Extended Abstract

SpecGen: RL-Driven Compiler Verification

Motivation. Manual ISO C++ conformance suites lag rapid language evolution (C++23/26). Stochastic fuzzers such as Csmith Yang et al. (2011) and YARPGen Livinskii et al. (2020) find crashes but rarely target specific standard features (e.g., deducing `this`, multidimensional subscripting, monadic `std::expected`). **SpecGen** automates synthesis of functional test programs via reinforcement learning with a **compiler oracle**.

Method. We implement **cppgen**, a pipeline in which an LLM generates a self-contained C++ translation unit from an ISO-linked prompt; a Python oracle compiles it with both **nvcc** (host-only, GCC backend) and **g++** at `-std=c++23`, runs it in a sandbox, and scores extract / compile / run / check gates. The scalar reward feeds **GRPO** Shao et al. (2024); Fang et al. (2026) fine-tuning of Qwen2.5-Coder-1.5B with LoRA (TRL).

Implementation. Prompts are JSONL `PromptSpec` objects with spec sections, check modes, and timeouts. Baselines use an API model or local HuggingFace inference; GRPO uses oracle-backed rollouts. We ablate optimizer steps (12/24/48) and rollout dtype (fp16 vs. bf16 on NVIDIA L4).

Results. On a 30-prompt hard C++23 benchmark, a strong API baseline achieves **27/30 passed** (mean reward 0.91), validating prompts and oracle. Local Qwen zero-shot on an 8-prompt pilot: **2/8 passed** (0.36). GRPO with broken fp16 rollouts collapsed to 0/8 extracted; with bf16, 12 steps regressed to 1/8, while 24 and 48 steps **tied zero-shot at 2/8**.

Discussion. GRPO trains stably after bf16 fixes (non-zero rewards, finite gradients) but does not exceed zero-shot on this pilot—likely due to sparse compiler rewards, sampled-rollout vs. greedy-eval mismatch, and limited scale. AST-based verification-density shaping (planned at milestone) was deferred.

Conclusion. SpecGen delivers a reproducible compiler-grounded RL loop for ISO-targeted C++ synthesis and documents critical systems lessons for GRPO with small code models. Future work: libclang structural rewards, curriculum training, and full 30-prompt GRPO evaluation.

SpecGen: RL-Driven Compiler Verification

Monami Dutta Gupta
Department of Computer Science
Stanford University
monamidg@stanford.edu

Abstract

We present **SpecGen** (**cppgen**), which fine-tunes a code language model using Group Relative Policy Optimization (GRPO) with rewards from a dual nvcc/gcc compiler oracle on ISO C++23 tasks. We validate a 30-prompt benchmark with a strong API baseline (27/30 pass) and pilot GRPO on eight hard prompts with Qwen2.5-Coder-1.5B. After fixing fp16 sampling failures, GRPO trains with non-zero oracle rewards but does not improve over zero-shot pass rate (2/8). We analyze reward sparsity, dtype engineering, and step-count ablations.

1 Introduction

Modern C++ evolves faster than conformance suites can follow. C++23 introduces multidimensional subscripting, deducing `this`, new ranges adaptors, `if consteval`, and monadic `std::expected` operations—features poorly covered by grammar-based fuzz generators. Large language models can produce standard-conformant programs from natural-language spec snippets, yet probabilistic generation alone does not optimize for compiler stress or targeted verification yield.

SpecGen asks whether a code LLM can be fine-tuned with reinforcement learning when supervision comes only from a real compiler pipeline—compile, run, and functional check—without human labels. We implement this as **cppgen**, treating nvcc and g++ as a multi-stage reward oracle and applying GRPO to a small open-weights coder.

Our research questions are: (1) Can a dual-compiler oracle reliably score ISO-targeted C++23 programs? (2) Does GRPO improve a 1.5B coder over zero-shot on hard conformance prompts? (3) What engineering constraints (dtype, rollout length, step budget) govern feasibility?

Findings. The oracle and prompt bank are sound (27/30 with a strong API model). Local Qwen2.5-Coder-1.5B solves 2/8 poster prompts zero-shot. GRPO is feasible with bf16 rollouts but does not beat zero-shot on the pilot; longer training stabilizes performance without adding passes.

2 Related Work

Stochastic compiler fuzzing. Csmith Yang et al. (2011) and YARPGen Livinskii et al. (2020) generate random programs to expose miscompilations and crashes. They excel at breadth but do not target specific ISO clauses.

LLMs with verifiable rewards. DeepSeek-R1 Guo et al. (2025) demonstrates that reinforcement learning with external verifiers improves reasoning without human preference labels. LLM-VeriOpt Fang et al. (2026) applies GRPO with formal optimization verification (Alive2) on LLVM IR. SpecGen adapts verification-in-the-loop RL to **ISO functional conformance** using production compilers rather than an IR prover.

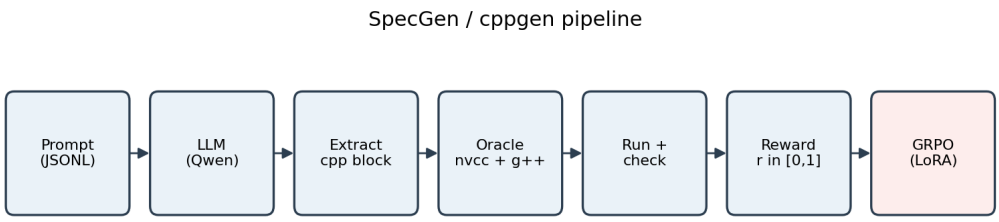

Figure 1: SpecGen / cppgen pipeline. The same oracle score is used for baseline reporting and GRPO rewards.

Gap. Prior LLM compiler RL focuses on optimization correctness or generic code generation. SpecGen targets standard-feature conformance with a **dual nvcc/gcc host pipeline**, reflecting toolchain divergence in CUDA-centric workflows.

3 Method

3.1 Problem Formulation

Given prompt s (ISO-linked task description), policy π_θ generates completion a (a C++ translation unit). Oracle $\mathcal{O}(s, a) \to (v, r)$ returns verdict v and scalar reward $r \in [0, 1]$.

Each **PromptSpec** (JSONL) includes `description`, `spec_section`, `check_mode` (`compile_only`, `exit_zero`, or `stdout_match`), optional `expected_stdout`, and `timeout_sec`.

3.2 Pipeline

Generation. A system prompt requires a single fenced `cpp` block. Regex extraction accepts fenced blocks or raw translation units containing `#include` or `int main`.

Oracle. The pipeline (1) compiles with nvcc and g++ independently (both must succeed), (2) runs the binary under `rlimit` sandbox limits, and (3) applies the prompt-specific check.

Reward. Weighted sum of gates (normalized to $[0, 1]$): extract (0.1), compile (0.3), run (0.2), check (0.4).

3.3 GRPO Training

We fine-tune `Qwen/Qwen2.5-Coder-1.5B-Instruct` with LoRA on attention projections using HuggingFace TRL. For each prompt, the model samples $G{=}4$ completions; `OracleRewardFunction` scores each via $\mathcal{O}$. GRPO computes relative advantages within each group, avoiding a separate critic. Training configs are layered YAML overlays (`grpo_l4.yaml` for bf16 on NVIDIA L4).

4 Experimental Setup

Compilers. CUDA 13.3 nvcc with GCC-15 host; g++-15 at `-std=c++23`.

Datasets. `hard_cpp23.jsonl` (30 prompts) spans deducing-`this`, ranges adaptors, `if consteval`, monadic `expected`, etc.; `poster_cpp23.jsonl` (8 prompts) is our GRPO pilot subset.

Baselines. (1) **API baseline**—NVIDIA Inference API (Claude Opus 4.5) on 30 prompts. (2) **Local zero-shot**—Qwen2.5-Coder-1.5B, greedy decoding, bf16 inference on 8 prompts.

GRPO ablations (8 prompts): fp16 poster config (failed); bf16 with 12, 24, and 48 optimizer steps.

Metrics. Pass rate ($n_{\text{passed}}/n_{\text{prompts}}$), mean oracle reward, and verdict breakdown.

Table 1: API baseline on full hard C++23 benchmark.

Metric	Value
Passed	27 / 30 (90%)
Compile failed	3 / 30
Mean reward	0.91

Table 2: Pass rate and mean reward on 8-prompt pilot.

Method	Passed	Compiled	Mean reward
Qwen zero-shot	2 / 8	3 / 8	0.36
GRPO bf16 (12 step)	1 / 8	2 / 8	0.25
GRPO bf16 (24 step)	2 / 8	3 / 8	0.36
GRPO bf16 (48 step)	2 / 8	3 / 8	0.36
GRPO fp16 (12 step)	0 / 8	0 / 8	0.00

5 Results

5.1 Oracle Validation (30 Prompts, API Model)

Failures included multidimensional subscript syntax (`multidim-subscript-001`), nested ranges adaptors (`ranges-repeat-001`), and `zip_transform` template matching—confirming long-tail C++23 difficulty.

5.2 Local Model and GRPO Pilot (8 Prompts)

Zero-shot and best GRPO runs pass `static-operator-call-001` and `byteswap-001`. Persistent failures include deducing-`this` recursive lambda, `views::zip`, monadic `expected` chains, and multidim subscript; `if-consteval-001` compiles but fails runtime checks.

5.3 Training Diagnostics

fp16 failure mode. GRPO rollouts with fp16 weights produced completions clipped at max length, zero oracle reward, `NaN` gradients, and eval output degenerate strings (e.g., repeated `!` characters). Multinomial sampling hit invalid probability tensors; an lm_head fp32 hook did not restore valid C++ extraction.

bf16 recovery. With bf16 on L4, completions terminate naturally ($\sim$70–250 tokens), training logs show non-zero `rewards/oracle_reward/mean` (up to $\sim$0.78 on early steps) and finite `grad_norm` when `reward_std` > 0. Nevertheless, greedy eval does not surpass zero-shot—consistent with sparse rewards and limited step budget.

5.4 Qualitative Analysis

The 12-step bf16 run **regressed** on `byteswap-001` (passed zero-shot, failed after training); 24+ steps recovered it. This suggests checkpoint sensitivity and unstable advantage signals on sparse rewards rather than monotonic GRPO improvement. Training-sampling (temperature 0.7–0.8) vs. greedy eval may also limit transfer.

6 Discussion

Our milestone revised the hypothesis: binary compile success alone is insufficient for "adversarial" synthesis when capable models already pass most prompts. We planned libclang AST-based **verification density** rewards; implementation was deferred to prioritize GRPO infrastructure (dtype bugs, TRL batch constraints, config merge order).

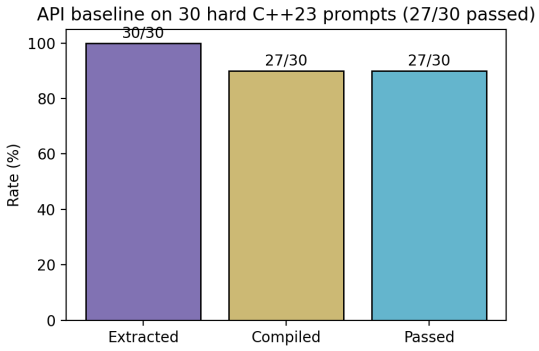

Figure 2: API baseline on the full 30-prompt hard C++23 benchmark.

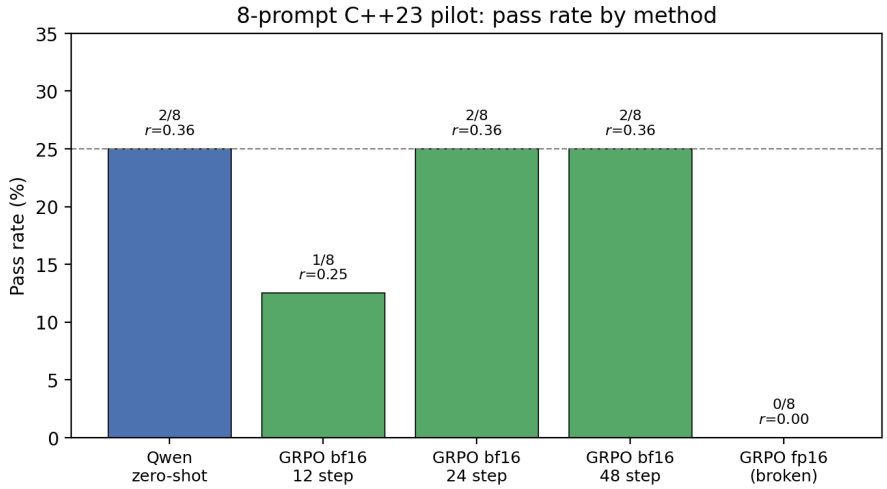

Figure 3: Pass rate on the 8-prompt pilot: zero-shot Qwen vs. GRPO ablations.

Why GRPO did not win. (1) **Reward sparsity**—most rollouts score 0.1 (extract only). (2) **Train/eval mismatch**—sampled rollouts vs. greedy evaluation. (3) **Scale**—8 prompts, 1.5B parameters, $\leq$48 steps. (4) **Engineering overhead**—compiler latency dominates step time ($\sim$5–20 s).

Lessons. Rollout dtype must match model numerical stability; TRL requires `generation_batch_size` divisible by `num_generations`; YAML overlay order matters (smoke config accidentally forced fp16 over bf16).

7 Conclusion

SpecGen/cppgen demonstrates a complete compiler-oracle GRPO loop for ISO C++23 test synthesis. The dual nvcc/gcc oracle and 30-prompt benchmark are validated by a strong API baseline. On a local 1.5B pilot, GRPO with bf16 rollouts trains stably but does not improve pass rate over zero-shot (2/8). The project provides an open foundation for targeted conformance testing and documents systems constraints for RL with compiler feedback.

8 Team Contributions

Solo project—all components below were implemented by Monami Dutta Gupta.

Changes from Proposal. The proposal emphasized adversarial verification density and AST-based targeting (`R_target`, `R_adv`). The delivered system uses a **functional conformance oracle**

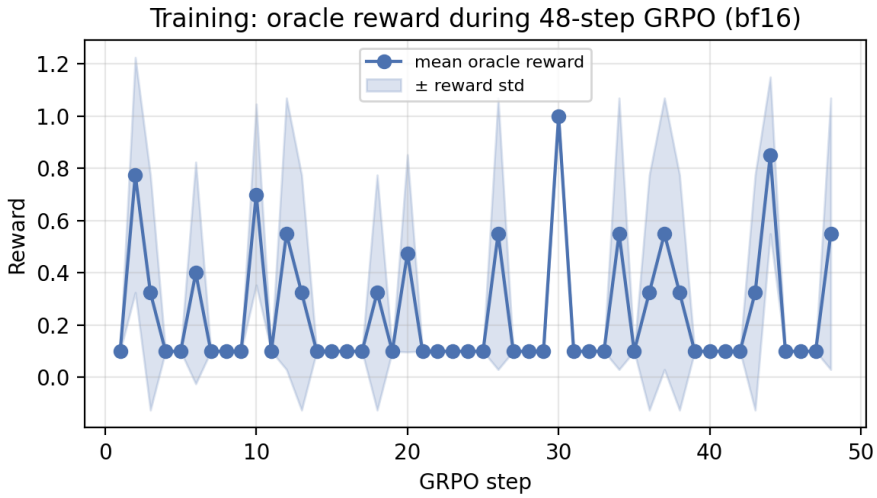

Figure 4: Mean oracle reward during 48-step GRPO training (bf16 rollouts).

(extract/compile/run/check) without libclang shaping or self-BLEU diversity metrics, deferred to prioritize end-to-end GRPO correctness. The milestone's 27/30 result validated the pipeline on 30 prompts; full-set GRPO was scoped to an 8-prompt pilot due to compute and compiler-in-the-loop latency. Original multi-tier adversarial rewards were simplified to a single normalized oracle score aligned with baseline metrics, which enabled direct GRPO integration but may limit exploration of non-idiomatic edge cases.

References

Xiangxin Fang, Jiaqin Kang, Rodrigo Rocha, Sam Ainsworth, and Lev Mukhanov. 2026. LLM-VeriOpt: Verification-Guided Reinforcement Learning for LLM-Based Compiler Optimization. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 740–755.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, et al. 2025. DeepSeek-R1 Incentivizes Reasoning in LLMs through Reinforcement Learning. *Nature* 645, 8081 (2025), 633–638.

Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300* (2024).

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294.

A Reproducibility

Key commands (from repository root):

```
source .venv/bin/activate
cppgen grpo-check
```

```
cppgen baseline \
  --prompts data/prompts/hard_cpp23.jsonl \
  --out data/outputs/baseline_hard

cppgen grpo-eval --config configs/grpo_l4.yaml \
  --model Qwen/Qwen2.5-Coder-1.5B-Instruct \
  --prompts data/prompts/poster_cpp23.jsonl \
  --out data/outputs/poster_qwen_zero

cppgen grpo --config configs/grpo_l4.yaml \
  --config configs/grpo_48steps.yaml \
  --prompts data/prompts/poster_cpp23.jsonl \
  --model Qwen/Qwen2.5-Coder-1.5B-Instruct \
  --out data/outputs/poster_grpo_bf16_48steps
```

B Implementation Details

Repository layout: `src/cppgen/` (oracle, generation, GRPO, CLI), `configs/` (YAML overlays), `data/prompts/` (JSONL benchmarks), `data/outputs/` (artifacts per run). Reward weights and sandbox limits are configured in `configs/default.yaml`. GRPO requires bf16 (or fp32) rollouts for Qwen2.5-Coder-1.5B on NVIDIA L4; fp16 sampling produces invalid logits during training.