small sampling budgets, because a more sophisticated specification may also introduce additional proof obligations or a less usable proof interface. In other tracks and at larger budgets, however, CGS raises the attainable success level by making the intended behavior easier to recover. Its effect is therefore conditional rather than uniformly positive or negative.

VGCR is more consistently beneficial across the curves. Verifier feedback provides a mechanism for converting additional samples into targeted repairs, so the VGCR curves generally dominate their Direct counterparts or approach the same ceiling more quickly. This effect is complementary to CGS: when CGS supplies a better-aligned specification, VGCR can spend its repair budget on implementation and proof obligations rather than compensating for missing requirements. The combined CodeNova curves consequently recover cases where CGS alone is harmful while retaining its gains where specification guidance is useful.

The curves also show why pass@1 can understate the potential of the full pipeline. Several configurations continue to improve as the budget grows, with CodeNova often retaining useful headroom through pass@5. This pattern is consistent with complementary candidate diversity: different samples may discover distinct specifications, implementations, or verifier-proof annotations. At the same time, the flattening of some curves indicates that sampling cannot remove a persistent specification bottleneck or an unrealizable obligation. Language-specific verifier semantics and specification idioms further shape both the initial success rate and the rate of saturation, so pass@k should be read as a budget-sensitivity diagnostic rather than a single ranking independent of the track.

D Case Study

This section gives artifact-level examples of how constraint extraction and verification-guided repair affect the two components of joint success. Examples are selected from the Claude-Sonnet-5 runs.

D.1 CGS CAN REPAIR MISSING BEHAVIORAL COVERAGE

Here CGS makes the specification more explicit while preserving verifier validity, so a previously incomplete specification becomes a joint success.

C, problem 13 (separate equal and unequal behaviors). The base specification expresses equality with one biconditional and covers only $4/6$ target clauses. CGS splits the two behaviors into disjoint, complete ACSL behaviors:

```
behavior all_equal:
  assumes \forall integer i; 0 <= i < n ==> a[i] == b[i];
  ensures \result == 1;
behavior not_equal:
  assumes \exists integer i; 0 <= i < n && a[i] != b[i];
  ensures \result == 0;
complete behaviors;
disjoint behaviors;
```

The generated code remains valid and coverage rises to $6/6$, producing a joint success.

Rust, problem 36 (vector mutation). The base specification uses one subrange equation and covers $2/3$ targets. CGS spells out both the length change and the unchanged prefix:

```
ensures v.len() == old(v).len() - 1
ensures forall|i: int| 0 <= i < v.len() ==> v[i] == old(v)[i]
```

The decomposition matches the evaluator's independent length, changed-element, and frame targets, giving $3/3$ coverage while retaining verifier acceptance.

Python, problem 57 (complementary branch). The base specification covers only the true branch $(1/2)$:

```
Ensures(Implies(flag, Result() == x))
```

CGS adds the missing false branch, raising coverage to $2/2$:

```
Ensures(Implies(flag, Result() == x))
Ensures(Implies(not flag, Result() == y))
```

D.2 CGS MAY ALSO MAKE AN ALREADY-VALID PROGRAM HARDER TO PROVE

These cases have full target coverage in both runs, but the richer specification introduces a proof obligation that the native verifier cannot discharge.

C, problem 5 (pointer aliasing and frames). The base program verifies with coverage $2/2$. CGS adds the semantically useful frame fact that the read pointer is unchanged:

```
/*@ ensures *a == \old(*a) + \old(*b);
    ensures *b == \old(*b); */
```

The implementation changes a but says nothing that excludes a and b from aliasing. Frama-C/WP proves four of five goals and times out on the new frame goal (10 seconds); the base run proves four of four. Thus CGS preserves coverage but changes `code_valid` from true to false.

Java, problem 16 (existential loop invariant). The base specification and implementation verify with coverage $5/5$. CGS uses an existential witness in the loop invariant:

```
/*@ loop_invariant (\exists int k; 0 <= k && k < i && a[k] == min) || i == 0;
  @ loop_invariant \forall int k; 0 <= k && k < i; min <= a[k];
  @*/
```

OpenJML cannot establish preservation of the first invariant for the generated loop. The specification remains fully covered ($5/5$), but validity is lost because the proof interface is stronger than the implementation can establish.

Python, problem 94 (length-preservation disjunction). The base specification verifies with coverage $4/4$. CGS adds a length case split:

```
Ensures(key in d)
Ensures(len(d) == Old(len(d)) or
        (key not in Old(d) and len(d) == Old(len(d)) + 1))
Ensures(d[key] == key)
```

Nagini fails to prove the disjunctive postcondition for the generated update, although all four target clauses are covered. This is a semantic proof difficulty rather than a parser or type failure.

D.3 VGCR COMPENSATES HARDER SPECIFICATIONS

The following cases satisfy the strictest preservation test: base and CGS+VGCR are joint successes, while CGS alone is not. VGCR edits executable code or implementation-level annotations only; the CGS specification and its coverage denominator remain fixed.

Java, problem 16 (explicit witness). VGCR replaces the unprovable existential reasoning with a concrete index while retaining the same $5/5$ specification coverage:

```
int min = a[0];
int minIdx = 0;
int i = 1;
/*@ loop_invariant 0 <= minIdx && minIdx < i;
  @ loop_invariant a[minIdx] == min;
  @ loop_invariant \forall int k; 0 <= k && k < i; min <= a[k];
  @*/
while (i < a.length) {
    if (a[i] < min) { min = a[i]; minIdx = i; }
    i++;
}
```

Java, problem 3 (exception guards). CGS adds null and empty-array exceptional behaviors. VGCR makes those cases explicit before the indexed read, restoring validity and $4/4$ joint success:

```
if (a == null) {
    throw new NullPointerException();
}
if (a.length == 0) {
    throw new ArrayIndexOutOfBoundsException();
}
return a[0];
```

C, problem 21 (header restoration). The CGS precondition mentions `INT_MIN` and `INT_MAX`. The first generated source omits `<limits.h>`, so Frama-C aborts during specification parsing. VGCR restores the include; both target clauses remain covered and the program returns to joint success.

Rust, problem 23 (mutability repair). CGS adds the frame fact `ensures *v == *old(v)`, but the generated signature uses an immutable reference where the implementation expects a mutable one. VGCR repairs the reference mutability and preserves the specification, turning the type error into a verified artifact. This is a code/interface repair, not a specification weakening.

D.4 CODE GENERATION PERFORMANCE IS SENSITIVE TO RECEIVED SPECIFICATION

Oracle specifications can remove failures that VGCR cannot repair. For C problem 10 (`array_double.c`), the self-spec Direct artifact times out with 12 of 13 goals proved, and self-spec VGCR does not recover it. The failing proof interface omits the loop variable from the loop

frame and does not expose the arithmetic range needed for doubling array elements. In the oracle-conditioned Direct artifact, the loop frame includes both the index and the modified array range, and the contract makes the no-overflow condition explicit; Frama-C/WP proves all 14 goals. Rust problem 42 (`VecCopyPrefix`) exhibits the same sensitivity at the contract-syntax level:

```
Self:   ensures dst.len() == old(dst).len()
Oracle: ensures final(dst).len() == old(dst).len()
        ensures forall|i: int| 0 <= i < n ==> final(dst)[i] == src[i]
```

Verus rejects the self-generated form because a mutable-reference postcondition must disambiguate the pre- and post-state with `old`/`final`; the oracle-conditioned Direct implementation verifies. These cases show that a specification can hinder downstream generation through its proof interface even when repair is available.

Oracle specifications can also be harder to satisfy. The direction is not universal. For Rust problems 54 (`SquareBounded`) and 59 (`SafeMulSmall`), the self-spec Direct implementations verify, whereas the oracle-conditioned implementations fail with possible arithmetic overflow. The oracle contracts require the model to establish safe multiplication, an obligation absent from the self-generated contracts. Similarly, Java problem 16 (`ArrayMin`) verifies under the self-generated Direct contract but fails under the oracle-conditioned artifact because the generated `loop_writes` clause omits modified locals. An oracle contract may therefore reveal real safety obligations while also making the associated code-and-proof task strictly more demanding.

Equal validity can conceal a large adequacy gap. The clearest example is Kimi-K2.7-Code on the Rust track. Direct generation has exactly the same validity under Self and Oracle specifications, yet the self-generated contracts cover substantially fewer requirement targets:

Table 5: Kimi-K2.7-Code Direct results on Rust. Oracle contracts have full target coverage by construction, whereas equal validity under Self does not imply adequate generated specifications.

Specification source	Validity	Coverage	Joint success
Self	91	0.697	57
Oracle	91	1.000	91

At the problem level, the equal validity totals arise from offsetting changes. Oracle contracts recover four failures (`OptionIncrement`, `IsEven`, `ResultMapIncrement`, and `VecSwapFirstLast`) but make four previously valid cases fail (`VecZeroPrefix`, `SquareBounded`, `SafeMulSmall`, and `VecTruncateOne`). The unchanged total of 91 therefore hides both directions of specification sensitivity, while the 34-point joint-success gap exposes the missing requirement coverage in the self-generated contracts.

D.5 Takeaway

Across these artifacts, CGS improves the semantic completeness of specifications and therefore micro coverage, but can lower validity when it introduces alias, quantifier, disjunction, or exceptional-proof obligations. VGCR is useful precisely at this boundary: it uses verifier diagnostics to change the implementation while keeping the contract frozen. The resulting behavior explains why the aggregate table can show higher coverage and higher final joint success even when CGS alone lowers verifier acceptance. The oracle cases further show that verifier acceptance and specification adequacy are non-interchangeable: validity is sensitive to the supplied proof interface, while joint success additionally detects contracts that omit required behavior.

E Prompt Records

This appendix gives compact, semantically complete records of every prompt stage used by the released pipeline, with runtime fields represented by their names (requirement, specification, code, and diagnostics). System prompts require either strict JSON or code only, while user prompts

define the schema and the frozen-specification boundary. The executable templates contain the same instructions with literal JSON schemas and language-specific placeholders. CGS means the constraint-extraction, constraint-to-specification, and self-check/refinement prompts. VGCR means the repair-analysis and focused candidate prompts. The same stage order is used in all four native verifier tracks.

E.1 C / ACSL

```
SYSTEM (specification): You are an expert in ACSL specification design for C code.
Return strict JSON only.

USER (specification): Given the requirement below, design a minimal but useful
ACSL function specification. Return JSON with function_signature, acsl_block,
code_annotation_hints (loop_invariants, loop_assigns, loop_variants), and notes.
The signature hint must be copied exactly. The ACSL block is a function
contract only; put loop annotations in code_annotation_hints.

SYSTEM (CGS extraction): You are an expert in translating requirements into
verification constraints. Return strict JSON only.

USER (CGS extraction): Extract structured verification constraints from the
requirement. Return function_signature, atomic preconditions, postconditions,
invariants, and notes. Keep every constraint testable and concise.

SYSTEM (CGS translation): You are an expert ACSL specification engineer.
Convert constraints into a complete ACSL contract. Return strict JSON only.

USER (CGS translation): Create ACSL spec JSON using requirement + structured
constraints + signature hint. Include requires/assigns/ensures; keep loop
annotations outside the function contract and use valid/valid_read for pointers.

SYSTEM (CGS self-check): You are a strict requirement-spec alignment reviewer.
Return strict JSON only.

USER (CGS self-check): Check whether the generated specification misses
requirement constraints. Return is_aligned, missing_constraints,
inconsistent_items, refinement_hints, and notes. Use empty arrays when aligned.

SYSTEM (CGS refinement): You refine ACSL specs to improve requirement
alignment. Return strict JSON only.

USER (CGS refinement): Refine the generated specification from the original
requirement, structured constraints, current specification, and alignment
findings. Return the same specification schema. Preserve the signature; keep
the ACSL block as a function contract and move loop annotations to code hints.

SYSTEM (code generation): You are an expert C developer writing
verification-friendly code. Output C code only.

USER (code generation): Implement one C function from the requirement,
signature, frozen ACSL block, and code-annotation hints. Keep exactly one
matching implementation, keep the contract directly above it, do not change
the contract, and place loop annotations immediately before their loops.

SYSTEM (VGCR analysis): You are a verification-guided C/ACSL repair planner.
Return strict JSON only.

USER (VGCR analysis): Analyze a failed Frama-C/WP attempt. Return a failure
summary, typed subgoals with verifier evidence and repair hints, a global
strategy, and risk_notes. Treat the ACSL function contract as frozen.

SYSTEM (VGCR candidate): You are an expert C developer using
prove-as-you-generate repair. Output C code only.

USER (VGCR candidate): Repair the implementation using the verifier-derived
plan. Keep the frozen ACSL block directly above the function; do not change,
weaken, reorder, or delete contract clauses. Change only the body and
statement-level loop annotations. Output complete C source and no explanation.
```

E.2 JAVA / JML

```
SYSTEM (specification): You are an expert in JML specification design for
Java code. Return strict JSON only.

USER (specification): Given the Java requirement, class name, signature hint,
and fixed type context, return function_signature, jml_block,
code_annotation_hints, helper_declarations, and notes. Use OpenJML-compatible
requires, assignable, ensures, and exceptional behavior. Preserve exact field
names and narrow mutation frames.

SYSTEM (CGS extraction): You are an expert in translating Java requirements
into verification constraints. Return strict JSON only.
```

```
USER (CGS extraction): Extract atomic preconditions, postconditions,
frame_conditions, exceptional_behaviors, invariants, and helper declarations.
Include nullability, bounds, frames, object invariants, overflow, and
exception constraints when relevant.

SYSTEM (CGS translation): You are an expert JML specification engineer.
Convert Java verification constraints into a complete OpenJML-compatible JML
contract. Return strict JSON only.

USER (CGS translation): Create JML spec JSON using requirement, constraints,
signature hint, and exact type context. Emit independent clauses, narrow
assignable frames, and valid \old/\result/\forall/\exists syntax.

SYSTEM (CGS self-check): You are a strict Java requirement/JML-spec alignment
reviewer. Return strict JSON only.

USER (CGS self-check): Check for missing or inconsistent Java constraints.
Reject renamed fields, widened frames, combined atomic requirements, and
conditional weakenings. Return is_aligned, missing_constraints,
inconsistent_items, refinement_hints, and notes.

SYSTEM (CGS refinement): You refine JML specs to improve Java requirement
alignment. Return strict JSON only.

USER (CGS refinement): Refine the current JML specification using the original
requirement, structured constraints, alignment findings, and fixed type
context. Preserve exact field names, atomic clauses, narrow frames, and
unconditional guarantees; keep loop annotations outside the method contract.

SYSTEM (code generation): You are an expert Java developer writing
OpenJML-friendly code. Output Java source code only.

USER (code generation): Implement one complete source file from the
requirement and frozen JML specification. Define one public class with the
required name, keep the contract directly above the method, include required
helpers, and add only implementation-level loop annotations.

SYSTEM (VGCR analysis): You are a verification-guided Java/JML repair planner.
Return strict JSON only.

USER (VGCR analysis): Analyze the OpenJML result and produce typed subgoals,
evidence, repair hints, a global strategy, and risk notes. Treat the JML method
contract as frozen; prefer body, helper, and loop-annotation changes.

SYSTEM (VGCR candidate): You are an expert Java/OpenJML developer using
prove-as-you-generate repair. Output Java source code only.

USER (VGCR candidate): Repair the implementation from the verifier-derived
plan. Keep the frozen JML contract directly above the method and do not change,
weaken, reorder, or delete clauses. Output one complete public class.
```

E.3 RUST / VERUS

```
SYSTEM (specification): You are an expert in Rust verification with Verus.
Return strict JSON only.

USER (specification): Given the Rust requirement and signature hint, return
function_signature, verus_contract, atomic verus_clauses, code_annotation_hints,
and notes. Use named returns, old/final for mutable references, and v@ sequence
views. Do not invent null, validity, ownership, wf, or panics predicates.

SYSTEM (CGS extraction): You are an expert in translating Rust requirements
into Verus verification constraints. Return strict JSON only.

USER (CGS extraction): Extract atomic preconditions, postconditions,
panic_freedom, mutation_frame, and invariants. Include ownership/borrowing,
vector bounds, Option/Result cases, overflow, and mutation effects when needed.

SYSTEM (CGS translation): You are an expert Verus specification engineer.
Convert Rust constraints into a complete Verus function contract. Return strict
JSON only.

USER (CGS translation): Emit requires and ensures for every functional case,
return value, mutation, and unchanged region. Use only v@.len(), v@[i as int],
old(v)@, final(v)@, and valid Verus datatype projections.

SYSTEM (CGS self-check): You are a strict Rust requirement/Verus-spec alignment
reviewer. Return strict JSON only.

USER (CGS self-check): Check functional coverage and reject invented sequence
members, dropped return cases, or dropped mutation/frame constraints. Return
is_aligned, missing_constraints, inconsistent_items, refinement_hints, and notes.

SYSTEM (CGS refinement): You refine Verus specs to improve Rust requirement
alignment. Return strict JSON only.
```

```
USER (CGS refinement): Refine the Verus specification from the requirement,
constraints, current specification, and findings. Preserve every correct
functional clause; add or correct missing cases without deleting return,
mutation, or unchanged-region facts. Keep loop annotations in code hints.

SYSTEM (code generation): You are an expert Rust developer writing
Verus-friendly code. Output Rust source code only.

USER (code generation): Implement one complete Rust/Verus file from the
requirement and frozen contract. Include vstd::prelude, one verus! block, and
main outside it. Preserve requires/ensures semantics, use safe Rust, and add
implementation-level invariants or decreases clauses when needed.

SYSTEM (VGCR analysis): You are a verification-guided Rust/Verus repair
planner. Return strict JSON only.

USER (VGCR analysis): Analyze the Verus result and return typed subgoals,
verifier evidence, concrete repair hints, a global strategy, and risk notes.
Treat the Verus contract as frozen.

SYSTEM (VGCR candidate): You are an expert Rust/Verus developer using
prove-as-you-generate repair. Output Rust source code only.

USER (VGCR candidate): Repair the implementation from the plan while keeping
the frozen requires/ensures semantics. Change only body, helper proof code,
and loop annotations; use safe Rust and preserve the verus! wrapper and main.
```

E.4 PYTHON / NAGINI

```
SYSTEM (specification): You are an expert in Python formal verification with
Nagini. Return strict JSON only.

USER (specification): Given the Python requirement and signature hint, return
function_signature, nagini_contract, atomic nagini_clauses,
code_annotation_hints, and notes. Use Requires, Ensures, Acc, Old, Result(),
Implies, list_pred, and dict_pred as appropriate; keep the contract first in
the function body.

SYSTEM (CGS extraction): You are an expert in translating Python requirements
into Nagini verification constraints. Return strict JSON only.

USER (CGS extraction): Extract atomic preconditions, postconditions,
permission_conditions, exception_freedom, and invariants. Preserve boundary
cases and use Old(...) for values read from pre-state containers.

SYSTEM (CGS translation): You are an expert Nagini specification engineer.
Convert Python constraints into a complete Nagini contract. Return strict JSON
only.

USER (CGS translation): Emit atomic Requires/Ensures/Acc clauses. Use Result(),
Implies(condition, conclusion), list_pred/dict_pred, Old(...), and ordinary
Python Boolean expressions. Do not use wildcard, Forall, lambda triggers, or
unsupported helper APIs.

SYSTEM (CGS self-check): You are a strict Python requirement/Nagini-spec
alignment reviewer. Return strict JSON only.

USER (CGS self-check): Check the original requirement as well as extracted
constraints. Return is_aligned, missing_constraints, inconsistent_items,
refinement_hints, and notes; preserve mutation facts, None cases, permissions,
and exception freedom.

SYSTEM (CGS refinement): You refine Nagini specs to improve Python requirement
alignment. Return strict JSON only.

USER (CGS refinement): Refine the Nagini specification using the requirement,
constraints, current specification, and findings. Preserve correct atomic
clauses, mutation and permission facts, boundary cases, and legal Nagini
syntax; keep loop invariants in code hints.

SYSTEM (code generation): You are an expert Python developer writing
Nagini-friendly code. Output Python source code only.

USER (code generation): Implement one complete Python source file from the
requirement and frozen Nagini specification. Keep the target signature and
contract as the first function statements, add no tests or top-level execution,
and use simple Nagini-compatible code and supplied invariants only.

SYSTEM (VGCR analysis): You are a verification-guided Python/Nagini repair
planner. Return strict JSON only.

USER (VGCR analysis): Analyze Nagini diagnostics and produce typed subgoals,
evidence, repair hints, a global strategy, and risk notes. Treat the Nagini
contract as frozen and do not suggest unsupported Fold/Unfold or element APIs.

SYSTEM (VGCR candidate): You are an expert Python/Nagini developer using
prove-as-you-generate repair. Output Python source code only.
```

```
USER (VGCR candidate): Repair the implementation using the plan. Preserve the
target signature and frozen contract semantics; change only body logic and
Invariant annotations. Keep the required imports and use simple Nagini Python.
```