

SVE-Loom: Improving Session-Level Reliability for Natural-Language-to-SVE Code Generation with Bounded Feedback

Haoyuan Gui^{1,2} Hongjiao Sheng^{2,3} Yifan Zhang^{1,2} Huiyuan Li^{2,3,*}

¹University of the Chinese Academy of Sciences

²Institute of Software, Chinese Academy of Sciences

³Key Laboratory of System Software, Institute of Software, Chinese Academy of Sciences
huiyuan@iscas.ac.cn

Abstract

Architecture-specific vector programming remains a difficult frontier for language models: an implementation can satisfy surface syntax yet fail because of a hallucinated intrinsic, an invalid predicate shape, a sizeless-type misuse, or a subtle tail-handling error. This work studies NL2SVE, a function-level code generation problem that maps natural-language specifications and fixed C/C++ signatures to ARM Scalable Vector Extension (SVE) implementations. We introduce SVE-Loom, a model-agnostic generation-and-repair framework that operates within a bounded compile-and-run feedback session. SVE-Loom combines ACLE schema repair, execution feedback, and reference-assisted Regional Semantic Bootstrap when scalar semantics are available. From an initial prompt under a bounded feedback budget, wrapped models using SVE-Loom achieve higher success rates, producing more legal and correct SVE implementations across evaluation benchmarks. We further mitigate the scarcity of public SVE supervision by constructing a NL2SVE corpus from GitHub-mined SVE code, template-generated kernels, and SVE-Loom-guided API-model candidates. This third layer makes the corpus extensible: scalar references or capability gaps can be converted into NL-to-SVE examples, while candidate SVE implementations are retained only after schema checks, compilation, execution testing, speedup measurement, and human audit.

1 Introduction

Large language models have made substantial progress in code generation, from function-level synthesis to competitive programming and feedback-guided refinement (Chen et al., 2021; Austin et al., 2021; Li et al., 2022). Most evaluations, however, still focus on general-purpose scalar programs. Architecture-specific vector code remains harder: the generated program must be

both functionally correct and legal under a concrete instruction-set interface.

ARM SVE makes this challenge practically important. Its vector-length-agnostic (VLA) predication is deployed across supercomputing, cloud, and server-class processors (Stephens et al., 2017; Fujitsu, 2026; ARM Ltd., 2026a; Amazon Web Services, 2026). As scalar C/C++, ARMv8/Neon, and x86-oriented kernels migrate to SVE targets, developers need implementations that exploit scalable-vector semantics rather than merely preserve scalar behavior. High-quality public SVE code remains sparse, limiting supervision for data-driven generation.

Beyond data scarcity, SVE exposes a large legality surface through ACLE intrinsics: operation families, element types, predication forms, argument shapes, pointer categories, immediates, and tail handling are encoded in names and signatures (ARM Ltd., 2025a). Models may recover scalar intent while still producing nonexistent intrinsics, invalid predicates, or incorrect scalar-vector boundaries, consistent with SIMDBench’s finding that SIMD-intrinsic generation remains difficult for LLMs (He et al., 2025).

We study this problem as NL2SVE: natural-language-to-SVE code generation. An NL specification provides a portable semantic interface for migration: it abstracts away artifacts of scalar, Neon, or x86 implementations while preserving the intended computation, boundary conditions, and input-output contract. Given such a specification, a fixed C/C++ signature, and executable tests, the system must synthesize a self-contained SVE implementation. Thus, NL2SVE differs from repository-level software engineering and conventional compiler auto-vectorization: the model must directly instantiate legal and correct SVE idioms.

We make two main contributions. First, we propose SVE-Loom, a model-agnostic framework for bounded-feedback NL2SVE generation and repair.

Within this framework, NameFix and ShapeFix use an ACLE schema to guide repairs toward legal intrinsic names and compatible call shapes. Regional Semantic Bootstrap (RSB) converts a validated scalar reference into an AST-grounded, loop-level representation that identifies vectorizable regions and records dependence evidence. The framework integrates both components with bounded compile-and-run feedback. Evaluated on SIMD-Bench and ARM SIMD Loops, this framework improves session-level reliability over direct generation: starting from an initial prompt and using only a bounded feedback budget, wrapped models produce legal and correct SVE implementations at higher rates. Second, to mitigate the scarcity of public SVE training data, we construct a NL2SVE corpus organized into three provenance layers: GitHub-mined SVE implementations, schema-tracked template-generated kernels, and SVE-Loom-guided candidates proposed by API-hosted code models. This third layer extends the corpus using new scalar references or tasks designed to cover underrepresented SVE capabilities. Candidates generated in either case are retained only after schema validation, compilation, correctness testing, speedup measurement, and human audit. Our code and dataset are available online.¹

2 Background and Related Work

ARM SVE and ACLE intrinsics. SVE is ARM’s scalable vector extension, designed around VLA execution and predicated operations (Stephens et al., 2017). Its C/C++ interface is exposed through ACLE intrinsics, whose names and signatures encode operation families, element types, predication forms, vector/scalar argument shapes, pointer categories, and immediate-operand constraints (ARM Ltd., 2025a). These constraints make SVE a strict target for language-to-code generation: a model can understand the scalar semantics but still produce invalid code because the selected intrinsic, predicate shape, lane type, or tail-handling pattern is illegal. At the same time, API legality is only one part of the problem. Correct SVE generation also requires deciding where vectorization is valid, where computation should remain scalar or lane-local, and where expression rewriting can remove apparent dependencies.

NL interfaces for code generation. NL has become a standard interface for specifying code-

generation tasks, from early work on domain-specific programs and regular expressions to neural decoders with syntax- and AST-aware structure (Raza et al., 2015; Desai et al., 2016; Locascio et al., 2016; Yin and Neubig, 2017; Rabinovich et al., 2017). Prior work has established function-level generation and execution-based evaluation from NL specifications across semantic parsing, program synthesis, and code-generation benchmarks (Iyer et al., 2018; Yin et al., 2018; Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022). These settings, however, largely target scalar, architecture-agnostic programs. SIMD-Bench is closely aligned with NL2SVE because it evaluates explicit SIMD-intrinsic generation rather than architecture-agnostic scalar code (He et al., 2025).

LLM-based code generation and refinement.

Recent code LLM work improves program synthesis through code-specific instruction tuning, explicit planning, multi-agent decomposition, multi-turn reasoning, and execution- or compiler-guided refinement (Luo et al., 2024; Muennighoff et al., 2024; Islam et al., 2024; Zheng et al., 2025a; Chen et al., 2024; Bi et al., 2024; Dou et al., 2024; Gehring et al., 2025). These approaches strengthen code generation through better supervision, decomposition, reasoning, or feedback use. Their targets, however, are usually general-purpose programming, competitive programming, or repository-level tasks. Among these methods, compiler- and execution-guided approaches use diagnostics or runtime outcomes as repair signals for generated code. SVE-Loom follows this bounded-feedback view, but specializes the repair surface to ARM SVE: before invoking general compile-and-run refinement, NameFix and ShapeFix use an ACLE schema derived from `arm_sve.h` to correct hallucinated intrinsic names and invalid call shapes under the fixed target signature. This turns common SVE API failures from open-ended textual repair into schema-constrained correction.

Program-analysis-guided code generation.

Program synthesis has long used sketches and syntax-guided constraints to restrict the search space and expose checkable program structure (Solar-Lezama et al., 2006; Alur et al., 2013). Recent neural code-generation work similarly uses semantic scaffolds, static-analysis facts, and monitor-guided decoding to condition generation on program structure beyond raw text (Zhong et al.,

¹<https://github.com/guihaoyuan/SVE-Loom>

2020; Mukherjee et al., 2021; Agrawal et al., 2023). For vectorization, LLM-Vectorizer and VecTrans use existing code to improve vectorization, either by generating validated vectorized loops or by rewriting code for compiler auto-vectorization (Taneja et al., 2025; Zheng et al., 2025b). Together, these lines suggest that structured program evidence can improve generation beyond raw text. SVE-Loom likewise uses scalar reference code, but converts it into RSB: an AST-grounded bootstrap that exposes vectorization-relevant loop regions, including scheduling loops, data lanes, lane-local scalar computation, reductions, scans, compact writes, expression rewrites, and conservative regions. Thus, RSB is not a generic program sketch; it is a reference-derived semantic scaffold for generating more stable SVE vectorization candidates under fixed C/C++ signatures.

3 SVE-Loom Framework

3.1 Overview

SVE-Loom turns direct NL2SVE generation into bounded-feedback refinement. Figure 1 gives the high-level pipeline. The framework is model-agnostic: it wraps an LLM with the same bounded repair controller at inference time. Its components separate initial generation, ACLE API legality repair, execution feedback, and optional reference-assisted repair.

3.2 SVE ACLE Name/Shape Fix

SVE-Loom uses an ACLE schema extracted from Clang’s `arm_sve.h` to examine SVE intrinsic calls. The schema records legal intrinsic names, argument signatures, and return types. NameFix identifies names absent from the schema and searches for legal replacements using common naming corrections, operation prefixes, type markers, and spelling similarity. For calls with legal names, ShapeFix checks the argument count and compares argument and return types using four categories: predicate, vector, pointer, and scalar.

When either stage detects a violation, SVE-Loom provides schema-based diagnostics for revising the candidate without changing the target signature. This pre-compilation step addresses common name and call-shape errors before compile-and-run repair. Constraints outside its scope, including compile-time immediate requirements, are handled by compiler feedback. Appendix B describes the schema, repair procedure, and representative cases.

Item	Repair-test oracle	Hidden-test oracle
SIMDBench	Official harness, ITERATIONS=1000	ITERATIONS=5000; expanded Small_Arg
ASL sizes	{0, 1, 7, 31, 64}	{0, 1, 2, 3, 4, 7, 8, 15, 16, 17, 31, 32, 33, 63, 64, 65}
Model feedback	Status + diagnostics	None
I/O exposed	No	No
Used during re-pair	Yes	No

Table 1: Repair-test and hidden-test oracle configurations for repaired candidates.

3.3 Compile-and-Run Feedback

After schema-guided repair, SVE-Loom compiles and executes each candidate in the target ARM/SVE environment. The feedback is black-box to the model: compilation exposes only compiler diagnostics, while execution exposes only coarse outcomes such as success, crash, timeout, or logical mismatch. The model never observes concrete test inputs, expected outputs, or the oracle implementation. Thus, compile-and-run feedback serves as a bounded evaluator-in-the-loop signal rather than a source of explicit input-output examples or reference solutions.

Each repair round revises the current candidate using the available feedback without changing the target signature. We allow at most two repair rounds and stop once a candidate compiles successfully and passes all correctness tests at runtime. Performance measurements are used only for evaluation and are not provided to the repair process.

Evaluation oracles. For repair-enabled settings, we assess the final candidates using two correctness oracles. The repair-test oracle uses the normal benchmark configuration and provides only status and diagnostic feedback. The hidden-test oracle uses a stronger configuration to assess post-repair robustness, and its results are not exposed to the model. Table 1 lists the benchmark-specific oracle configurations and feedback visibility.

3.4 Regional Semantic Bootstrap

We introduce Regional Semantic Bootstrap (RSB), which uses the Clang abstract syntax tree (AST) to convert a validated scalar reference into loop-level semantic regions for reference-assisted SVE generation. A scalar reference is considered validated

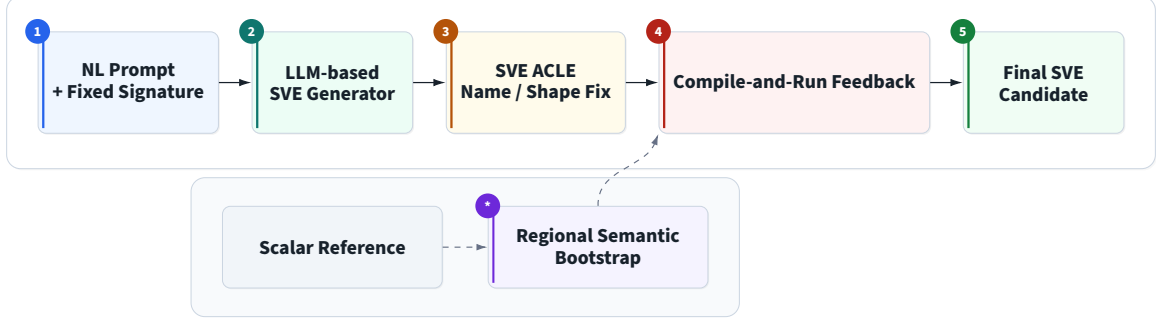


Figure 1: The SVE-Loom pipeline. The main path synthesizes SVE code from an NL specification and fixed signature; bootstrap is invoked only when scalar references are available.

if it compiles and passes the repair-test oracle. In the repair controller, RSB is used only when the initial SVE candidate fails compile-and-run validation and a validated scalar reference is available. The resulting bootstrap is then provided as a repair signal for the failed candidate.

The analysis classifies each loop using its index expressions, scalar updates, control flow, memory accesses, and dependence evidence. Vectorizable regions use `vector_range`, `lane`, and `active` to represent vector iteration and lane activity. Dedicated markers record reductions, prefix scans, compact writes, conflict-prone updates, and closed-form rewrites. Regions with irregular control flow, dynamic containers, string manipulation, or unresolved dependencies remain in ordinary pseudocode.

AST-derived program representation. Clang parses the validated scalar reference into an AST, which RSB converts into a loop tree that preserves loop nesting. Each loop node records its induction variable, iteration bounds, step, and nested body. During this conversion, assignments, branch conditions, returns, calls, and memory accesses are extracted from the loop body. Each memory access is associated with a base object and a normalized index expression. This representation provides the input to the recursive projection and dependence analysis described below.

Recursive semantic projection. Using this representation, RSB recursively processes each loop node and the statements in its body. During this traversal, a context Γ records the induction variables, iteration bounds, and steps of enclosing loops, together with relevant scalar assignments, branch conditions, and memory accesses with normalized index expressions. For each node n , the

projection produces a bootstrap representation b :

$$b = \Phi(n, \Gamma). \quad (1)$$

Consider a loop $L(v, D, B)$, where v is the induction variable, D is its iteration domain, and B is its body. For a data loop, the lane-specific index is

$$v_\ell = \text{vec_base} + \text{lane_id} \times \text{step},$$

where vec_base is the first scalar index in the current vector chunk, lane_id is the lane offset within that chunk, and step is the source-loop stride. When $\text{step} = 1$, this reduces to $\text{vec_base} + \text{lane_id}$.

Let Γ_v and Γ_ℓ denote the context Γ extended with v and v_ℓ , respectively. For loop nodes, the projection is

$$\Phi_{\text{loop}}(L(v, D, B), \Gamma) = \begin{cases} \text{ScalarLoop}(v, D, \Phi(B, \Gamma_v)), & \text{sched}(v), \\ \text{VectorLoop}(v_\ell, D, \Phi(B[v := v_\ell], \Gamma_\ell)), & \text{data}(v), \\ \text{ConservativeLoop}(v, D, \Phi(B, \Gamma_v)), & \text{otherwise.} \end{cases} \quad (2)$$

Here, $B[v := v_\ell]$ denotes the substitution of the lane-specific index v_ℓ for v in the loop body. The three constructors represent scalar, vector, and conservative loop forms.

The predicates $\text{sched}(v)$ and $\text{data}(v)$ determine how RSB projects the loop controlled by the induction variable v . If v organizes the work of a nested loop, such as by determining its bounds or selecting the pointer it processes, then $\text{sched}(v)$ holds. RSB keeps this loop scalar and continues to analyze its nested loops. If v directly indexes independent array or vector elements and the analysis finds no condition that requires sequential or conservative handling, then $\text{data}(v)$ holds. RSB replaces v with the lane-specific index v_ℓ and projects the loop as a vector loop. Otherwise, it uses the

Source	Train	Dev	Total
GitHub-mined SVE	901	103	1,004
Template-generated	958	98	1,056
SVE-Loom-guided	3,462	425	3,887
Total	5,321	626	5,947

Table 2: Source composition and train/development split of the final NL2SVE corpus snapshot.

conservative loop form. This classification is derived from the AST-derived program representation before the bootstrap is emitted.

Dependence evidence. For each loop, RSB records the base object and normalized index of every memory read and write. A read and a write to the same object at different normalized indices indicate a possible dependence between loop iterations. RSB also detects scalar values that are updated inside the loop and later affect a branch or an output. These dependencies normally prevent the loop from being classified as a data loop. Recognized reductions, prefix scans, compact writes, and closed-form rewrites are handled separately.

RSB classifies each loop node independently, so a scalar scheduling loop may contain a nested data loop rendered as a `vector_range` region. When the available evidence is insufficient, the loop retains a conservative scalar representation.

Figure 2 summarizes three representative RSB projections: separating scheduling loops from data loops, retaining lane-local scalar computation inside a data-parallel loop, and rewriting an affine recurrence as a closed-form vector range. Appendix C provides detailed analyses and successful repair cases.

4 NL2SVE Corpus Construction

4.1 Corpus Construction and Provenance

We construct the NL2SVE corpus from three provenance layers, summarized in Table 2.

First, we mine public SVE and ACLE code from GitHub repositories (OpenMathLib, 2025; ARM Ltd., 2025b; LandiSimone, 2025). These examples expose the model to naturally occurring SVE idioms, including predicated loops, ACLE intrinsic usage, VLA execution, and memory-access patterns used in real code.

Second, we build template-generated kernels from schema-tracked kernel families. These templates provide controlled coverage of common vectorization structures, including contiguous

map, zip, and triad operations; integer reductions; masked if-conversion; mixed-type conversion and scaling; and simple stencil patterns. Template generation systematically varies data types, loop bounds, predicates, memory layouts, and arithmetic operators while preserving a known scalar reference.

Third, we use SVE-Loom to guide proprietary API-hosted code models in generating additional SVE candidates (OpenAI, 2026; Moonshot AI, 2026). This subset improves coverage of SVE capabilities that are sparse or unevenly distributed in public code, including predicate-controlled memory access, mixed-width arithmetic, reductions, scans, scalar-control regions, loop-carried dependencies, gather/scatter or strided access, array-of-structures (AoS) and interleaved layouts, numerical edge cases, and ACLE call-shape constraints.

For the SVE-Loom-guided layer, we first obtain a scalar reference and derive an NL prompt from its semantics. API-hosted models then generate candidate SVE implementations under SVE-Loom guidance. We retain only candidates that pass schema validation and compilation, match the scalar reference in correctness tests, achieve a measured speedup above 1.0 under the available performance oracle, and pass manual review. The models therefore serve as candidate generators, while executable validation and human review determine corpus admission. The same pipeline can extend the corpus from new scalar references or underrepresented SVE capabilities.

Before training, each example is normalized into a function-level prompt-response pair with an NL specification, required includes, and a fixed target signature. Responses are reduced to the target SVE implementation, excluding unrelated wrappers, main functions, helper routines, and extra top-level APIs. We preserve the target signature and apply admission checks for signature consistency, SVE API legality, compilation, and execution correctness. Appendix F reports capability-target counts for the template-generated and SVE-Loom-guided layers.

4.2 Benchmark Decontamination Statement

To prevent benchmark contamination, our training and development corpus was constructed independently of the SIMDBench and ARM SIMD Loops evaluation tasks; these benchmark tasks were never used for training. After corpus construction, we performed a multi-level corpus-vs-benchmark non-

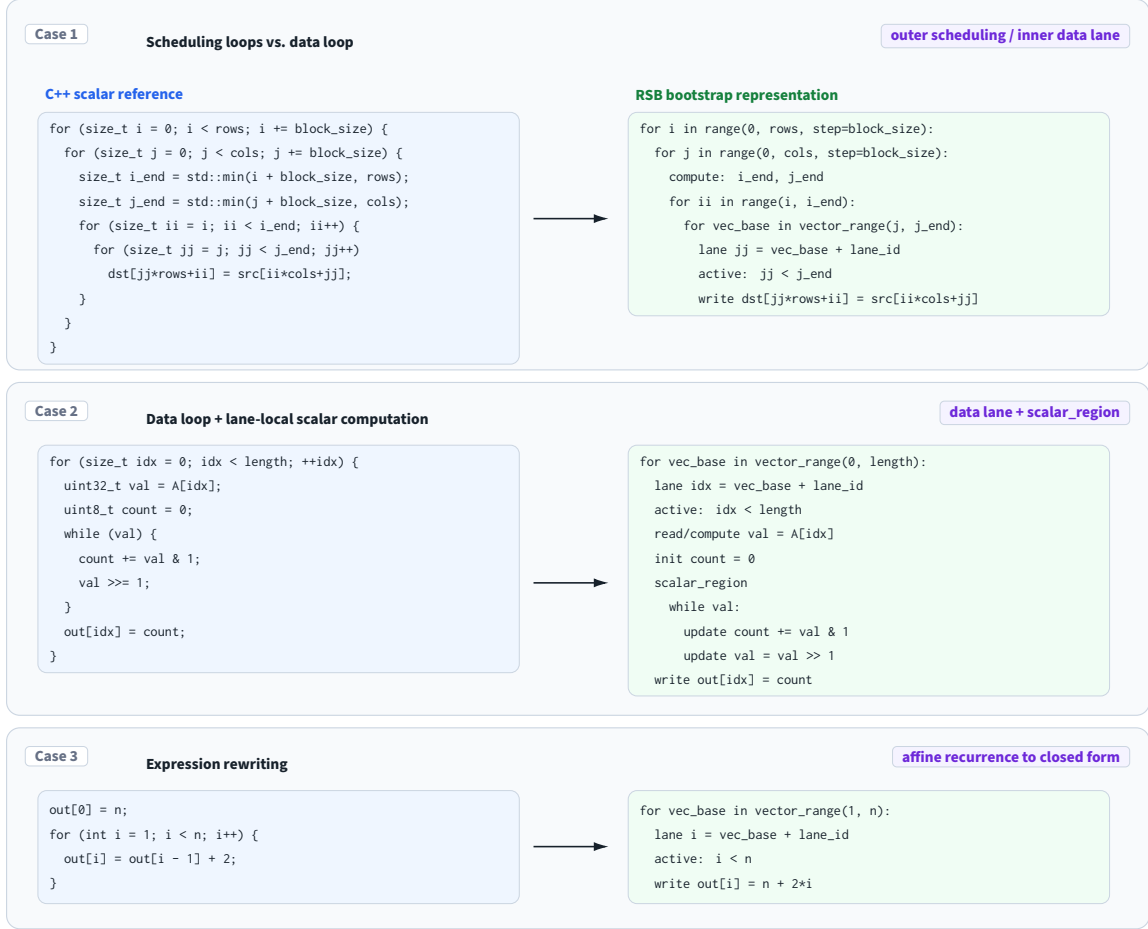


Figure 2: Examples of Regional Semantic Bootstrap projections.

overlap audit over identifiers, prompts, signatures, code, text fingerprints, AST structure, and abstract semantic patterns, using standard near-duplicate and code-parsing methods (Python Software Foundation, 2026; Broder, 1997; Broder et al., 1998; Charikar, 2002; Manku et al., 2007; tree-sitter contributors, 2026b,a). The audit found no exact, near-duplicate, AST-level, or retained semantic high-risk overlap; Appendix G reports the full comparison units, thresholds, and final status.

5 Experimental Setup

5.1 Benchmarks and Protocol

We evaluate SVE-Loom on two function-level SVE generation benchmarks. SIMDBench provides native SVE tasks with NL specifications, fixed C/C++ signatures, scalar references, correctness tests, and performance tests, requiring models to generate explicit architecture-specific SIMD intrinsics rather than scalar C/C++ (He et al., 2025). ARM SIMD Loops provides a complementary ARM-oriented SVE/SVE2 subset of 54 loop ker-

nels with C references, ACLE-intrinsic variants, build targets, and correctness validation (ARM Ltd., 2026b). For ARM SIMD Loops, we manually write SIMDBench-style NL prompts under the fixed target signatures. Both benchmarks are evaluated by compiling generated SVE code and running correctness tests. To mirror the SVE-Loom-guided corpus construction process, the RSB used in experiments is derived from Codex-5.2-generated scalar references that compile and pass the repair-test oracle, rather than benchmark-provided scalar references.

For each task, we sample five candidates. In feedback-enabled settings, each candidate may undergo at most two repair rounds, with early stopping once it compiles and passes all correctness tests. We report Pass@1 and Pass@5 for both benchmarks. Following SIMDBench (He et al., 2025), we compute Pass@ k over a task set $\mathcal{T}$ as:

$$\text{Pass}@k = \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \left(1 - \frac{\binom{n_i - c_i}{k}}{\binom{n_i}{k}} \right). \quad (3)$$

where n_i is the number of sampled candidates

for task i , and c_i is the number of candidates that compile and pass all correctness tests; we report $k = 1$ and $k = 5$. Across all reported metrics, a generated candidate is counted as passing only if it passes the corresponding correctness tests while using Arm SVE/ACLE vector intrinsics. Functionally correct scalar fallbacks are excluded from Pass@ k , hidden-test, repair-path, and speedup statistics. Speedup is used only for evaluation; repair is guided by correctness-oriented compile-and-run feedback rather than performance measurements.

5.2 Hardware and Inference Settings

Appendix D summarizes the hardware and inference settings used in our experiments. The hosted API baselines use Codex-5.2 (OpenAI, 2026), Kimi-K2.5 (Moonshot AI, 2026), Gemini-2.5 Pro (Google, 2026), GLM-4.7 (Z.AI, 2026), and DeepSeek-R1 (DeepSeek-AI, 2026). For local fine-tuning, we use the smaller distilled model DeepSeek-R1-Distill-Qwen-32B (DeepSeek-AI, 2025). Appendix E reports the local-model training setup and prompt details.

6 Experiments

We organize the experiments around four questions. First, we assess NL-only NL2SVE direct generation under fixed signatures. Second, we wrap the two strongest direct-generation baselines with SVE-Loom to evaluate session-level reliability in the bounded-feedback setting. Third, we isolate component-level effects with a controlled local model, since hosted proprietary APIs may be affected by model-version or serving-system drift; for these hosted models, we instead report repair-trigger and pathway statistics. Fourth, we test whether SVE-Loom yields more correctness-passing implementations with positive speedup.

6.1 NL-only Direct Baselines

We first evaluate representative code models in a strict NL-only direct-generation setting. Table 3 reports the corresponding SIMDBench and ARM SIMD Loops results. This setting measures NL2SVE difficulty without interactive assistance.

6.2 Inference-Time Repair on Strong Backbones

We next evaluate how SVE-Loom changes session-level success rates for strong backbones in the bounded-feedback setting. This experiment uses

Model	SIMDBench		ASL	
	Pass@1 (%)	Pass@5 (%)	Pass@1 (%)	Pass@5 (%)
Kimi-K2.5	51.62	83.82	39.63	66.67
Codex-5.2 Medium	51.18	81.62	43.33	66.67
Gemini-2.5 Pro	37.50	69.12	26.30	51.85
GLM-4.7	27.35	64.71	12.96	35.19
DeepSeek-R1 (official API)	13.53	25.74	11.85	24.07

Table 3: NL-only direct-generation baselines on SIMDBench and ARM SIMD Loops (ASL).

Benchmark	Model	Repair-test		Hidden-test	
		+SVE-Loom	P@1 P@5	P@1 P@5	P@1 P@5
SIMDBench	Codex-5.2-M	96.47	99.26	95.00	97.79
SIMDBench	Kimi-K2.5	90.29	99.26	88.24	98.53
ASL	Codex-5.2-M	86.67	94.44	85.93	94.44
ASL	Kimi-K2.5	81.85	88.89	81.85	88.89

Table 4: Full-controller SVE-Loom results for strong direct-generation backbones under a bounded repair session. P@1/P@5 denote success over bounded repair sessions, not one-shot direct generation.

the two strongest NL-only direct baselines in Table 3. The hosted backbones are not fine-tuned. SVE-Loom applies all of its inference-time components. Table 4 reports post-repair Pass@1 and Pass@5 on the repair-test and hidden-test oracles. To compare direct generation and bounded repair under the same evaluation criterion, we also reevaluate the unchanged direct candidates with the hidden-test oracle. The SIMDBench direct scores differ slightly from Table 3, which uses the normal benchmark configuration.

Under the hidden-test oracle, Codex-5.2 changes from 50.00%/80.15% to 95.00%/97.79% Pass@1/Pass@5 on SIMDBench, while Kimi-K2.5 changes from 50.44%/81.62% to 88.24%/98.53%. On ARM SIMD Loops, the corresponding results change from 43.33%/66.67% to 85.93%/94.44% for Codex-5.2 and from 39.63%/66.67% to 81.85%/88.89% for Kimi-K2.5. These comparisons use the same hidden-test oracle and the same nominal five-candidate sampling budget. Only the SVE-Loom settings apply bounded repair, so the direct-to-post-repair differences characterize session-level reliability without conflating the comparison with a change in the evaluation oracle.

Appendix H provides diagnostic repair-path counts: many successful repairs activate API-surface markers, and harder candidates often activate the RSB path. Because these triggers are not mutually exclusive, we treat the pathway analysis as diagnostic rather than causal.

NameFix/ ShapeFix	Compile-and-run feedback	Bootstrap	Pass@1 (%)	Pass@5 (%)
No	No	None	29.41	59.56
Yes	No	None	36.03	63.97
No	Yes	None	40.88	69.85
Yes	Yes	None	50.44	76.47
Yes	Yes	Scalar code	51.18	80.15
Yes	Yes	RSB	57.65	86.03

Table 5: Component ablation on SIMDBench using the same fine-tuned local backbone.

6.3 Component Ablation

After fine-tuning a local backbone on the constructed NL2SVE corpus, we evaluate the inference-time components of SVE-Loom on SIMDBench. All rows in Table 5 use the same fine-tuned checkpoint, and all feedback-enabled settings use the same repair budget. In the last two rows, NameFix/ShapeFix and feedback remain fixed while the bootstrap input changes. These comparisons measure component-level effects for the local backbone; the gains may differ for other models.

The untuned local backbone produces no passing SVE implementations. After fine-tuning, direct generation without inference-time repair reaches 29.41% Pass@1 and 59.56% Pass@5, providing the common baseline for the ablation. NameFix/ShapeFix alone reaches 36.03% and 63.97%, while compile-and-run feedback alone reaches 40.88% and 69.85%. Combining both components raises Pass@1 to 50.44% and Pass@5 to 76.47%. Relative to feedback alone, NameFix/ShapeFix adds 9.56 and 6.62 percentage points under the same feedback budget.

With NameFix/ShapeFix and feedback fixed, scalar-code bootstrap reaches 51.18% Pass@1 and 80.15% Pass@5. RSB reaches 57.65% and 86.03%, outperforming scalar code by 6.47 and 5.88 percentage points. For this backbone, the structured RSB representation therefore provides more effective reference assistance than supplying the scalar code directly.

6.4 Speedup Evaluation

We evaluate speed on SIMDBench only after correctness: a candidate enters runtime measurement only if it passes the SVE correctness tests. The repair loop receives compile-and-run feedback for functional correctness and ACLE legality, but it does not observe runtime measurements. Thus, speedup is used for evaluation rather than as a repair signal. Using the SIMDBench per-

Model	Correct+> 1×	Task-best faster	Sample p90
Codex-5.2	25.1%	43.4%	11.837
Codex-5.2 + SVE-Loom	51.3%	61.8%	13.334
Kimi-K2.5	27.3%	50.0%	17.125
Kimi-K2.5 + SVE-Loom	47.8%	61.1%	20.709

Table 6: SIMDBench speedup evaluation under -O3.

formance harness, generated implementations and scalar baselines are compiled with `-std=c++17 -march=armv8-a+sve -O3` on the AWS Graviton3-class ARM/SVE machine reported in Appendix D; the observed SVE vector length is 256 bits, and each implementation is timed with 5 repetitions.

Table 6 reports sample-level correct-and-faster rate, task-level faster coverage, and p90 speedup among correct samples. Rows with SVE-Loom use the same bounded repair protocol as Table 4.

Under -O3, SVE-Loom produces more correct SVE implementations that are also faster than the scalar baseline. Correct-and-faster rates rise from 25.1% to 51.3% for Codex-5.2 and from 27.3% to 47.8% for Kimi-K2.5. Appendix I further compares only samples and tasks passed by both the direct and SVE-Loom-wrapped systems, showing that aggregate speedup-positive gains mainly come from converting previously failing generations into correct, sometimes faster implementations.

7 Conclusion

This work studied NL2SVE, a function-level language-to-code problem for synthesizing ARM SVE implementations from NL specifications and fixed C/C++ signatures. SVE-Loom is a model-agnostic generation-and-repair framework that combines ACLE schema repair, bounded compile-and-run feedback, and reference-assisted Regional Semantic Bootstrap when scalar semantics are available. To mitigate the scarcity of public SVE supervision, we also constructed a NL2SVE corpus from GitHub-mined SVE code, template-generated kernels, and SVE-Loom-guided API-model candidates, with this layer providing a reusable route for extending coverage through validated new examples.

Across SIMDBench and ARM SIMD Loops, SVE-Loom improves session-level SVE generation reliability in the bounded-feedback setting. Wrapped models achieve higher rates of legal and correct SVE implementations, and SIMDBench speedup measurements show that many repaired outputs are speedup-positive. Hidden-test checks

further indicate that repaired candidates often remain robust beyond the repair-time oracle.

Limitations

This work studies function-level ARM SVE generation on SIMDBench and a 54-task ARM SIMD Loops subset; it does not address repository-level integration, build systems, multi-file transformations, or larger application-specific HPC kernels. Feedback-enabled results are bounded interactive repair outcomes because the correctness harness is queried during repair, and RSB assumes a validated scalar reference or AST-derived bootstrap. Speedups may vary with hardware, compiler, flags, and harness, while hosted-model results may drift with model versions and serving systems. Generated low-level SVE implementations should not be used outside the evaluated function-level setting without additional validation, since incorrect behavior may arise and correctness or performance may depend on hardware, compiler settings, and benchmark coverage.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 12471348).

References

- Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-guided decoding of code LMs with static analysis of repository context. In *Advances in Neural Information Processing Systems*, volume 36, pages 32270–32298.
- Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Sethia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design*, pages 1–8.
- Amazon Web Services. 2026. AWS Graviton: C/c++ optimization guide. [AWS Graviton C/C++ optimization guide](#). Accessed 2026.
- ARM Ltd. 2025a. ARM C Language Extensions. [ARM C Language Extensions repository](#). Accessed during paper drafting.
- ARM Ltd. 2025b. ARM Optimized Routines. [ARM Optimized Routines repository](#). Accessed 29 December 2025.
- ARM Ltd. 2026a. ARM Neoverse V1. [ARM Neoverse V1 product page](#). Accessed 2026.
- ARM Ltd. 2026b. SIMD Loops. [ARM Learning Paths: SIMD Loops](#). Accessed 2026.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Zhangqian Bi, Yao Wan, Zheng Wang, Hongyu Zhang, Batu Guan, Fangxin Lu, Zili Zhang, Yulei Sui, Hai Jin, and Xuanhua Shi. 2024. [Iterative refinement of project-level code context for precise code generation with compiler feedback](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2336–2353.
- Andrei Z. Broder. 1997. [On the resemblance and containment of documents](#). In *Proceedings of Compression and Complexity of Sequences*, pages 21–29.
- Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 1998. [Min-wise independent permutations](#). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 327–336.
- Moses S. Charikar. 2002. [Similarity estimation techniques from rounding algorithms](#). In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 380–388.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. 2024. [Teaching large language models to self-debug](#). In *International Conference on Learning Representations*, pages 8746–8825.
- DeepSeek-AI. 2025. DeepSeek-R1-Distill-Qwen-32B. [Hugging Face model page](#). Accessed 5 December 2025.
- DeepSeek-AI. 2026. Reasoning Model (deepseek-reasoner). [DeepSeek API documentation](#). Accessed 4 January 2026.
- Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh Ramayanam, and Subhajit Roy. 2016. [Program synthesis using natural language](#). In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356.

- Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. 2024. [StepCoder: Improving code generation with reinforcement learning from compiler feedback](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4571–4585.
- Fujitsu. 2026. [Fujitsu A64FX processor. Fujitsu A64FX processor page](#). Accessed 2026.
- Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. 2025. [RLEF: Grounding code LLMs in execution feedback with reinforcement learning](#). In *Forty-second International Conference on Machine Learning*.
- Google. 2026. Gemini 2.5 Pro. [Google AI for Developers documentation](#). Accessed 4 January 2026.
- Yibo He, Shuoran Zhao, Jiaming Huang, Yingjie Fu, Hao Yu, Cunjian Huang, and Tao Xie. 2025. [SimdBench: Benchmarking large language models for SIMD-intrinsic code generation](#). *Preprint*, arXiv:2507.15224.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. [MapCoder: Multi-agent code generation for competitive problem solving](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944.
- Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. [Mapping language to code in programmatic context](#). In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652.
- LandiSimone. 2025. [Vectorization-of-BAN-library. Vectorization-of-BAN-library repository](#). Accessed 29 December 2025.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. [Competition-level code generation with AlphaCode](#). *Science*, 378(6624):1092–1097.
- Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. [Neural generation of regular expressions from natural language with minimal domain knowledge](#). In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1918–1923.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2024. [WizardCoder: Empowering code large language models with Evol-Instruct](#). In *International Conference on Learning Representations*, pages 27168–27188.
- Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. [Detecting near-duplicates for web crawling](#). In *Proceedings of the 16th International Conference on World Wide Web*, pages 141–150.
- Moonshot AI. 2026. Kimi K2.5. [Kimi official model page](#). Accessed 2 February 2026.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. 2024. [OctoPack: Instruction tuning code large language models](#). In *International Conference on Learning Representations*, pages 7604–7623.
- Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas Reps, Swarat Chaudhuri, and Chris Jermaine. 2021. Neural program generation modulo static analysis. In *Advances in Neural Information Processing Systems*, volume 34, pages 18984–18996.
- OpenAI. 2026. GPT-5.2-Codex Model. [OpenAI API documentation](#). Accessed 4 January 2026.
- OpenMathLib. 2025. OpenBLAS. [OpenBLAS repository](#). Accessed 29 December 2025.
- Python Software Foundation. 2026. [difflib: Helpers for computing deltas. Python documentation](#). Accessed 2026.
- Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. [Abstract syntax networks for code generation and semantic parsing](#). In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149.
- Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. 2015. Compositional program synthesis from natural language and examples. In *Proceedings of IJCAI*, pages 792–800.
- Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. [Combinatorial sketching for finite programs](#). In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415.
- Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico,

and Paul Walker. 2017. [The ARM scalable vector extension](#). *IEEE Micro*, 37(2):26–39.

Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. [LLM-Vectorizer: LLM-based verified loop vectorizer](#). In *2025 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 137–149.

tree-sitter contributors. 2026a. [tree-sitter-cpp: C++ grammar for tree-sitter](#). [tree-sitter-cpp repository](#). Accessed 2026.

tree-sitter contributors. 2026b. [tree-sitter documentation](#). [tree-sitter official documentation](#). Accessed 2026.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. [Learning to mine aligned code and natural language pairs from stack overflow](#). In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 476–486.

Pengcheng Yin and Graham Neubig. 2017. [A syntactic neural model for general-purpose code generation](#). In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450.

Z.AI. 2026. [GLM-4.7. Z.AI developer documentation](#). Accessed 4 January 2026.

Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. 2025a. [What makes large language models reason in \(multi-turn\) code generation?](#) In *International Conference on Learning Representations*, pages 40144–40181.

Zhongchun Zheng, Kan Wu, Long Cheng, Lu Li, Rodrigo C. O. Rocha, Tianyi Liu, Wei Wei, Jianjiang Zeng, Xianwei Zhang, and Yaoqing Gao. 2025b. [VecTrans: Enhancing compiler auto-vectorization through LLM-assisted code transformations](#). *arXiv preprint arXiv:2503.19449*.

Ruiqi Zhong, Mitchell Stern, and Dan Klein. 2020. [Semantic scaffolds for pseudocode-to-code generation](#). In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2283–2295.

A Artifact Licenses and Terms

We use third-party repositories, benchmarks, models, and APIs as research artifacts under their publicly stated licenses or access terms; released derived artifacts, if any, will be distributed with accompanying license and attribution information.

B ACLE Whitelist Schema and API Repair Details

SVE-Loom represents the SVE API surface using a structured whitelist derived from Clang’s `arm_sve.h`. The source header contains function declarations, overload annotations, builtin aliases, type definitions, and extension guards. We normalize the relevant declarations into three queryable structures. Let $\mathcal{N}$ denote the set of legal intrinsic names. For each $f \in \mathcal{N}$, let $\mathcal{S}(f)$ denote the set of recorded argument-type signatures and let $\mathcal{R}(f)$ denote the set of recorded return types. NameFix queries $\mathcal{N}$, whereas ShapeFix uses $\mathcal{S}(f)$ and $\mathcal{R}(f)$.

Schema organization. At runtime, SVE-Loom stores $\mathcal{N}$ as a set for membership checks and as an ordered list for candidate lookup. It stores $\mathcal{S}$ and $\mathcal{R}$ as tables indexed by intrinsic name. For example, $\mathcal{S}(\text{svld1_u8})$ contains the argument signature `[svbool_t, uint8_t const*]`, while $\mathcal{R}(\text{svld1_u8})$ contains the return type `svuint8_t`. Overloaded intrinsics have multiple signatures under the same name.

NameFix. For a candidate program P , let $\text{Call}_{\text{SVE}}(P)$ denote the names of SVE-like calls in P . NameFix identifies invalid intrinsic names as

$$\mathcal{I}(P) = \text{Call}_{\text{SVE}}(P) \setminus \mathcal{N}.$$

For each invalid name in $\mathcal{I}(P)$, NameFix searches $\mathcal{N}$ for a legal replacement. It first checks common naming errors, such as a missing `sv` prefix or an unnecessary `_x`, `_z`, or `_m` suffix. A suffix is removed only when the resulting name belongs to $\mathcal{N}$. NameFix also searches legal names that share the same operation prefix, namely the part before the first underscore. Type markers in the invalid name, such as `u8`, `s32`, or `f32`, further narrow the search. If the invalid name has no type marker, NameFix uses type markers found elsewhere in P .

Names with similar spellings are considered when these checks do not identify a suitable replacement. Such a name must have a similarity score of at least 0.92, match the operation in the invalid name, and preserve any explicit type marker. If no legal replacement is found, the repair prompt requires the call to be rewritten using legal ACLE intrinsics or scalar C/C++ code. Repairs that introduce new invalid names or placeholder text are rejected, and smaller changes near the original call are preferred.

ShapeFix. ShapeFix examines calls whose names already belong to $\mathcal{N}$. For an intrinsic f , it retrieves $\mathcal{S}(f)$ and compares the number of supplied arguments with the recorded signatures. If no signature has the same number of arguments, ShapeFix reports the observed argument count and the valid signatures for f .

The supplied arguments and the argument types recorded in $\mathcal{S}(f)$ are grouped into four categories: predicate, vector, pointer, and scalar. The category of each supplied argument is inferred from its expression form and any associated parameter or local-variable declaration. These categories are then compared with each signature in $\mathcal{S}(f)$. If no signature matches at every argument position, a type mismatch is reported.

The return-type check maps the return types in $\mathcal{R}(f)$ to the same four categories and compares them with the category of the variable receiving the call result. A mismatch is reported if none of the resulting categories matches that of the variable.

Repair scope. The whitelist checks intrinsic existence, arity, and coarse argument and return categories. It does not implement the full C/C++ type system or establish functional correctness, memory safety, alias safety, or performance. Compile-time immediate constraints are handled by the later compiler-diagnostic path. Executable tests assess functional behavior, and performance is measured only after correctness validation.

Successful repair examples. Figure 3 shows one repair from each stage. Both occur before the later RSB semantic-repair stage. Their recorded paths pass through `name_fix` and `shape_fix`, respectively, between code generation and final completion.

In the NameFix case, the generated program calls the nonexistent intrinsic `svptest_all`. Because this name is absent from $\mathcal{N}$, NameFix requires a rewrite using legal predicate operations. The repaired code uses `svnot_z` to form the complement predicate and `svptest_any` to detect a violating active lane.

In the ShapeFix case, `svst2_s32` belongs to $\mathcal{N}$, but the generated call passes two vectors as separate arguments. The signature in $\mathcal{S}(\text{svst2_s32})$ instead requires one `svint32x2_t` tuple. The repaired code constructs the tuple with `svcreate2_s32` and passes it as the final argument.

C RSB Projection and Repair Examples

This section expands the three projections summarized in Figure 2 and provides representative RSB-path repair cases.

Case 1: Scheduling loops vs. data loop. The tiled transpose example separates the loops that select a tile from the loop mapped to vector lanes. The loops over i and j determine the tile boundaries,

$$\begin{aligned} i_{\text{end}} &= \min(i + \text{block_size}, \text{rows}), \\ j_{\text{end}} &= \min(j + \text{block_size}, \text{cols}). \end{aligned} \quad (4)$$

RSB keeps these tile loops, together with the row loop over ii , in scalar form and then analyzes the innermost jj loop. For a fixed ii , the source and destination indices are

$$\begin{aligned} \text{src_index} &= ii \cdot \text{cols} + jj, \\ \text{dst_index} &= jj \cdot \text{rows} + ii. \end{aligned} \quad (5)$$

Different values of jj therefore refer to distinct source and destination elements. RSB maps jj to vector lanes while preserving the surrounding tiled loop structure.

Case 2: Data loop with lane-local scalar computation. The popcount example contains an independent outer loop even though each iteration includes its own scalar while loop. The outer variable `idx` selects one input element `A[idx]` and its corresponding output element `out[idx]`. The inner loop processes `val`, initialized from `A[idx]`, and updates only `val` and `count` for that element. RSB therefore maps `idx` to vector lanes and retains the inner loop as a lane-local `scalar_region`. This representation makes clear that the inner computation does not couple different `idx` iterations.

Case 3: Expression rewriting. The affine recurrence example illustrates a closed-form rewrite. The scalar loop

$$\text{out}[i] = \text{out}[i - 1] + 2, \quad \text{out}[0] = n,$$

contains a loop-carried dependence through `out[i - 1]`. For this recurrence, RSB derives the closed form

$$\text{out}[i] = n + 2i.$$

After the rewrite, each `out[i]` depends only on n and i . RSB can therefore represent the range $1 \leq i < n$ as a `vector_range` rather than preserving the sequential recurrence.

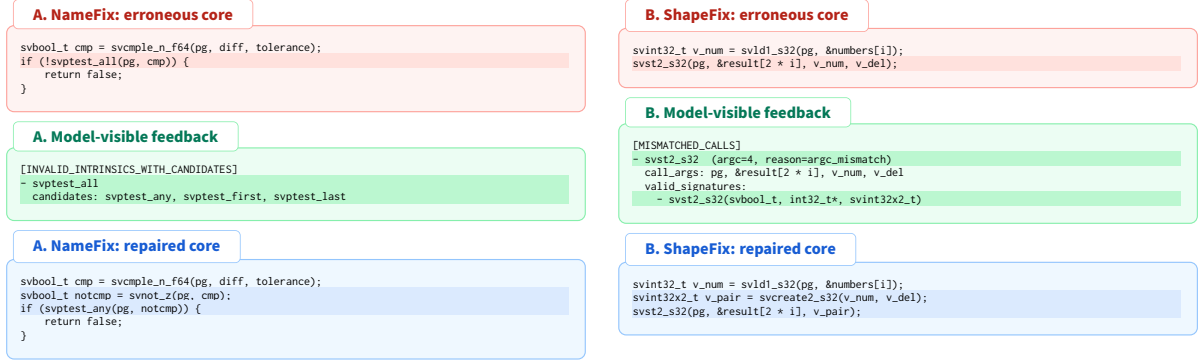


Figure 3: NameFix and ShapeFix repair cases. The left case replaces a nonexistent predicate-test intrinsic with legal predicate operations. The right case constructs the tuple required by `svst2_s32`. Highlighted rows show the invalid call, the schema feedback, and the repair.

Successful RSB repair examples. Figure 4 shows two successful RSB-path repairs. For each case, the figure presents the erroneous candidate, the bootstrap visible to the model, and the repaired implementation.

In the compact-write case, `compact_write` specifies that values from lanes satisfying the predicate must be packed into consecutive output positions starting at the current output position index. The failed candidate instead used a predicated store at the original lane positions, leaving gaps whenever some lanes did not satisfy the predicate. The repaired implementation compacts the selected values with `svcompact_f32` and stores them contiguously with `svst1_f32`.

In the masked conditional-move case, the where `mask[i]` guard inside the `vector_range` region specifies that `dst[i]` is written only when `mask[i]` is true. The failed candidate wrote `src[i]` to every active lane and then set the false-mask positions to zero, although those positions should remain unchanged. The repaired implementation constructs an SVE predicate from `mask[i]` and uses it to guard the store.

D Hardware and Inference Settings

Table 7 summarizes the system and inference settings used in our experiments.

E Local Model Training and Prompt Details

Table 8 gives the training configuration for the local SVE-specialized model used in the component-level ablation. We use LoRA/PEFT rather than full-parameter fine-tuning.

Item	Setting
<i>System</i>	
OS	Ubuntu 22.04.3 LTS
CPU	AWS Graviton3 class
GPU	4 × NVIDIA H20
GPU memory	96 GB per GPU
SVE vector length	256 bits
Local training budget	~3 wall-clock hours; ~12 H20 GPU-hours
Local inference/eval budget	~6 wall-clock hours on the same system
<i>Inference</i>	
Codex-5.2	reasoning_effort=medium
Kimi-K2.5	temperature=1.0, top_p=0.95; thinking="enabled"
Gemini-2.5 Pro	temperature=0.8, top_p=0.95; reasoning_effort=medium
GLM-4.7	temperature=0.8, top_p=0.95, do_sample=true; thinking="enabled"
DeepSeek-R1	temperature=0.8, top_p=0.95
DS-R1-Distill-32B	temperature=0.8, top_p=0.95
<i>Hosted API token limits</i>	
Code generation	16384
NameFix	8192
ShapeFix	4096
Compile-feedback	8192
Serial / bootstrap	4096

Table 7: Hardware and inference settings used in experiments. DS denotes DeepSeek.

Training and inference prompt format. Training and inference use the same chat-message prompt skeleton and tokenizer chat template, rather than a hand-written ### Instruction template. Each prompt contains system and user messages. During training, the assistant message contains the complete target C/C++ implementation, and the loss is applied only to assistant output tokens; during inference, this assistant message is omitted and generated by the model. The system-level instruc-

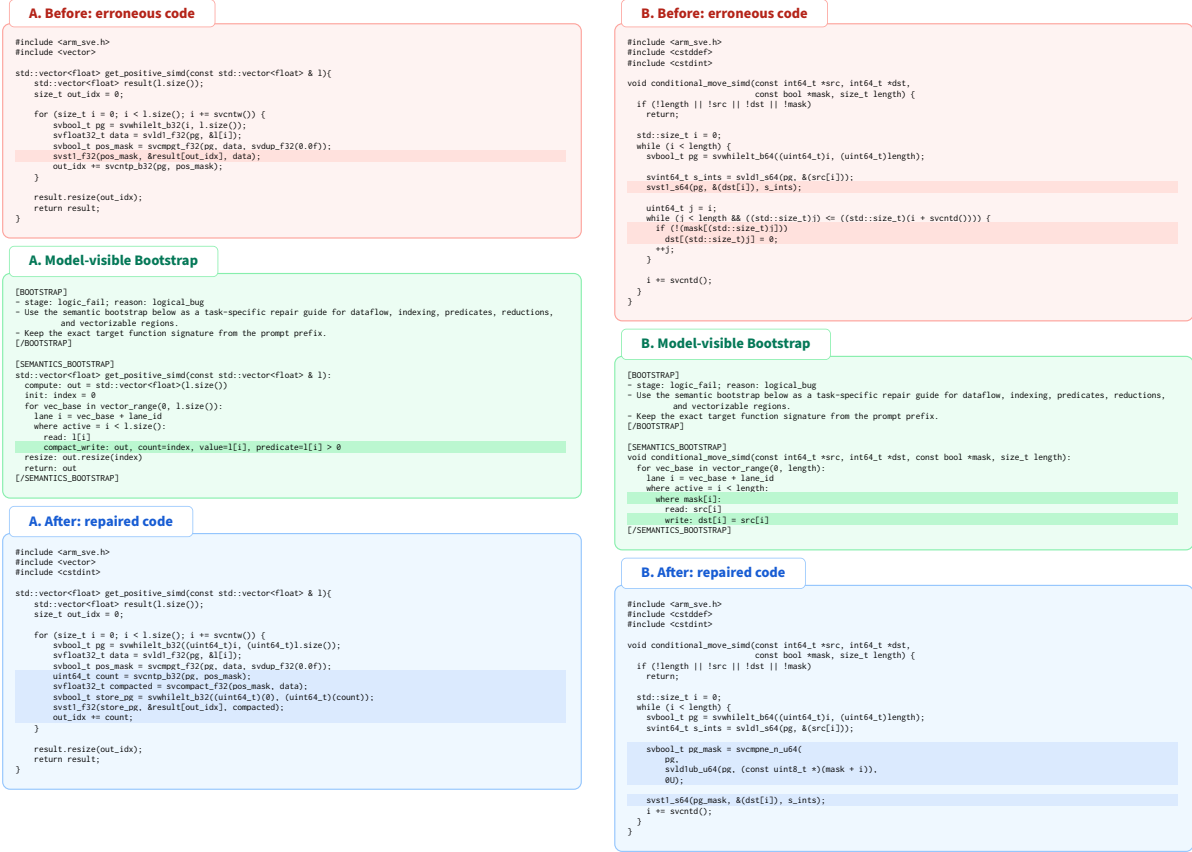


Figure 4: Two RSB repair cases. The left case repairs compact-write semantics; the right case repairs masked conditional-move semantics. Highlighted rows mark the erroneous behavior, the relevant bootstrap semantics, and the corresponding repair.

tion is summarized as:

You are an Arm SVE code generator. Implement the target function from the provided NL task description. Preserve the exact target function signature. Output only the final full C/C++ implementation using Arm SVE where appropriate.

F Corpus Capability Distribution

For corpus transparency, Table 9 reports the capability targets used to characterize the template-generated and SVE-Loom-guided corpus layers. These tags are non-exclusive: a single example may target multiple SVE capabilities, so column totals need not equal the number of rows in each layer. Counts are measured on the final post-high-risk-holdout snapshot, where the template-generated layer contains 1,056 examples and the SVE-Loom-guided layer contains 3,887 examples.

G Decontamination Audit Details

Table 10 summarizes the audit scope. We compared all corpus examples against the held-out benchmark tasks pairwise over normalized prompts, fixed signatures, function entry points, code bodies, tree-sitter AST representations, and abstract semantic signatures.

Table 11 gives the flagging criteria used in the audit. All flagged cases were inspected before finalizing the corpus.

H Repair Pathway Analysis

We analyze repair-path diagnostics for the strong-backbone settings in Table 4. The counts below are diagnostic rather than mutually exclusive: a candidate may trigger multiple repair markers before reaching a final repaired output.

Table 12 shows that repair is not limited to rare corner cases. For Codex-5.2 on SIMDBench, 209 of 680 samples activate at least one repair marker, covering 82 of 136 tasks; 189 of these samples pass after repair. For Kimi-K2.5 on SIMDBench,

Item	Setting
Base model	DeepSeek-R1-Distill-Qwen-32B
Fine-tuning method	LoRA / PEFT
LoRA rank / alpha / dropout	$r = 32$, $\alpha = 32$, dropout = 0.1
LoRA target modules	q_proj, k_proj, v_proj, o_proj, up_proj, gate_proj, down_proj
Epochs	3
Per-device batch size	1
GPUs / gradient accumulation	4 GPUs, accumulation 4
Effective batch size	$1 \times 4 \times 4 = 16$
Learning rate	2×10^{-4}
Scheduler	cosine
Warmup ratio	0.03
Weight decay	0.0
Maximum sequence length	4096
Precision	bf16
Gradient checkpointing	on
Optimizer	adamw_torch
Seed	42

Table 8: Local fine-tuning configuration for the SVE-specialized DeepSeek-R1-Distill-Qwen-32B model.

Capability target	Template-generated	SVE-Loom-guided	Examples
Predicate-controlled memory access	171	505	Predicated load/store, masked update, tail predicates
VLA execution	293	274	Tail handling, unknown trip count, active-lane masking
Mixed-width arithmetic	422	1,107	Widen/narrow conversion, signed/unsigned conversion, scale-and-convert
Reductions and scans	384	1,322	Sum, min/max, predicate reduction, prefix-like dependence
Scalar-control regions	70	1,035	Per-element scalar loops, data-dependent inner control
Loop-carried dependencies	82	209	Recurrence, neighbor dependence, running state
Irregular memory access	132	652	Gather/scatter, strided access, indexed access
Memory-layout transformation	159	893	AoS, interleaved layout, transpose, tuple-style load/store
Numerical semantics	336	1,166	Integer division/remainder, rounding, floating-point casts
ACLE call-shape constraints	25	815	Predicate width, vector element type, pointer/value argument shape

Table 9: Capability-target distribution in the template-generated and SVE-Loom-guided corpus layers. Tags are non-exclusive and are intended to document coverage targets rather than define mutually exclusive categories.

Item	Count / unit
Corpus examples	5,947
Train / dev split	5,321 / 626
Benchmark tasks	190
SIMDBench / ARM	136 / 54
SIMD Loops	
Comparison mode	Corpus-vs-benchmark pairwise
Compared fields	Prompt, signature, entry point, code, AST, semantic signatures

Table 10: Scope of the benchmark decontamination audit.

the corresponding coverage is broader: 359 of 680 samples activate at least one repair marker, covering 116 of 136 tasks, and 293 samples pass after repair. The ARM SIMD Loops settings show a similar pattern, with repair markers covering 35 tasks for Codex-5.2 and 45 tasks for Kimi-K2.5.

Table 13 separates two main effects. First, NameFix and ShapeFix directly repair a substantial fraction of API-surface failures, especially for Codex-5.2 on SIMDBench. Second, RSB contributes additional successful repairs for harder

candidates whose intended dataflow is not recovered by API/name or shape fixes alone. For Kimi-K2.5 on SIMDBench, RSB succeeds on 199/265 triggered samples, while NameFix/ShapeFix directly resolves 94/197 triggered samples. Among repaired candidates that pass the repair-test oracle, hidden-test retention remains high: 184/189 for Codex-5.2 on SIMDBench, 285/293 for Kimi-K2.5 on SIMDBench, 82/84 for Codex-5.2 on ARM SIMD Loops, and 85/85 for Kimi-K2.5 on ARM SIMD Loops.

I Common-Pass Speedup Analysis

To separate yield effects from runtime changes among already-correct SIMDBench generations, we compare each direct baseline against its SVE-Loom-wrapped counterpart only on samples or tasks that pass correctness in both systems. In the following tables, A denotes the direct baseline and B denotes the same backbone wrapped with SVE-Loom.

Tables 14 and 15 show a split between sample-level and task-level effects. For Kimi-K2.5, the

Layer	Flagging rule	Final status
Exact identity	Task id, sample id, function entry point, prompt hash, signature hash, or code hash exact match.	0 overlap
Token near-duplicate	Prompt token Jaccard ≥ 0.80 ; code token Jaccard ≥ 0.85 .	0 near match
Text fingerprint	n -gram Jaccard ≥ 0.80 , containment ≥ 0.95 , edit ratio ≥ 0.88 , MinHash similarity ≥ 0.84 , or SimHash Hamming distance ≤ 4 .	0 exact/near
AST structural	tree-sitter C/C++ typed, untyped, or composite AST hash exact match; composite exact matches are treated as high-risk.	0 typed / 0 untyped / 0 composite
Semantic high-risk	Same abstract access/control signature plus at least one matching structural component among typed AST, untyped AST, composite AST, array-access tree, or assignment tree.	0 retained high-risk overlap

Table 11: Decontamination audit criteria and final retained-overlap status.

Setting	Total	Any trigger	Final pass	Final fail	Model repair	NameFix	ShapeFix	RSB trigger
Codex SIMDBench	680/136	209/82	189/80	20/11	84/36	140/58	27/24	83/36
Kimi SIMDBench	680/136	359/116	293/111	66/36	265/95	166/74	55/36	265/95
Codex ASL	270/54	120/35	84/31	36/15	75/25	76/29	21/15	75/25
Kimi ASL	270/54	134/45	85/37	49/15	105/35	65/29	25/19	105/35

Table 12: Repair trigger coverage by setting. Each cell reports samples/tasks. Any trigger denotes candidates that activated at least one repair marker recorded by the controller; component columns are not disjoint.

wrapped system is modestly faster on common-pass samples: the sample-level geometric-mean speedup increases from 2.284 to 2.439 and the median from 1.334 to 1.487; task-best geometric-mean speedup also increases from 3.657 to 3.756. Thus, the performance benefit comes mainly from increasing the chance of producing a faster correct candidate per task, not from making every common-passing sample faster.

Setting	RSB pass/trigger	RSB hidden/pass	N/S direct/trigger	N/S direct/final	N/S hidden/direct	All hidden/final
Codex SIMDBench	63/83 (75.9%)	61/63 (96.8%)	126/157 (80.3%)	126/147 (85.7%)	123/126 (97.6%)	184/189 (97.4%)
Kimi SIMDBench	199/265 (75.1%)	192/199 (96.5%)	94/197 (47.7%)	94/163 (57.7%)	93/94 (98.9%)	285/293 (97.3%)
Codex ASL	39/75 (52.0%)	37/39 (94.9%)	45/84 (53.6%)	45/64 (70.3%)	45/45 (100.0%)	82/84 (97.6%)
Kimi ASL	56/105 (53.3%)	56/56 (100.0%)	29/84 (34.5%)	29/52 (55.8%)	29/29 (100.0%)	85/85 (100.0%)

Table 13: Repair pathway success and hidden-test retention. Percentages are sample-level rates. N/S denotes the union of NameFix and ShapeFix triggers; direct means the candidate passes after NameFix or ShapeFix without requiring the later RSB-path repair.

Pair	Common	A geo.	B geo.	A med.	B med.
Codex base vs. Codex+SVE-Loom	266/88	2.340	2.666	1.617	1.616
Kimi base vs. Kimi+SVE-Loom	251/84	2.284	2.439	1.334	1.487

Table 14: Sample-level common-pass speedup comparison under -O3. Common reports common passing samples / covered tasks.

Pair	Tasks	A best geo.	B best geo.	A best med.	B best med.
Codex base vs. Codex+SVE-Loom	88	3.508	3.585	2.221	2.225
Kimi base vs. Kimi+SVE-Loom	94	3.657	3.756	1.722	1.797

Table 15: Task-best common-pass speedup comparison under -O3.