Original scientific article
UDK 004.4:621.3
DOI 10.7251/BLCZR0126237R

EVALUATING THE EXECUTION SAFETY AND SIMD OPTIMIZATION QUALITY OF LLM-GENERATED MODERN C++ CODE

Dušan Rajčević [1]
Ivona Brajević[2]
Goran Jocić [3]

Keywords: *Modern C++, Code Generation, SIMD Vectorization, Memory Safety, Compiler Optimization, Empirical Benchmarking*

Abstract: *Large Language Models (LLMs) are great at generating functional code, but using their output effectively in modern C++ (like C++20 or C++23) for low-latency systems involves some challenges. It's crucial to ensure memory safety and that the code works well with hardware vectorization. This paper presents an easy-to-follow, container-based framework to test the safety and optimization quality of code generated by LLMs for workloads that require a lot of computing power. By using GCC and Clang toolchains, the framework combines dynamic safety tools (like ASan and UBSan) with inspection of target-specific vectorization behavior, with AVX-2, AVX-512 and Arm Neon representing relevant SIMD target classes. The empirical calibration reported here was executed on an Apple M1 Pro in an ARM64 Linux Docker environment and focused on Arm Neon. We also review common issues, such as pointer aliasing and memory allocation patterns, that can complicate vectorization or cause memory problems. In the end, this paper provides performance engineers with a solid way to evaluate, guide and verify code for performance-critical applications. The final calibration produced all six expected sanitizer findings across GCC 14.2.0 and Clang 18.1.3. Each compiler vectorized three of five hot loops, but they selected different kernels: GCC vectorized the stride-two case and explicitly introduced alias-versioned loops, whereas Clang eliminated the temporary allocation in the allocation-heavy kernel and vectorized the transformed loop. Runtime distributions were collected over 50 repetitions as secondary evidence because Docker Desktop executes Linux inside a virtual machine. The results demonstrate that safety validation and SIMD optimization quality require complementary evidence from runtime diagnostics, compiler reports, and machine-code inspection. Model-level prevalence estimates and LLM rankings are outside the scope of the present calibration because no multi-model corpus of generated C++ programs was evaluated.*

1 Faculty for Applied Management, Economics and Finance, Belgrade
2 Faculty for Applied Management, Economics and Finance, Belgrade
3 Faculty for Applied Management, Economics and Finance, Belgrade

PROCJENA SIGURNOSTI IZVRŠAVANJA I KVALITETA SIMD OPTIMIZACIJE MODERNOG C++ KODA GENERISANOG POMOĆU VELIKIH JEZIČKIH MODELA

Ključne riječi: *moderni C++, generisanje koda, SIMD vektorizacija, sigurnost memorije, optimizacija kompajlera, empirijsko testiranje performansi*

Sažetak: *Veliki jezički modeli (LLM) veoma su uspješni u generisanju funkcionalnog koda, ali efikasna primjena njihovih rezultata u modernom C++ jeziku, kao što su C++20 ili C++23, u sistemima koji zahtijevaju nisku latenciju donosi određene izazove. Od ključnog je značaja obezbijediti sigurnost memorije, kao i mogućnost efikasne hardverske vektorizacije koda. U ovom radu predstavljen je jednostavan okvir zasnovan na kontejnerima za testiranje sigurnosti i kvaliteta optimizacije koda generisanog pomoću LLM-ova za računarski zahtjevna radna opterećenja. Korištenjem GCC i Clang skupova alata, predloženi okvir kombinuje dinamičke alate za provjeru sigurnosti, kao što su ASan i UBSan, sa analizom ponašanja vektorizacije specifične za ciljnu arhitekturu, pri čemu AVX-2, AVX-512 i Arm Neon predstavljaju relevantne klase SIMD arhitektura. Empirijska kalibracija prikazana u ovom radu sprovedena je na računaru Apple M1 Pro, u ARM64 Linux Docker okruženju, i bila je usmjerena na Arm Neon. Takođe se razmatraju uobičajeni problemi, poput preklapanja pokazivača (pointer aliasing) i obrazaca alokacije memorije, koji mogu otežati vektorizaciju ili dovesti do problema sa sigurnošću memorije. Rad pruža inženjerima performansi pouzdan metodološki okvir za procjenu, usmjeravanje i verifikaciju koda namijenjenog aplikacijama kod kojih su performanse od presudnog značaja. Završna kalibracija identifikovala je svih šest očekivanih nalaza sanitizatora u GCC 14.2.0 i Clang 18.1.3 okruženjima. Oba kompajlera vektorizovala su tri od pet kritičnih petlji, ali su odabrala različite programske cjeline: GCC je vektorizovao slučaj sa korakom dva i eksplicitno uveo verzionisane petlje radi rješavanja mogućeg preklapanja pokazivača, dok je Clang eliminisao privremenu alokaciju u programskoj cjelini sa intenzivnom alokacijom memorije i vektorizovao transformisanu petlju. Distribucije vremena izvršavanja prikupljene su tokom 50 ponavljanja i korištene kao sekundarni dokaz, s obzirom na to da Docker Desktop izvršava Linux unutar virtuelne mašine. Rezultati pokazuju da provjera sigurnosti i procjena kvaliteta SIMD optimizacije zahtijevaju kombinovanje različitih izvora dokaza: dijagnostike tokom izvršavanja, izvještaja kompajlera i analize mašinskog koda. Procjena zastupljenosti ovih karakteristika na nivou različitih modela, kao i rangiranje LLM-ova, nisu obuhvaćeni ovom kalibracijom, jer nije analiziran korpus C++ programa generisanih pomoću više različitih modela.*

1. Introduction

Large Language Models (LLMs) have practically changed the role of automatic synthesis of programs. Systems trained on large corpuses of code can generate complete functions from specifications using natural language, they can also fix existing code and suggest implementations in different programming languages. Early evaluations such as HumanEval, Mostly Basic Programming Problems (MBPP) and APPS established functional correctness as the dominant measure of code-generation quality (Austin et al., 2021; Chen et al., 2021; Hendrycks et al., 2021). Newer models oriented towards code have significantly improved on these benchmarks (Guo et al., 2024; Rozière et al., 2023).

However, functional correctness isn't the same as being production ready. Liu et al. (2023) showed that stronger test suits can find errors that conventional HumanEval does not. A second source of uncertainty is generation variability. Ouyang et al. (2024) showed that repeated requests may produce significantly different programs even when the task

stays the same. These findings are important when implementation generated by LLMs is used in performance sensitive C++ system, where a program can pass basic tests, but still preserve undefined behavior or implementation paradigms that prevent compiler optimizations.

Security research shows similar concern. Empirical evaluations have found vulnerable code among AI-generated solutions and have shown that access to code assistants does not guarantee secure programming outcomes (Pearce et al., 2022; Perry et al., 2023). SecurityEval, CyberSecEval, SALLM and related work have consequently expanded code-generation evaluation beyond conventional functional tests (Bhatt et al., 2023; Siddiq & Santos, 2022; Siddiq et al., 2024). Other studies have examined security hardening, API misuse and the trade-off between correctness and security (Black et al., 2025; He & Vechev, 2023; Zhong & Wang, 2024). However, most of these approaches, treat generated programs primarily from a software-security perspective rather than examining the interaction between C++ execution safety and low-level compiler optimization.

A parallel research direction dealt with computer efficacy. Mercury, EffiBench, EvalPerf, ECCO and ENAMEL measure execution time, memory usage and relative performance against relevant solutions (Du et al., 2024; Huang et al., 2024; Liu et al., 2024; Qiu et al., 2025; Waghjale et al., 2024). Execution feedback has also been used for improving generated solutions rather than purely score them (Peng et al., 2025). EffiBench-X extend this work to other languages, including C++ (Qing et al., 2025). These tests show that correct solution doesn't necessarily be efficient. However, they offer unlimited information about why compiled C++ implementation use, or does not use available SIMD hardware.

The distinction is especially important for modern C++. Optimizing compilers employ loop and superword-level parallelism (SLP) vectorizers, guided by dependence analysis, alias information, memory-access regularity and machine-specific cost models (Larsen & Amarasinghe, 2000; LLVM Project, n.d.; Nuzman et al., 2006). provides 256-bit vector operations on appropriate x86 processors, while AVX-512 extend the architecture with 512-bit registers and masks. Arm Neon usually operates on 128-bit vectors (Arm Limited, 2020; Intel Corporation, 2024). Purely compilation for an instruction-set architecture (ISA), does not mean that the program will use its widest vectors. Compiler remains free to choose a narrower vectorization factor when its usage model predicts a better outcome.

Modern C++ introduces a second complication. Optimized code can contain pointer arithmetic, manual memory management, alignment assumptions and transformations whose validity depends on language rules. Standard C++ specifies behavior on which compilers may rely on during optimization (International Organization for Standardization [ISO], 2024). AddressSanitizer (ASan) can expose invalid memory accesses, while UndefinedBehaviorSanitizer (UBSan) can detect classes of unspecified behavior, including misalignment and signed integer overflow (Clang Project, n.d.-a, n.d.-b; Serebryany et al., 2012). A performance evaluation that ignores such findings may inadvertently reward fast but invalid programs.

This paper addresses the gap between three research traditions:

1. functional evaluation of generated code,
2. security analysis and
3. performance benchmarking.

The unresolved question is not simply whether LLM generated C++ code can compile or execute quickly, but to establish if its securely executed and produces machine code that will allow compilers to use appropriate SIMD resources.

RQ1: Can reproduced GCC/Clang procedure combine dynamic sanitizer evidence with regular execution tests in order to differentiate C++ code clean of errors from the code with the representing security problems of memory or undefined behavior defects?

RQ2: Can reports about compiler vectorization and generated assembler list combined differentiate vectorization friendly and vectorization hostile C++ patterns under AArch64/ Neon compilation with GCC and Clang?

RQ3: What properties on a source code level in controlled C++ kernels, especially pointer aliasing, non-contagious access, indirect access and allocation inside a hot loop, produce noticeable difference in SIMD compiler decisions?

Contribution is reproduceable procedure evaluation that treats functional correctness, dynamic execution security, SIMD optimization quality and execution performance as separate dimensions instead of another unified score. The controlled calibration reported here executes the safety and SIMD stages in a native-architecture ARM64 Linux environment on Apple Silicon computer and preserves compiler diagnostics, generated assembly, raw sanitizer output, benchmark summaries, source checksums as experimental evidence. Runtime measurements are treated as secondary because the Linux environment is virtualized with Docker Desktop. No model-level comparison is presented because the LLM-generated corpus required for such a comparison was not supplied. This distinction prevents calibration evidence from being misrepresented as evidence about any particular LLM.

2. Methods

Study design

The framework works like a containerized pipeline, broken into five main steps:

1. Bring in the code (ingestion).
2. Compile and check that it runs correctly (functional validation).
3. Run sanitizers to catch memory errors or undefined behavior.
4. Recompile with optimizations and collect vectorization diagnostics.
5. Inspect the machine code to see what the compiler actually produced.

There's also the sixth step which is testing during execution. In the final calibration, this phase was used to collect the median and p95 secondary performance measurements within the same ARM64 environment.

For a full LLM experiment, analysis unit would be a single generated C++ program. Every candidate would be tied to a permanent record: original prompt, name and version of model (or version of endpoint), date of generation, sampling parameters such as temperature, top-p, seed (if supported) and raw output of the model. As Ouyang et al. (2024) pointed out, these sampling configurations are not only incidental metadata, but they directly affect the program that's being produced.

However, in this paper, there was no archived corpus of the model output. That means that researchers couldn't assign things such as sample size, temperature or ranking of the

model. Instead, they focused on controlled calibration programs, manually chosen examples where expected security features and optimization behaviors were already known.

Calibration environment

The calibration was executed on an Apple MacBook Pro equipped with an Apple M1 Pro processor and 16 GB of unified memory. The host system ran macOS 26.6.2 and Docker Desktop 29.7.2. Experiments were executed inside an ARM64 Linux container based on Ubuntu 24.04.4 LTS. The container reported the aarch64 architecture and exposed Arm Advanced SIMD (ASIMD) support, allowing AArch64 binaries to execute without x86 instruction-set emulation.

The container provided GCC 14.2.0 (g++-14, Ubuntu 14.2.0-4ubuntu2~24.04.1) and Clang 18.1.3 (Ubuntu 1ubuntu1). C++20 was used throughout the calibration. Optimized builds used -O3 -std=c++20 -march=armv8-a+simd for both compilers. Clang's AArch64 ASan and UBSan runtime libraries were supplied by libclang-rt-18-dev and the container build verified their presence before the experiment was allowed to run.

GCC vectorization information was collected with -fopt-info-vec-all. Clang used -Rpass=loop-vectorize, -Rpass-missed=loop-vectorize and -Rpass-analysis=loop-vectorize. Compiler diagnostics were retained together with generated assembly so that vectorization decisions could be checked against the emitted machine code.

Docker has isolated compiler dependencies and saved a reproducible Linux tool chain. Every run logged the host configuration, exact compiler version, container architecture, compiling flags, SHA-256 control sums of source files, sanitizer logs, vectorization reports and generated assembly. Runtime measurements were also collected in this environment but were treated as a secondary evidence becase Docker Desktop runs Linux within a virtual machine on macOS.

Safety calibration

Security calibration used AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan). Programs tested with ASan were compiled using the options: -O1, -g, -std=c++20, -fno-omit-frame-pointer and -fsanitize=address. The UBSan tests used the same configuration, but with -fsanitize=undefined. A moderate level of optimization (-O1) kept the compiler's transformations realistic, while the debugging and frame-pointer options improved the diagnostic information produced by the sanitizers.

Three deliberately defective programs were evaluated. ASan was used to detect heap-buffer-overflow caused by writes outside the allocated array. UBSan was used to detect unaligned integer storage and signed overflow with integers. Each program was independently compiled with the GCC and Clang compilers, yielding a total of six security observations.

A test was considered successful when the program was compiled and the expected sanitizer diagnostics appeared in the recorded execution log. The analysis did not rely solely on the exit status of the program, as UBSan can report undefined behavior while still allowing execution to continue.

For future evaluation of LLM-generated code, the "sanitizer-clean" rate is defined as the proportion of successfully executed programs for which neither ASan nor UBSan reports an error.

$$S_{clean} = \frac{N_{executed} - N_{with\ sanitizer\ findings}}{N_{executed}}$$

Individual sanitizer findings should also be recorded by category, such as buffer overflow, misaligned access or signed integer overflow, rather than reporting only the overall sanitizer-clean rate.

SIMD calibration kernels

Five small C++20 kernels were evaluated in the AArch64 environment. They represented patterns of source code often found in numerical or data processing code:

- contiguous element-by-element kernel whose pointers are declared as __restrict;
- same operation without restrictive alias information;
- access to the entrance with step two;
- indirect indexed access;
- and a loop that constructs a four-element std::vector<float> in each iteration.

All kernels are compiled with both GCC and Clang compilers using the same armv8a+simd target. A marker in the source code identified a "hot" loop so that the analysis could link diagnostics to the intended loop rather than loops generated by the libraries or the compiler itself. GCC loop versioning messages were retained when a possible pointer aliasing situation caused the compiler to generate stored vector and scalar paths.

Calibration did not treat a successful vectorization annotation as sufficient evidence in itself. The generated AArch64 assembly code is inspected for 128-bit Neon operations and vector register forms such as q-register loads/writes and vN.4s arithmetic. For Clang, the reported vectorization factor (VF) was retained when it was present. Allocate and deallocate calls were also inspected in the allocation-intensive kernel to determine if the temporary container at the source code level survived the optimization.

For a corpus of generated code, SIMD quality should be reported through at least three separate metrics:

- the percentage of suitable "hot" loops that are vectorized,
- the broadest vector class observed in the final assembly code,
- the presence of optimization qualifiers such as runtime alias checks.

A single binary variable of type "AVX-512 supported" is not enough.

Evaluation metrics

Table 1 summarizes the proposed measurements. The framework deliberately avoids a single weighted overall score because such a score could conceal unsafe behavior behind high performance or reward a fast implementation that fails functional tests.

Dimension	Metric	Evidence	Interpretation
Compilation	Compilation success rate	Compiler exit status and diagnostics	Fraction accepted by the specified toolchain

Functional correctness	Test-pass rate	Deterministic unit/property tests	Fraction satisfying the task specification
Execution safety	Sanitizer-clean rate	ASan and UBSan diagnostics	Fraction with no observed sanitizer finding
SIMD eligibility	Vectorized-loop rate	GCC/Clang optimization reports	Fraction of analyzed hot loops widened
SIMD width	Widest observed vector class	Generated assembly	Actual register class used
Alias overhead	Runtime versioning/check indicator	Compiler remarks and assembly	Whether overlap uncertainty required guarded paths
Runtime	Median execution time	Repeated ARM64 Docker runs	Within-environment central runtime
Tail performance	p95 execution time	Repeated ARM64 Docker runs	Within-environment tail behavior

Table 1. Operational metrics for evaluating generated C++
Note. Source: Original framework definition and final ARM64 calibration.

Runtime measurements are reported as secondary evidence because Docker Desktop executes Linux inside a virtual machine. Systems benchmarking can be affected by environmental noise and should therefore report distributions rather than isolated timings (Chen & Revels, 2016; Kalibera & Jones, 2013; Mytkowicz et al., 2009).

For the exploratory phase at runtime, each kernel has processed $N = 262.144$ of output elements using deterministic input generation with seed 12,345. Five warm-up repetitions preceded the 50 measured repetitions. To obtain measurable sample intervals, each measured repetition performed:

- contig-restrict, contig-alias and stride-two kernels 50 times each,
- indirect kernel 20 times,
- allocation kernel once.

Elapsed time is normalized to microseconds per kernel call. Median and p95 latency were predefined as primary run-time performance summaries, while mean and standard deviation were retained as supplemental dispersion statistics. Per-kernel checksums were compared between GCC and Clang builds to verify that the programs tested produced identical outputs.

3. Results

RQ1: Safety detection

Table 2 shows the final calibration of the sanitizer. Both compiler toolchains detected all three intended classes of defects. ASan classified an invalid heap write as a heap-buffer-overflow, while UBSan reported unaligned storage and signed integer overflow in programs compiled with both the GCC and Clang compilers.

Defect case	Sanitizer	GCC 14.2.0	Clang 17.0.0
Heap write at index 4 of four-element allocation	ASan	Detected: heap-buffer-overflow	Detected: heap-buffer-overflow
Misaligned int store	UBSan	Detected: misaligned store	Detected: misaligned store
INT_MAX + 1 signed overflow	UBSan	Detected: signed integer overflow	Detected: signed integer overflow

Table 2. Dynamic safety calibration results
Note. Six of six predefined compiler-defect observations produced the expected sanitizer finding. The heap out-of-bounds programs exited nonzero under ASan, whereas the UBSan cases could complete with exit status 0 despite emitting the expected diagnostic. Source: Original final ARM64 calibration

The outcome answered RQ1 at the level for which the calibration was designed. All six predefined observations were detected, giving an expected detection rate of 100% for this controlled set (6/6). The result confirms the measurement path and diagnostic parsing for selected defect classes; it does not estimate the frequency of such defects in LLM-generated code or establish the completeness of the sanitizer for unexecuted paths.

RQ2: SIMD diagnostics across compiler and ISA targets

Table 3 shows the final vectorization results reported by the compilers, along with the corresponding assembly code evidence. Both toolchains analyzed the same five "hot" loops under the armv8a+simd target. For Clang, VF indicates the number of 32-bit floating-point elements processed per vector operation when the vectorization factor is reported by the compiler.

Kernel	GCC 14.2.0	Clang 18.1.3
Contiguous, restrict	Vectorized, 16 B Neon	VF=4, 16 B Neon
Contiguous, possible aliasing	Vectorized, 16 B Neon; alias-versioned	VF=4, 16 B Neon
Stride-two access	Vectorized, 16 B Neon; alias-versioned	Not vectorized; cost model
Indirect indexed access	Not vectorized	Not vectorized
Allocation inside loop	Not vectorized; allocation/deallocation retained	VF=4, 16 B Neon; allocation eliminated

Table 3. Compiler vectorization outcomes for the calibration kernels
Note. B = bytes; VF = vectorization factor. Both compilers reported three of five hot loops as vectorized (60%), but the selected kernels differed. Neon evidence was confirmed in generated AArch64 assembly. Source: Original final ARM64 calibration.

GCC and Clang each vectorized three of the five "hot" loops, corresponding to the same aggregate vectorization rate of 60%. Aggregate-level agreement masked different optimization choices. GCC vectorized the contiguous-restrict, contiguous-alias and stride-two kernels. Clang vectorized two contiguous kernels and an intensive allocation kernel, but did not extend the stride-two loop because its cost model reported that vectorization and interleaving were not useful.

Inspection of the assembly code confirmed 128-bit Neon operations for each loop reported as vectorized. GCC's implementation of stride-two included ld2 interleaved loading and 128-bit vector arithmetic. For the GCC kernel with possible aliasing, the optimization report explicitly stated that the loop was versioned due to possible aliasing. The Restrict variant is vectorized without that qualifier.

Clang's allocation kernel provided a contrasting transformation. Although the source code constructed a four-element std::vector<float> in each iteration, the optimized assembly contained no allocation or deallocation calls, and Clang emitted a Neon loop with VF=4. In the GCC build, calls to operator new and sized operator delete remained in the optimized function and the "hot" loop was not vectorized.

RQ3: Source-level optimization barriers and compiler divergence

An aliased kernel showed the difference between successful vectorization and simple vectorization. GCC still vectorized the loop, but explicitly declared loop versioning because pointers can overlap (alias). His optimized program therefore required a stored vector path, rather than using the simpler dependency information available in the restrict variant.

Incorrect memory accesses produced architecture-dependent decisions. Both x86 toolchains vectorized the stride-two and indirect calibration kernels under the configurations tested, although GCC chose a significantly narrower representation for the indirect kernel under the AVX-512 target. In AArch64 compilation, Clang's cost model refused to expand stride-two and indirect loops.

The allocation kernel produced the clearest compiler divergence. GCC's optimized assembly retained calls to the allocation and deallocation routines in the outer loop, and its diagnostics indicated that the outer loop could not be vectorized. Clang's generated assembly did not contain the appropriate allocation calls in the optimized "hot" path. Instead, it transformed the computation into direct arithmetic and issued YMM vector instructions. Thus, the same pattern at the source code level led to different optimization outcomes, as one compiler removed the temporary allocation while the other kept it.

Exploratory runtime measurements

Table 4 summarizes the secondary runtime measurements collected in the ARM64 Docker environment. All kernels processed 262,144 output elements and were measured through 50 iterations after a warm-up phase. Per-kernel checksums matched between GCC and Clang builds.

Kernel	GCC median (µs)	GCC p95 (µs)	Clang median (µs)	Clang p95 (µs)
Contiguous, restrict	38.14	46.15	424.96	432.38
Contiguous, possible aliasing	38.15	42.58	435.28	480.34
Stride-two access	597.03	624.38	104.06	109.06
Indirect indexed access	205.85	356.42	210.23	230.03
Allocation inside loop	2704.27	3304.71	492.94	530.00

Table 4. Exploratory ARM64 runtime measurements
Note. N = 262,144 output elements; 50 measured repetitions; values are microseconds per normalized kernel call. These timings were collected inside Docker Desktop's ARM64 Linux virtual machine and are therefore descriptive within this environment rather than native bare-metal compiler rankings. Source: Original final ARM64 calibration.

Medians within the same environment differed significantly by kernel. GCC produced lower medians for contiguous-restrict and possible-aliasing kernels (38.14 and 38.15 µs) than Clang (424.96 and 435.28 µs). Clang produced lower medians for the stride-two approach (104.06 vs. 597.03 µs) and inside the loop allocation (492.94 vs. 2704.27 µs), while the medians for the indirect approach were similar (205.85 vs. 210.23 µs).

The direction of these differences did not directly match the binary vectorization decision: for example, Clang's non-vectorized stride-two build was faster in this environment than GCC's vectorized version. Because the measurements were obtained in a virtualized Linux environment and were not designed to isolate causal hardware effects, they are reported descriptively and are not used to make claims about general compiler superiority.

4. Discussion

The calibration provides a limited answer to the three research questions, avoiding claims about an LLM corpus that has never been evaluated.

For **RQ1**, the combined ASan and UBSan phase provided detectable evidence for all intentionally introduced defects. This supports the use of "sanitizer cleanliness" as a separate evaluation dimension after functional execution. The distinction is important because passing functional examples cannot confirm the absence of undefined behavior. Also, the sanitizers are dynamic analyzes and confirm only that no included defects were observed on the performed paths. They do not prove memory security for all possible inputs. The original work on AddressSanitizer and the current sanitizer documentation clearly highlight this scope.

For **RQ2**, the combination of optimization diagnostics and code assembly inspection was more informative than compiler-level vectorization rate alone. GCC and Clang both vectorized 60% of the controlled "hot" loops, but the overlap was not complete. GCC accepted the stride-two loop and emitted a vector path based on ld2, while Clang's cost model rejected the same loop. Clang, in turn, eliminated the allocation-intensive temporary container and vectorized the resulting arithmetic, while GCC retained the allocation and deallocation calls. These results are consistent with the compiler literature showing that

vectorization depends on information about dependencies, memory access structure, prior scalar transformations, and cost models, and not just on the appearance of the source code.

For **RQ3**, the calibration identified features of the source code that merit explicit coding in the LLM evaluation dataset. Potential pointer aliasing did not prevent contiguous kernel vectorization, but GCC introduced loop versioning at runtime. The strided approach was vectorized by GCC, but rejected by Clang's cost model, while the indirect approach was not vectorized by any compiler. Allocation inside the "hot" loop was compiler-dependent: Clang removed the temporary container before vectorization, while GCC retained dynamic allocation operations.

This result has an important methodological consequence. Classifying the generated source code solely according to visible patterns is not enough. Allocation appearing in code is a potential optimization bottleneck, but it is not proof that the allocation survives optimization. Similarly, a loop that receives a positive vectorization annotation may require runtime alias checks or other stored paths. Exploratory measurements also show that the "vectorized/non-vectorized" binary is not a reliable indicator of observed runtime in each kernel. Evaluation therefore requires categorization at the source code level, compiler diagnostics, evidence from code assemblies, and, where appropriate, carefully qualified timing measurements.

The proposed framework complements existing LLM benchmarks for code, rather than replacing them. HumanEval and APPS determine whether the generated programs satisfy the given tasks, while EvalPlus demonstrates the importance of stronger correctness testing. SecurityEval, CyberSecEval, and SALLM indicate that security features require special evaluation rather than being derived from correctness. Studies on Copilot and AI-assisted development similarly show that generated or assisted code can retain security defects.

Efficiency benchmarks add another layer. Mercury and EffiBench report differences between functional success and computational efficiency, while EvalPerf and ENAMEL suggest more careful approaches to performance-oriented evaluation. ECCO and PerfCodeGen further demonstrate that execution feedback can be used to modify the generated solutions. EffiBench-X is particularly relevant because it includes C++ among the supported languages. The framework in this paper contributes a lower-level perspective: it asks not only whether one C++ program runs faster than another, but also whether the safety instrumentation remains clean and what the optimizing compiler actually did with the loop.

This difference can improve the evaluation of prompts and models. If two functionally correct candidates differ because one requires alias checks while the other provides a clearer alias-free structure, the performance engineer gets information that the runtime alone cannot provide. Likewise, if one compiler removes allocation at the source code level while another keeps it, classifying the source as "allocation-intensive" would miss a material feature of the broadcast program. Calibration therefore supports evaluation of optimization quality through convergent evidence rather than through a single compiler flag or performance number.

Limitations and requirements for a model-comparison study

The paper has several limitations.

1. The calibration kernels were deliberately kept small. Five ARM64 optimization kernels and three sanitizing defects are sufficient to validate instrumentation paths and

detect compiler-specific transformations, but cannot characterize the distribution of patterns found in actual LLM output.

2. No LLM-generated corpus is provided. Model names, prompt templates, number of samples, generation parameters, and frequency estimates are therefore absent by design. No claim in this paper should be construed as evidence that one LLM is safer or more suitable for vectorization than another.
3. The final calibration was run on a single Apple M1 Pro host and an ARM64 Linux environment provided by Docker Desktop. AArch64 binaries were executed in an ARM64 virtual machine, but the study did not perform native x86-64 AVX2 or AVX-512 execution. References to those instruction sets therefore describe the broader scope and motivation of the framework rather than the empirical results of the final M1 calibration.
4. Runtime latency has been measured but remains secondary evidence. Docker Desktop virtualizes Linux on macOS, and the study did not control for processor frequency, core distribution, operating system deployment, or bare-metal effects. Distributions of 50 replicates are useful for describing behavior within a fixed experimental environment, but should not be generalized into architecture- or compiler-level performance rankings.
5. Sanitizer-clean execution is not equivalent to proof of security. ASan and UBSan work on executed paths and specific fault classes. Static analysis, fuzzing, stronger property-based testing, ThreadSanitizer to detect "data race" situations, and domain-specific validation may be required for production systems.
6. Calibration used C++20. Although the framework can compile C++23 candidates, C++23 was not specifically evaluated in the reported experiment.
7. An extension to model comparison would require supplementing the existing reproducibility framework with a versioned set of prompts and an archived corpus containing correctly generated programs, model identifiers and versions, generation parameters, and resamples where non-deterministic generation is used. Each candidate could then be processed through the same phases of functional testing, sanitizer, vectorization diagnostics, code assembly review, and controlled benchmarking already implemented in the archived framework. This would extend the existing calibration artifact into a model comparison dataset, while preserving the same traceability between reported results and retained experimental evidence.

Extending this work to model comparison would require supplementing the existing reproducibility framework with a versioned set of prompts and an archived corpus containing correctly generated programs, model identifiers and versions, generation parameters, and repeated samples where non-deterministic generation is used. Each candidate could then be processed through the same phases of functional testing, sanitizer, vectorization diagnostics, code assembly review, and controlled benchmarking that are already implemented in the archived framework. This would extend the existing calibration artifact into a model comparison dataset, while preserving the same traceability between reported results and retained experimental evidence.

5. Conclusion

This paper presented a reproducible framework for examining two properties of modern LLM-generated C++ code that conventional functional benchmarks do not sufficiently capture: execution safety under dynamic instrumentation and quality of compiler SIMD optimization.

Controlled calibration answered research questions at the instrumentation level. For RQ1, the sanitizer configurations compiled with GCC and Clang detected all six predefined observations that include out-of-heap access, misalignment, and signed overflow. For RQ2, GCC and Clang each vectorized three of the five "hot" loops under the same AArch64/Neon target, and code assembly review confirmed 128-bit Neon operations for the loops reported as vectorized. For RQ3, a pooled vectorization rate of 60% masked different transformations: GCC used loop versioning due to possible aliasing and vectorized the stride-two kernel, while Clang eliminated temporary allocation in the intensive allocation kernel and vectorized the transformed loop.

The broader implication is methodological. Correctness, execution safety, SIMD quality, and runtime behavior should not be treated as interchangeable indicators of code quality. A program can be correct in tests and yet contain undefined behavior; two compilers can report the same aggregate vectorization rate yet produce different optimized code for the same source patterns. Exploratory timing results further confirm that the vectorization status alone does not determine the observed execution time.

The framework therefore supports a more defensible evaluation procedure for performance-critical C++. In its current form, it does not determine which LLM produces the safest or most suitable vectorization code, as an archived corpus generated from multiple models has not been evaluated. Extending the calibration to replicate outputs of identified models, along with native x86-64 AVX2/AVX-512 executions where necessary, would allow the same stream of evidence to support model-level comparisons, without replacing missing data with imputed results.

Data and Code Availability

The complete reproducibility artifact for this paper, including the Docker configuration, C++ safety calibration programs, SIMD kernels, experiment orchestration scripts, compiler and sanitizer diagnostics, generated AArch64 assembly, benchmark outputs, processed CSV result files, environment metadata and source-file checksums, is publicly available in the associated archive. The archived release contains all materials required to reproduce the reported GCC and Clang calibration experiments on an ARM64 environment and to trace the numerical results presented in this paper to the underlying raw outputs.

The archived v1.0-paper release is permanently available on Zenodo (Brajević et al., 2026) at https://doi.org/10.5281/zenodo.22080067. The corresponding development repository is available at https://github.com/dusanrajcevic/llm-cpp-safety-simd.

Literature

- Arm Limited. (2020). Coding for Neon (Document 102159, Issue 04). Arm.
- Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q. V., & Sutton, C. (2021). Program synthesis with large language models. arXiv. arXiv:2108.07732.
- Brajević, I., Rajčević, D., & Jocić, G. (2026). Reproducibility artifact for "Evaluating the execution safety and SIMD optimization quality of LLM-generated modern C++ code" (Version 1.0-paper) [Computer software and data set]. Zenodo. https://doi.org/10.5281/zenodo.22080067
- Bhatt, M., Chennabasappa, S., Nikolaidis, C., Wan, S., Evtimov, I., Gabi, D., Song, D., Ahmad, F., Aschermann, C., Fontana, L., Frolov, S., Giri, R. P., Kapil, D., Kozyrakis, Y., LeBlanc, D., Milazzo, J., Straumann, A., Synnaeve, G., Vontimitta, V., Whitman, S., & Saxe, J. (2023). Purple Llama CyberSecEval: A secure coding benchmark for language models. arXiv. arXiv:2312.04724.
- Black, G. S., Rimal, B. P., & Vaidyan, V. M. (2025). Balancing security and correctness in code generation: An empirical study on commercial large language models. IEEE Transactions on Emerging Topics in Computational Intelligence, 9(1), 419-430. https://doi.org/10.1109/TETCI.2024.3446695
- Chen, J., & Revels, J. (2016). Robust benchmarking in noisy environments. arXiv. arXiv:1608.04295.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., et al. (2021). Evaluating large language models trained on code. arXiv. arXiv:2107.03374.
- Clang Project. (n.d.-a). AddressSanitizer. LLVM Project documentation.
- Clang Project. (n.d.-b). UndefinedBehaviorSanitizer. LLVM Project documentation.
- Du, M., Luu, A. T., Ji, B., Liu, Q., & Ng, S.-K. (2024). Mercury: A code efficiency benchmark for code large language models. Advances in Neural Information Processing Systems, 37, 16601-16622. https://doi.org/10.52202/079017-0529
- GNU Project. (2024). Using the GNU Compiler Collection (GCC) 14.2.0: Optimize options. Free Software Foundation.
- Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., & Liang, W. (2024). DeepSeek-Coder: When the large language model meets programming: The rise of code intelligence. arXiv. arXiv:2401.14196.
- He, J., & Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (pp. 1865-1879). Association for Computing Machinery. https://doi.org/10.1145/3576915.3623175
- Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., & Steinhardt, J. (2021). Measuring coding challenge competence with APPS. Advances in Neural Information Processing Systems: Datasets and Benchmarks Track, 1.
- Huang, D., Qing, Y., Shang, W., Cui, H., & Zhang, J. M. (2024). EffiBench: Benchmarking the efficiency of automatically generated code. Advances in Neural Information Processing Systems, 37, 11506-11544. https://doi.org/10.52202/079017-0367
- Intel Corporation. (2024). Intel Intrinsics Guide (Version 3.6.9).
- International Organization for Standardization. (2024). ISO/IEC 14882:2024: Programming languages - C++. ISO.
- Kalibera, T., & Jones, R. E. (2013). Rigorous benchmarking in reasonable time. In Proceedings of the 2013 International Symposium on Memory Management (pp. 63-74). Association for Computing Machinery. https://doi.org/10.1145/2464157.2464160

- Larsen, S., & Amarasinghe, S. (2000). Exploiting superword level parallelism with multimedia instruction sets. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (pp. 145-156). Association for Computing Machinery. https://doi.org/10.1145/349299.349320
- Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36.
- Liu, J., Xie, S., Wang, J., Wei, Y., Ding, Y., & Zhang, L. (2024). Evaluating language models for efficient code generation. In Proceedings of the First Conference on Language Modeling.
- LLVM Project. (n.d.). Auto-vectorization in LLVM. LLVM documentation.
- Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong! In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 265-276). Association for Computing Machinery. https://doi.org/10.1145/1508244.1508275
- Nuzman, D., Rosen, I., & Zaks, A. (2006). Auto-vectorization of interleaved data for SIMD. In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 132-143). Association for Computing Machinery.
- Ouyang, S., Zhang, J. M., Harman, M., & Wang, M. (2024). An empirical study of the non-determinism of ChatGPT in code generation. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3697010
- Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In 2022 IEEE Symposium on Security and Privacy (pp. 754-768). IEEE. https://doi.org/10.1109/SP46214.2022.9833571
- Peng, Y., Gotmare, A. D., Lyu, M. R., Xiong, C., Savarese, S., & Sahoo, D. (2025). PerfCodeGen: Improving performance of LLM generated code with execution feedback. In Proceedings of the 2025 IEEE/ACM 2nd International Conference on AI Foundation Models and Software Engineering (pp. 1-13). IEEE. https://doi.org/10.1109/FORGE66646.2025.00008
- Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (pp. 2785-2799). Association for Computing Machinery. https://doi.org/10.1145/3576915.3623157
- Qing, Y., Zhu, B., Du, M., Guo, Z., Zhuo, T. Y., Zhang, Q., Zhang, J. M., Cui, H., Yiu, S.-M., Huang, D., Ng, S.-K., & Luu, A. T. (2025). EffiBench-X: A multi-language benchmark for measuring efficiency of LLM-generated code. Advances in Neural Information Processing Systems, 38, 82577-82620. https://doi.org/10.52202/085713-2491
- Qiu, R., Zeng, W. W., Ezick, J., Lott, C., & Tong, H. (2025). How efficient is LLM-generated code? A rigorous and high-standard benchmark. In Proceedings of the 13th International Conference on Learning Representations.
- Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., et al. (2023). Code Llama: Open foundation models for code. arXiv. arXiv:2308.12950.
- Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. (2012). AddressSanitizer: A fast address sanity checker. In 2012 USENIX Annual Technical Conference (pp. 309-318). USENIX Association.
- Siddiq, M. L., & Santos, J. C. S. (2022). SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (pp. 29-33). Association for Computing Machinery. https://doi.org/10.1145/3549035.3561184

- Siddiq, M. L., Santos, J. C. S., Devareddy, S., & Muller, A. (2024). SALLM: Security assessment of generated code. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (pp. 54-65). Association for Computing Machinery. https://doi.org/10.1145/3691621.3694934
- Waghjale, S., Veerendranath, V., Wang, Z. Z., & Fried, D. (2024). ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (pp. 15362-15376). Association for Computational Linguistics. https://doi.org/10.18653/v1/2024.emnlp-main.859
- Zhong, L., & Wang, Z. (2024). Can LLM replace Stack Overflow? A study on robustness and reliability of large language model code generation. Proceedings of the AAAI Conference on Artificial Intelligence, 38(19), 21841-21849. https://doi.org/10.1609/aaai.v38i19.30185