

FHE-CodeEval: Benchmarking Large Language Models for Fully Homomorphic Encryption Code Generation

Qiuchu Yu^{1,2} Guangli Li^{2,3,*} Shuoming Zhang² Ruiyuan Xu²
 Qihan Cai³ Jingling Xue³ Xiaobing Feng² Huimin Cui^{1,2}

¹School of Advanced Interdisciplinary Sciences,
 University of Chinese Academy of Sciences, Beijing, China

²Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

³School of Computer Science and Engineering, UNSW Sydney, Australia

*Corresponding author.

Abstract

Large language models (LLMs) are increasingly used for code generation, but their ability to program specialized cryptographic systems remains poorly understood. Cheon–Kim–Kim–Song (CKKS), a fully homomorphic encryption (FHE) scheme supporting approximate arithmetic over packed real- and complex-valued vectors, is particularly relevant to machine-learning workloads. Prior work has not systematically examined the CKKS-specific challenges of LLM-generated encrypted programs. To address this gap, we introduce FHE-CodeEval, a 46-case benchmark that evaluates three core challenges: packing, nonlinear approximation, and level/scale management, as well as how models coordinate decisions across them. We further introduce *communication-aware efficiency*, an FHE-specific metric comparing generated programs with compiler baselines using server-side runtime and ciphertext footprint.

We evaluate eight recent proprietary and open-weight LLMs on FHE-CodeEval. Across all 46 cases, the best observed scores reach 51.7% Pass@1 and 69.6% Pass@5 with iterative execution feedback, but only 7.8% Pass@1 and 13.0% Pass@5 in the one-shot setting. For efficiency, even the best-performing model achieves a geometric-mean communication-aware efficiency of only 0.44 over its valid rollouts in the compiler-comparable subset. These results reveal a substantial gap between functional correctness and efficiency in LLM-generated CKKS programs. Code is available at <https://github.com/qiuchuy/FHE-CodeEval>.

1 Introduction

Fully homomorphic encryption (FHE) (Gentry, 2009) enables computation directly over encrypted data, allowing an untrusted server to process sensitive inputs without observing their plaintext values.

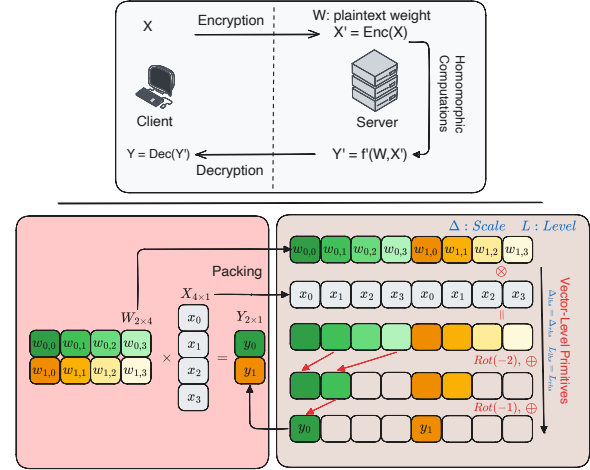


Figure 1: **Above:** The FHE workflow of client-side encryption and decryption with server-side homomorphic evaluation. **Below:** CKKS matrix–vector multiplication using packed slots, element-wise multiplication, rotations, and additions under explicit level/scale management.

This property makes FHE appealing for privacy-preserving machine learning, financial analytics, medical data processing, and outsourced computation over personal data. Among modern FHE schemes, Cheon–Kim–Kim–Song (CKKS) (Cheon et al., 2017) is especially relevant to machine-learning workloads because it supports approximate arithmetic over packed real-valued vectors.

Figure 1 illustrates the typical CKKS workflow. A client encrypts its plaintext inputs into ciphertexts and sends them to an untrusted server. The server evaluates the target computation over ciphertexts/plaintexts using homomorphic primitives such as addition, multiplication, and rotation, without observing the underlying plaintext values. It then returns the resulting ciphertext to the client, which decrypts it to recover the final plaintext result.

Despite this promise, writing CKKS programs is substantially different from writing plaintext pro-

grams. An FHE programmer must express the computation using primitives that operate on encrypted ciphertext slot vectors rather than ordinary scalar or tensor variables. Figure 1 illustrates this gap with matrix-vector multiplication. Here, W denotes server-held plaintext weights, whereas X is a client input encrypted as a ciphertext. In plaintext, the program can directly compute $W \times X$. In CKKS, by contrast, the programmer must first choose how to pack the matrix and vector into vector slots. For example, with 8 available slots, W can be packed row-wise into a plaintext p_1 , while X can be replicated across a ciphertext c_1 . Multiplying p_1 and c_1 then produces slot-wise ciphertext partial products, not the final matrix-vector output. The program must explicitly rotate and add ciphertexts to aggregate these partial products, while maintaining compatible levels and scales across all operands.

More broadly, this example shows that writing correct CKKS programs requires reasoning about how data is represented and transformed throughout an encrypted computation while keeping each step compatible with the cryptographic context and the library API. These requirements expose three central programming challenges under CKKS. **Packing:** tensor computations must be converted to slot-wise operations, and the programmer must choose a slot layout, align values before combining them, and insert explicit rotations to aggregate results across slots. **Nonlinear approximation:** CKKS supports limited primitives such as addition, multiplication by a plaintext or ciphertext, and slot rotation, whereas functions such as ReLU are unavailable and must be replaced by polynomial or iterative approximations valid over a bounded input range. **Level/scale management:** ciphertext-ciphertext multiplication increases noise and multiplies ciphertext scales. Because rescaling reduces the scale at the cost of consuming a level from the modulus chain, the programmer must budget multiplicative depth, place rescaling operations appropriately, and align ciphertext levels and scales before combining them. These challenges are interdependent: a higher-degree polynomial approximation may improve accuracy, but its additional level consumption can cause the circuit to exceed the multiplicative-depth budget under the existing packing strategy.

Researchers have widely studied LLMs for code generation. Existing evaluations cover general-purpose programming tasks, such as generating

standalone functions (Chen et al., 2021) or resolving repository issues (Jimenez et al., 2024), and domain-specific tasks, such as using data-science libraries (Lai et al., 2023) or generating efficient GPU kernels (Ouyang et al., 2025). These benchmarks show that code-generation ability cannot be evaluated solely through syntax or generic functional correctness; the benchmark must also reflect the execution model, constraints, and performance requirements of the target domain. CKKS program synthesis is precisely such a domain-specific setting; its encryption-specific constraints demand capabilities beyond general-purpose code generation, raising a key question: *Can LLMs, which have shown strong general code-generation ability, synthesize executable CKKS programs?*

Prior studies have explored related aspects of LLM-assisted FHE programming, but they do not systematically study this question in terms of the specific programming challenges posed by CKKS. FHE-Agent (Xu et al., 2025) selects CKKS compiler configurations for encrypted inference, while FHE-Coder (Kumar et al., 2026) generates FHE code from natural-language tasks with retrieval on a small task set. Neither work provides a fine-grained evaluation of the core CKKS programming challenges that arise when synthesizing encrypted programs from plaintext specifications.

FHE-CodeEval addresses this gap through a program-to-program generation setting that does not require reference CKKS implementations. Figure 2 provides an overview of the benchmark. Each benchmark case provides a PyTorch plaintext implementation and asks an LLM to synthesize a functionally equivalent CKKS program using OpenFHE (Badawi et al., 2022). The benchmark contains 46 cases spanning packing-oriented operators, approximation-oriented operators, compositional subgraphs and neural-network workloads. These cases are organized around three core capabilities required for realistic CKKS programming: packing, nonlinear approximation, and level/scale management, as well as how models coordinate combined decisions across them.

FHE-CodeEval evaluates each generated CKKS program by executing it on test inputs, decrypting the output, and comparing the result with the PyTorch reference within a numerical tolerance. To prevent spurious correctness, FHE-CodeEval uses a two-stage hacking detection pipeline, supported by a plaintext simulator, to identify fallbacks such as replacing homomorphic

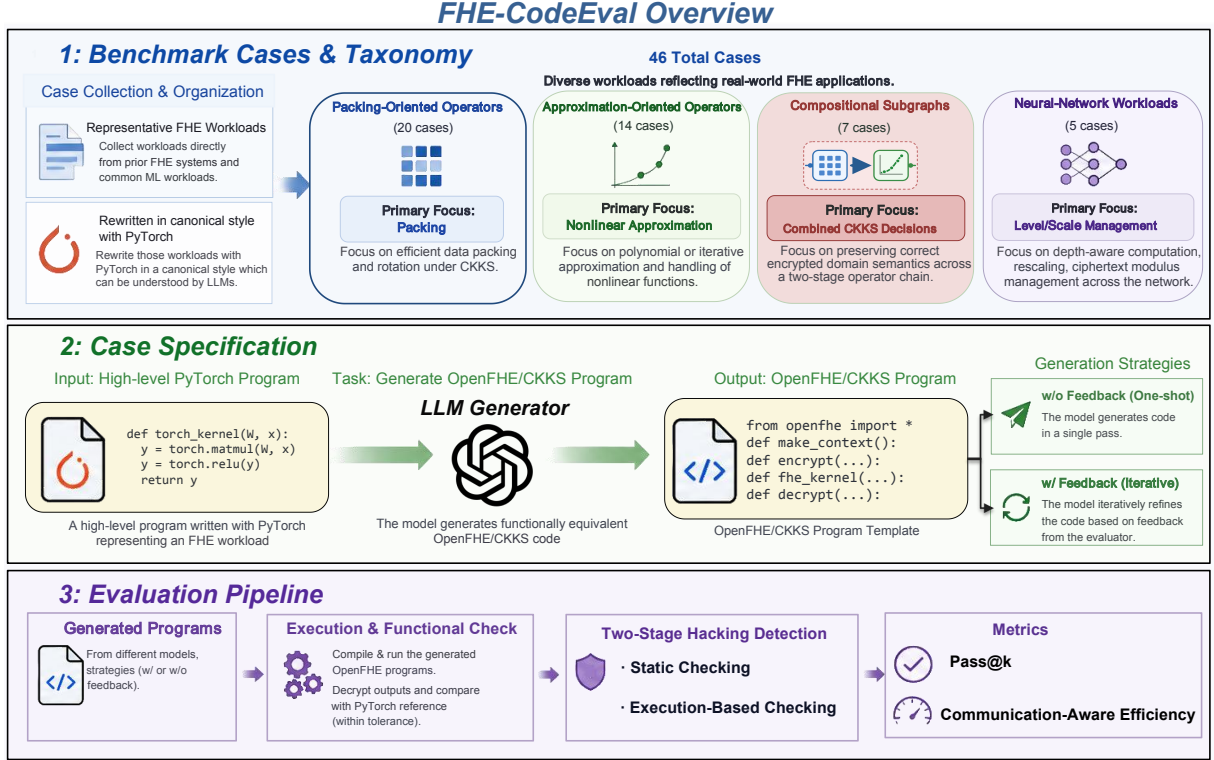


Figure 2: Overview of FHE-CodeEval benchmark construction, four-category taxonomy, task specification, and evaluation pipeline.

computation with PyTorch operations. We report Pass@1 and Pass@5 under this correctness criterion. To evaluate efficiency, we select five Rotom-derived cases (Chen et al., 2026) with compiler-generated baselines. For this subset, we introduce *communication-aware efficiency*, a metric that jointly captures server-side kernel runtime and ciphertext footprint, reflecting the compute-communication cost of an FHE workload. Together, these measures allow FHE-CodeEval to assess not only whether LLM-generated OpenFHE/CKKS programs are correct, but also whether they can match compiler-generated implementations under a realistic compute-communication trade-off.

Our contributions are as follows:

- We introduce FHE-CodeEval, a 46-case benchmark in four categories that evaluates packing, nonlinear approximation, and level/scale management, as well as how models coordinate combined decisions across the challenges.
- We introduce *communication-aware efficiency*, an FHE-specific metric that compares generated programs with compiler baselines while accounting for both server-side kernel runtime and ciphertext footprint.
- We evaluate eight proprietary and open-weight

LLMs on FHE-CodeEval. Across all 46 cases, iterative execution feedback raises the best Pass@1 from 7.8% in the one-shot setting to 51.7%, with Pass@5 reaching 69.6%. Among valid rollouts in the five-case compiler-comparable subset, the largest model-level geometric-mean communication-aware efficiency is 0.44.

- We build a complete evaluation pipeline for OpenFHE/CKKS code generation, including real OpenFHE execution and a two-stage hacking detection pipeline powered by a plaintext OpenFHE simulator, which enables rapid pre-validation and lightweight debugging of LLM-generated OpenFHE/CKKS programs.

2 Related Work

LLM for Code Generation. Prior work evaluates LLM code generation on general programming tasks, including standalone functions and repository-level issue resolution (Chen et al., 2021; Austin et al., 2021; Jimenez et al., 2024), as well as domain-specific tasks involving data-science libraries, GPU kernels, RTL, quantum programs, and SIMD intrinsics (Lai et al., 2023; Ouyang et al., 2025; Liu et al., 2023; Guo et al., 2025; He et al.,

2025). These benchmarks show that evaluation should reflect the execution model and constraints of the target domain. Recent LLM-for-FHE efforts address related workflow components: FHE-Agent (Xu et al., 2025) selects CKKS compiler configurations, while FHE-Coder (Kumar et al., 2026) generates FHE code from natural-language tasks with retrieval on a small task set. FHE-CodeEval instead systematically evaluates CKKS program synthesis from executable plaintext references, focusing on packing, nonlinear approximation, level/scale management and how models coordinate these decisions.

FHE Compilers. FHE compilers lower higher-level representations to optimized encrypted implementations. EVA (Dathathri et al., 2020) introduces an encrypted-vector language with automatic scale management; Rotom (Chen et al., 2026) proposes a tensor-level domain-specific language (DSL) that optimizes ciphertext layout and rotation schedules; CHEHAB (Seddiki et al., 2026) combines auto-vectorization with equality saturation over a scalar-based FHE DSL; and ANT-ACE (Li et al., 2025) lowers ONNX neural-network graphs to CKKS programs. These compilers substantially reduce manual FHE engineering, but each requires the input to be expressed in a supported abstraction. FHE-CodeEval instead evaluates whether LLMs can synthesize valid CKKS implementations directly from executable plaintext references, without relying on a compiler-specific input language. For a compiler-comparable subset of benchmark cases, we use Rotom-generated implementations as performance baselines.

3 Benchmark Design

3.1 Benchmark Case Construction and Organization

Table 1 summarizes the 46 representative computations curated by FHE-CodeEval from prior FHE systems and common machine-learning workloads. Each case is represented by an executable PyTorch (Paszke et al., 2019) reference function `torch_kernel(...)` and structured metadata defining input shapes, data roles, value ranges, and accuracy tolerances. Together, they form the plaintext specification that the generated OpenFHE/CKKS program must match. The cases form four categories: packing-oriented and approximation-oriented operators, compositional subgraphs, and neural-network workloads. The

two operator categories isolate packing and nonlinear approximation. The latter two evaluate how models coordinate these decisions with level/scale management at increasing scales.

Packing-Oriented Operators. We include 20 cases that test packing and rotation strategies. Five cases adapted from Rotom (Chen et al., 2026) stress tensor layout; twelve CHEHAB (Seddiki et al., 2026)-derived cases require lifting scalar code into slot-wise CKKS operations; and three cases cover reductions and pooling common in neural-network inference. All source workloads are manually translated from their original DSLs into PyTorch and validated against the original correctness checks. Communication-aware efficiency evaluation is restricted to the five Rotom-derived cases, which have comparable compiler-generated baselines.

Approximation-Oriented Operators. We include 14 cases that test approximation under bounded input ranges. In CKKS, non-polynomial functions must be approximated over bounded domains. Following prior FHE inference work that substitutes HE-compatible approximations for standard nonlinearities (Gilad-Bachrach et al., 2016; Brutzkus et al., 2019), we select 14 functions commonly used in machine-learning workloads. Each case specifies the target function and its input range, and requires the generated program to produce an accurate approximation within that domain.

Compositional subgraphs. We include 7 compositional subgraphs as an intermediate tier between isolated operators and neural-network workloads. Each case combines a linear arithmetic computation with an activation. The model must jointly select a packing layout, implement the activation, and maintain compatible depth, levels, and scales. These subgraphs serve as diagnostic components for testing whether a model can integrate the core CKKS challenges before scaling to larger neural-network workloads.

Neural-network workloads. We collect 5 cases to test whole-program CKKS synthesis across deeper computation graphs. These workloads are drawn from computations commonly targeted by FHE systems (Krastev et al., 2024; Ebel et al., 2025; Chen et al., 2026; Boemer et al., 2019) and jointly exercise tensor packing, nonlinear handling, and level/scale management at network scale, with level/scale management as the primary focus.

Category	Case names	Primary evaluation target
Packing-Oriented Operators (20)	matmul, double-matmul, ttm, convolution, logreg-matvecmul, dot-product, hamming-dist, l2-distance, lin-reg, poly-reg, poly-derivative, dct, box-blur, gx-kernel, gy-kernel, sobel, roberts-cross; sum, mean, avgpool	Packing
Approximation-Oriented Operators (14)	relu, gelu, sigmoid, silu, exp, sqrt, sign, min, softmax, layernorm, batchnorm, tanh, log, inverse	Nonlinear Approximation
Compositional subgraphs (7)	linear-relu, conv2d-relu, linear-square, conv2d-square, linear-softmax, sum-relu, sum-square	Combined CKKS Decisions
Neural-network workloads (5)	mlp, bert-attention, lenet5, resnet20, squeezenet	Level/Scale Management

Table 1: Summary of the 46 benchmark cases in FHE-CodeEval.

3.2 Case and Prompt Format

Each FHE-CodeEval benchmark case defines a task prompt with a plaintext specification and a required encrypted implementation interface. The specification consists of an executable PyTorch function `torch_kernel(...)` and structured metadata for each argument, including its shape, dtype, value range, and evaluation role as either encrypted client data or plaintext server data. The value range defines the testing domain and runtime test cases are randomly sampled from the corresponding interval.

The model must return a Python module with four entry points:

- `make_context()`: constructs the CKKS cryptographic context, generates keys, and registers rotation indices. Encryption parameters such as multiplicative depth and ring dimension are declared as module-level constants.
- `encrypt(cc, keys, ...)`: encodes each input according to its declared role and encrypts client data into ciphertexts while packing server-side weights or constants as plaintexts.
- `fhe_kernel(cc, keys, enc_inputs)`: runs the encrypted computation using homomorphic primitives such as addition, multiplication, and rotation. This function is the only wall-clock-timed region, following the convention used by Rotom (Chen et al., 2026).
- `decrypt(cc, keys, ct_out)`: decrypts the output ciphertexts and reconstructs a `torch.Tensor` with the same shape as the PyTorch reference output.

This interface exposes the main decisions required in CKKS programming: The model needs to choose global cryptographic parameters in `make_context`, decide how to pack tensors into ciphertext slots in `encrypt`, implement the homomorphic computation in `fhe_kernel`, and reconstruct the plaintext result in `decrypt`. Figure 3 shows how these materials are assembled into each

benchmark prompt; the full prompt template is provided in Appendix A.

4 Evaluation Pipeline

4.1 Runtime Environment

All benchmark evaluations are run on a server with dual Intel Xeon Gold 6248 CPUs (2.50 GHz) and 768 GB RAM, running Ubuntu 22.04. The software stack consists of the OpenFHE Python package OpenFHE 1.5.1, PyTorch 2.5.0, and Rotom at commit d1f4323.

4.2 Models

We evaluate eight recent LLMs from proprietary and open-weight families. The evaluated models are GPT-5.4, Sonnet-4.6, GLM-5.1, DeepSeek-V3.2, GPT-OSS-120B, Qwen3-Coder-480B-A35B-Instruct, Qwen3.5-35B-A3B, and Qwen3.5-397B-A17B. This model set covers proprietary frontier systems, large open-weight coding models, and general-purpose open-weight models at different scales.

4.3 Metrics

FHE-CodeEval evaluates generated FHE programs on two axes: functional correctness and communication-aware efficiency.

Functional correctness. Following Chen et al. (2021), we report Pass@1 and Pass@5 using the unbiased Pass@ k estimator. A candidate is correct if it executes end to end, passes hacking detection, and produces a decrypted output that matches the PyTorch reference with tolerance 10^{-2} under `torch.allclose`. The 10^{-2} threshold accounts for CKKS approximation noise (encoding, rescaling, polynomial approximations) while remaining tight enough to expose incorrect packing or poor approximations.

Communication-aware efficiency. For five cases we use directly comparable baselines generated by

Input Description		Plaintext Program Reference	Profiling Analysis (if available)	
Role	Ciphertext/Plaintext	import torch def torch_kernel(x: torch.Tensor): return torch.relu(x)	Layers	Observed Range
Data Range	[a, b]		layer1.relu	[c, d]
Data Type	float32, ...		layer2.relu	[e, f]
Shape	(m, n, ...)		layer3.relu	[g, h]
LLM Response				
from openfhe import * def make_context(): ... \n def encrypt(...): ... \n def fhe_kernel(...): ... \n def decrypt(...): ...				

Figure 3: Construction of a benchmark case prompt in FHE-CodeEval.

Rotom. For task i and correct candidate j , let T_i^R and T_{ij} be the Rotom and generated server-side kernel runtimes, and let $F_{ij} = C_{ij} \cdot N_{ij}$ approximate the online encrypted-data volume (ciphertext count times ring dimension), following the communication-cost model in Rotom. We define communication-aware efficiency as:

$$E_{ij}^{\text{comm}} = \frac{T_i^R}{T_{ij}} \cdot \frac{F_i^R}{F_{ij}}. \quad (1)$$

This metric jointly rewards candidates that are both faster and more communication-compact relative to the compiler baseline. Values above 1 indicate that the generated program outperforms Rotom under this combined compute–communication metric.

4.4 Prompt Settings

We evaluate two prompt settings:

- **Without feedback (one-shot).** The model receives the case prompt once and must directly return a complete OpenFHE/CKKS implementation. The prompt follows the case format in Section 3.2; it includes the PyTorch reference, input metadata, and required interface, but no evaluator result or reference FHE solution.
- **With iterative feedback.** The model starts from the same initial prompt. If the generated program fails, we send back the previous code together with evaluator feedback and ask the model to return a full corrected implementation. Feedback reports the observed failure, such as import errors, runtime exceptions, type mismatches, or numerical discrepancies, but never includes corrected code or a reference implementation. The feedback setting permits up to five repair rounds for each response. Table 2 summarizes representative feedback messages; Appendix Table 5 provides a more detailed taxonomy of common error types with incorrect and corrected code examples.

Description	Feedback
Incorrect Packing	The size [32768] of the vector with values should not be greater than ringDim/2 [16384] if the scheme is CKKS.
Insufficient Multiplicative Depth	The current multiplicative depth (1) is insufficient. Increase it.
Wrong Implementation	Sample 5/5 failed: max_abs_err = ..., avg_abs_err = ...

Table 2: Representative feedback messages used in the with-feedback setting.

4.5 Hacking Detection

FHE-CodeEval applies a two-stage hacking detection pipeline to ensure that accepted programs perform the target computation homomorphically rather than through plaintext shortcuts.

- **Static Checking.** The first stage checks the submitted module. It verifies the required interface and rejects source-level patterns that indicate plaintext fallback, such as case-specific PyTorch operators or forbidden encryption, decryption, and plaintext-extraction APIs inside `fhe_kernel`.
- **Execution-Based Checking.** The second stage runs the candidate with a plaintext OpenFHE simulator to track whether `fhe_kernel` invokes OpenFHE compute APIs; programs with no such calls are flagged as hacking. We also use `torch.profiler` to trace execution and detect case-specific PyTorch calls that would reveal a plaintext implementation of the target operation.

5 Results

Table 3 reports Pass@ k on the full benchmark and by benchmark category; Figure 4 summarizes communication-aware efficiency on the five Rotom-derived cases, with full rollout-level measurements deferred to Appendix C.

Category	Model	Pass@1		Pass@5	
		w/o feedback	w/ feedback	w/o feedback	w/ feedback
Overall	GPT-5.4	5.2%	26.1%	10.9%	45.7%
	Sonnet-4.6	7.8%	51.7%	10.9%	69.6%
	GLM-5.1	6.5%	24.3%	8.7%	37.0%
	DeepSeek-V3.2	1.7%	13.0%	6.5%	21.7%
	GPT-OSS-120B	3.9%	30.0%	10.9%	39.1%
	Qwen3-Coder-480B-A35B-Instruct	2.2%	14.3%	2.2%	23.9%
	Qwen3.5-35B-A3B	1.3%	3.9%	4.3%	8.7%
	Qwen3.5-397B-A17B	3.9%	26.1%	13.0%	43.5%
Packing-Oriented Operators	GPT-5.4	12.0%	46.0%	25.0%	75.0%
	Sonnet-4.6	17.0%	70.0%	20.0%	80.0%
	GLM-5.1	15.0%	37.0%	20.0%	55.0%
	DeepSeek-V3.2	4.0%	25.0%	15.0%	45.0%
	GPT-OSS-120B	9.0%	60.0%	25.0%	75.0%
	Qwen3-Coder-480B-A35B-Instruct	5.0%	27.0%	5.0%	45.0%
	Qwen3.5-35B-A3B	3.0%	9.0%	10.0%	20.0%
	Qwen3.5-397B-A17B	9.0%	46.0%	30.0%	70.0%
Approximation-Oriented Operators	GPT-5.4	0.0%	5.7%	0.0%	21.4%
	Sonnet-4.6	1.4%	51.4%	7.1%	85.7%
	GLM-5.1	0.0%	10.0%	0.0%	21.4%
	DeepSeek-V3.2	0.0%	0.0%	0.0%	0.0%
	GPT-OSS-120B	0.0%	0.0%	0.0%	0.0%
	Qwen3-Coder-480B-A35B-Instruct	0.0%	1.4%	0.0%	7.1%
	Qwen3.5-35B-A3B	0.0%	0.0%	0.0%	0.0%
	Qwen3.5-397B-A17B	0.0%	7.1%	0.0%	21.4%
Compositional Subgraphs	GPT-5.4	0.0%	28.6%	0.0%	42.9%
	Sonnet-4.6	0.0%	37.1%	0.0%	57.1%
	GLM-5.1	0.0%	34.3%	0.0%	42.9%
	DeepSeek-V3.2	0.0%	14.3%	0.0%	14.3%
	GPT-OSS-120B	0.0%	25.7%	0.0%	42.9%
	Qwen3-Coder-480B-A35B-Instruct	0.0%	14.3%	0.0%	14.3%
	Qwen3.5-35B-A3B	0.0%	0.0%	0.0%	0.0%
	Qwen3.5-397B-A17B	0.0%	25.7%	0.0%	42.9%
Neural-network workloads	All models	0.0%	0.0%	0.0%	0.0%

Table 3: Pass@ k results by category and prompt setting.

5.1 Correctness Analysis

Overall performance. Without feedback, overall correctness remains low. Sonnet-4.6 achieves the best Pass@1 at 7.8%, while Qwen3.5-397B-A17B obtains the best Pass@5 at 13.0%; all other models stay below these levels. With feedback, the results improve substantially but remain uneven across models. Sonnet-4.6 becomes the clear leader, reaching 51.7% Pass@1 and 69.6% Pass@5. All other models remain below 50% on both Pass@1 and Pass@5 even with feedback. These results show that current LLMs are not yet broadly reliable at OpenFHE/CKKS code generation: feedback can help strong models repair many local errors, but success still depends heavily on FHE-specific reasoning about interfaces, packing, parameters, and numerical approximation.

Results by category. To examine LLM performance across different types of FHE programming tasks in greater detail, we report correctness separately for each benchmark category.

- **Packing-oriented operators.** Packing-oriented operators are the most tractable, relying on standard homomorphic primitives rather than approximation choices. Without feedback, Sonnet-4.6 achieves the best Pass@1 at 17.0%, while Qwen3.5-397B-A17B obtains the best Pass@5 at 30.0%. Feedback leads to a clear improve-

ment: Sonnet-4.6 reaches 70.0% Pass@1 and 80.0% Pass@5, and five of the eight models exceed 50% Pass@5 with feedback. This indicates that many failures on packing-oriented operators are local and repairable once the model receives execution diagnostics.

- **Approximation-oriented operators.** Approximation-oriented operators are substantially harder. Without feedback, Sonnet-4.6 is the only model that ever produces a correct CKKS program, obtaining 1.4% Pass@1 and 7.1% Pass@5. Feedback again helps, but the gains are concentrated in Sonnet-4.6, which reaches 51.4% Pass@1 and 85.7% Pass@5; no other model exceeds 21.4% Pass@5. Success requires jointly choosing the approximation range, polynomial degree, and multiplicative depth while designing the approximation itself. An error in any one of these decisions can cause large numerical deviations, making this category challenging for most models.
- **Compositional subgraphs.** All eight models obtain 0% Pass@1 and Pass@5 without feedback. With feedback, Sonnet-4.6 performs best at 37.1% Pass@1 and 57.1% Pass@5. Sonnet-4.6 is the only model to solve sum-relu; the other models’ accepted successes are confined to square-based cases. We find three recurring

problems in failed subgraph generations. First, models implement the initial linear computation incorrectly through packing, indexing, or rotation mistakes. Second, approximation accuracy and cryptographic parameters are tightly coupled: increasing the polynomial degree also increases multiplicative depth and modulus-chain requirements, and models often fail to find a compatible combination within five feedback rounds. Third, misplaced or omitted rescaling causes level/scale incompatibilities or excessive numerical error. These failures already arise in two-operation subgraphs and compound across deeper neural-network workloads.

- **Neural-network workloads.** These cases remain unsolved under the standard evaluation settings: all models obtain 0% Pass@1 and Pass@5, with or without feedback. These workloads compound packing, depth, and approximation decisions across multiple layers, so local fixes are often insufficient to recover a valid CKKS implementation.

Effect of feedback. Feedback improves performance by turning many failures into actionable repair signals. For packing-oriented operators, models can use packing and runtime feedback to revise ciphertext layouts, rotations, and the overall CKKS program structure. For approximation-oriented operators, numerical-error feedback helps models adjust approximation ranges, polynomial degrees, and multiplicative depth. These signals allow models to iteratively move an initially invalid candidate toward a correct CKKS program rather than relying on independent one-shot samples. The compositional results show why feedback is necessary but insufficient: it often repairs exact-square circuits, while approximation-heavy ReLU and softmax compositions remain largely unsolved.

Model-scale comparison. Two Qwen3.5 variants exhibit a substantial within-family gap: with feedback, Qwen3.5-397B-A17B reaches 26.1% Pass@1 and 43.5% Pass@5, whereas Qwen3.5-35B-A3B reaches only 3.9% Pass@1 and 8.7% Pass@5. This gap is consistent with the larger variant being better able to interpret evaluator feedback and revise packing, parameter selection, and approximation design. The experiment does not isolate parameter count from architecture or training data, however, and cross-family rankings do not support a simple scale-only ex-

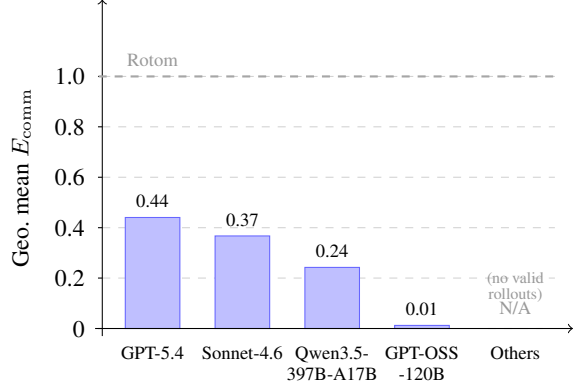


Figure 4: Geometric mean of E_{comm} per model across its valid retained rollouts drawn from the five Rotom-derived cases; models and cases without valid rollouts do not enter the mean. All four models with valid rollouts fall below parity. “Others” (GLM-5.1, DeepSeek-V3.2, Qwen3-Coder-480B-A35B-Instruct, Qwen3.5-35B-A3B) produced no valid retained rollouts on these cases. Rollout-level measurements are in Appendix Table 6.

planation; for example, GPT-OSS-120B achieves higher overall Pass@5 with feedback than GLM-5.1 (39.1% vs. 37.0%). This advantage over GLM-5.1 may reflect greater exposure of GPT-OSS-120B to OpenFHE/CKKS-related material during training, although the undisclosed composition of both models’ training corpora prevents us from testing this hypothesis.

5.2 Communication-Aware Efficiency Analysis

Figure 4 summarizes each model’s geometric mean communication-aware efficiency over its correct retained rollouts drawn from the five Rotom-derived cases. The covered cases and number of rollouts therefore differ by model; a model with no valid rollout has no efficiency score. Rollout-level measurements appear in Appendix (Table 6).

Communication-aware efficiency is important as server-side runtime alone overstates the efficiency of LLM-generated programs. A model may choose a simpler packing strategy that reduces rotations or ring dimension, but this often increases the number of ciphertexts. In a client-server setting, the larger ciphertext footprint adds encoding, encryption, serialization, transfer, and decryption cost. The footprint efficiency factor in Section 4.3 therefore penalizes implementations whose runtime gains are offset by higher communication volume.

The results show a substantial efficiency gap relative to Rotom. GPT-5.4 achieves the highest geo-

metric mean (0.44), followed by Sonnet-4.6 (0.37), Qwen3.5-397B-A17B (0.24), and GPT-OSS-120B (0.01). The remaining four models produce no valid retained rollouts on these benchmarks. This indicates that generating a correct CKKS program is not sufficient for efficiency: LLM-generated kernels often rely on element-wise or row-wise packing, which can make the kernel simple and sometimes fast in isolation but inflates ciphertext count and loses the communication efficiency achieved by compiler-generated layouts.

5.3 Agentic Neural-Network Synthesis

	mlp	bert-attention	lenet5	resnet20	squeezenet
Pass	1/5	1/5	0/5	0/5	0/5

Table 4: Agentic neural-network synthesis results. Sonnet-4.6 with a 90-minute Claude Code working budget per trial, 5 trials per case.

The results above show that all models achieve 0% on neural-network cases under the standard without-feedback and iterative-feedback settings. To probe whether extended iterative debugging can overcome this barrier, we run an agentic experiment: Sonnet-4.6 is given a 90-minute working budget per trial via Claude Code, with full access to the evaluation harness for validation with the same prompt. Each of the five neural-network cases is attempted five times independently. Trials that do not produce a terminal passing evaluation within the 90-minute budget are counted as failures.

Table 4 shows that even with extended agentic interaction, the model solves only the two simplest workloads (MLP and BERT-attention) with a single success each out of five attempts. LeNet-5, ResNet-20, and SqueezeNet remain unsolved. We observe that the agent repeatedly iterates on polynomial approximation parameters, multiplicative depth, and packing layout, and occasionally decomposes the network layer-by-layer to isolate precision errors. Despite these strategies, the agent cannot reliably synthesize a valid CKKS implementation within the time budget: errors in one layer propagate through subsequent layers, and fixing depth or approximation for one component often breaks another. These results suggest that encrypted neural-network synthesis remains beyond the reach of current agentic approaches.

6 Conclusion

We introduced FHE-CodeEval, a 46-case benchmark for evaluating whether LLMs can synthesize

executable OpenFHE/CKKS programs from PyTorch plaintext references. Its four categories span across packing-oriented operators, approximation-oriented operators, compositional subgraphs, and neural-network workloads. FHE-CodeEval systematically probes the core challenges including packing, nonlinear approximation, and level/scale management individually and jointly. We also introduced communication-aware efficiency, which accounts for server-side runtime and ciphertext footprint, and built a two-stage hacking detection pipeline to filter plaintext shortcuts.

Our evaluation of eight proprietary and open-weight models yields three main findings. First, one-shot CKKS code generation remains difficult; even the best model achieves only 7.8% Pass@1 without feedback. Second, execution feedback is a strong differentiator, boosting the best model to 51.7% Pass@1 and 69.6% Pass@5. Third, encrypted neural-network synthesis remains largely unsolved under both standard feedback and extended agentic interaction, indicating that compounding packing, approximation, and depth decisions across layers is beyond current LLM capabilities.

On the five-case compiler-comparable subset, all four models with valid retained rollouts have model-level geometric-mean efficiency below Rotor, primarily due to suboptimal packing strategies.

Limitations

Benchmark coverage. Although FHE-CodeEval covers 46 cases across 4 categories, the benchmark does not exhaustively represent all CKKS workloads. Operations such as scheme-switching are not included. The neural-network tier contains five workloads; larger or more diverse networks (e.g., full transformers with multi-head attention beyond a single layer) remain unexplored. Extending coverage to additional FHE schemes (BFV, BGV) and more complex network topologies is left to future work.

Single library and scheme. All empirical results use CKKS through OpenFHE. Other libraries and compiler-oriented systems, including Microsoft SEAL and Concrete, expose different APIs, supported operations, and optimization abstractions; PALISADE is OpenFHE’s predecessor rather than an independent modern backend (Badawi et al., 2022). Consequently, our quan-

titative conclusions should not be generalized to all FHE ecosystems. Cross-library and cross-scheme evaluation would require backend-specific interfaces, task definitions, and correctness criteria.

Evaluation cost. Each candidate requires full FHE execution for correctness checking, which is expensive: a single neural-network evaluation can take minutes, and the agentic experiment allocates 90 minutes per trial. This cost limits the number of models and sampling budgets we can evaluate, particularly for the neural-network tier and communication-aware efficiency analysis.

Limited simulator fidelity. Our plaintext simulator is intentionally limited to slot-level semantic checking and hacking detection. It does not attempt to replace real OpenFHE execution, because several properties that determine CKKS behavior are difficult to simulate accurately at low cost, including scale evolution, rescaling and modulus-chain consumption, noise growth, rounding and approximation error accumulation, rotation and key-switch behavior, and implementation-specific packing constraints. As a result, we use the simulator only as an auxiliary check for SIMD slot semantics and suspicious non-FHE behavior; final correctness and performance are still determined by executing generated kernels with OpenFHE.

Limited performance baselines. We report this metric only against Rotom-generated baselines on five packing-oriented cases. We do not compare against hand-optimized expert implementations or other FHE compilers (e.g., EVA), as these target different APIs or programming models and would require non-trivial adaptation to our benchmark format. A broader comparison across compilers and hand-tuned code would provide a more complete picture of where LLM-generated FHE programs stand relative to the state of the art.

Acknowledgements

This work was supported by the National Key Research and Development Program of China (2024YFB4505701), the Beijing Natural Science Foundation (L2607013), the National Natural Science Foundation of China (62302479, 62232015), and the China Postdoctoral Science Foundation (2023M733566).

Potential risks. LLMs may generate highly inefficient OpenFHE/CKKS programs that consume

substantial memory and may temporarily make memory-constrained host machines unresponsive. Users of the benchmark should therefore limit per-process memory usage based on the available memory of their host machines.

AI assistants. Claude Code and Codex were used for language polishing, LaTeX preparation and code assistance. The authors reviewed all resulting content and take responsibility for the paper.

Scientific artifacts. We use publicly released scientific artifacts exclusively for non-commercial academic research in accordance with their respective licenses. These include OpenFHE (Badawi et al., 2022), PyTorch (Paszke et al., 2019), and Rotom (Chen et al., 2026). LLM outputs are obtained through hosted API endpoints under their applicable terms of service. We do not redistribute model weights. Our benchmark and evaluation code is publicly available under an open-source license at <https://github.com/qiuchuy/FHE-CodeEval>.

References

- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *CoRR*, abs/2108.07732.
- Ahmad Al Badawi, Jack Bates, Flávio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, R. V. Saraswathy, Kurt Rohloff, Jonathan Saylor, Dmitriy Sponitsky, Matthew Triplett, and 2 others. 2022. [OpenFHE: Open-source fully homomorphic encryption library](#). In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Los Angeles, CA, USA, 7 November 2022*, pages 53–63. ACM.
- Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. [nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data](#). In *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*, pages 3–13. ACM.
- Alon Brutkus, Ran Gilad-Bachrach, and Oren Elisha. 2019. [Low latency privacy preserving inference](#). In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 812–821. PMLR.

- Edward Chen, Fraser Brown, and Wenting Zheng. 2026. [Bridging usability and performance: A tensor compiler for autovectorizing homomorphic encryption](#). In *35th USENIX Security Symposium (USENIX Security 26)*, Baltimore, MD. USENIX Association.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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](#). *CoRR*, abs/2107.03374.
- Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. [Homomorphic encryption for arithmetic of approximate numbers](#). In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer.
- Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. [EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation](#). In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 546–561. ACM.
- Austin Ebel, Karthik Garimella, and Brandon Reagen. 2025. [Orion: A fully homomorphic encryption framework for deep learning](#). In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pages 734–749. ACM.
- Craig Gentry. 2009. [Fully homomorphic encryption using ideal lattices](#). In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM.
- Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. [CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy](#). In *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org.
- Xiaoyu Guo, Minggu Wang, and Jianjun Zhao. 2025. [QuanBench: Benchmarking quantum code generation with large language models](#). In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*, pages 2657–2669. IEEE.
- 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](#). *CoRR*, abs/2507.15224.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sánchez. 2024. [A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption](#). *Proc. ACM Program. Lang.*, 8(PLDI):126–150.
- Mayank Kumar, Jiaqi Xue, Mengxin Zheng, and Qian Lou. 2026. [FHE-Coder: Benchmarking secure agentic code generation for fully homomorphic encryption](#). In *International Conference on Learning Representations*, volume 2026, pages 81257–81273.
- Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. [DS-1000: A natural and reliable benchmark for data science code generation](#). In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.
- Long Li, Jianxin Lai, Peng Yuan, Tianxiang Sui, Yan Liu, Qing Zhu, Xiaojing Zhang, Linjie Xiao, Wenguang Chen, and Jingling Xue. 2025. [ANT-ACE: an FHE compiler framework for automating neural network inference](#). In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2025, Las Vegas, NV, USA, March 1-5, 2025*, pages 193–208. ACM.
- Mingjie Liu, Nathaniel Ross Pinckney, Bruce Khailany, and Haoxing Ren. 2023. [Invited paper: VerilogEval: Evaluating large language models for Verilog code generation](#). In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*, pages 1–8. IEEE.
- Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. [KernelBench: Can LLMs write efficient GPU kernels?](#) In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net.
- Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca

Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. [PyTorch: An imperative style, high-performance deep learning library](#). In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

Abdessamed Seddiki, Arab Mohammed, Zakaria Hebbal, Aimad Chabounia, Eduardo Chielle, Karima Benatchba, Challal Yacine, Djamel Eddine Menacer, Michail Maniatakos, and Riyadh Baghdadi. 2026. [CHEHAB: automatic compiler code optimization for fully homomorphic encryption](#). In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction, CC 2026, Sydney, NSW, Australia, 31 January 2026 - 1 February 2026*, pages 81–93. ACM.

Nuo Xu, Zhaoting Gong, Ran Ran, Jinwei Tang, Wujie Wen, and Caiwen Ding. 2025. [FHE-Agent: Automating CKKS configuration for practical encrypted inference via an LLM-guided agentic framework](#). *CoRR*, abs/2511.18653.

A Prompt Template

Each benchmark case is converted into a self-contained prompt. The prompt gives the model the plaintext semantics, the input contract, the required OpenFHE interface, the response format, and a small in-context example that demonstrates the expected file structure. Listing 1 shows the template used for the initial generation attempt. In the with-feedback setting, evaluator feedback is appended after the previous failed program and before the next answer.

Listing 1: Prompt template for one benchmark case.

```
You are an expert in Fully Homomorphic Encryption (FHE).
Your task is to convert a PyTorch kernel into a functionally equivalent implementation using the OpenFHE Python bindings with the CKKS scheme.

You should implement the original plaintext program with HE primitives provided by OpenFHE. Do not use PyTorch or NumPy operations as fallback solutions in fhe_kernel.

Case: {category}/{case_name}
Task: {task_description}

## Input Specification

{n_sample} random test inputs in {input_range}
Accuracy metric: torch.allclose (atol=0.01, rtol=0.01)
```

```
- {input_name}: torch.Tensor shape={shape}, dtype={dtype}, range={range} [{role}]
...

### Notes

- All float tensors use dtype=torch.float32 unless otherwise stated.
- Independent random test inputs will be generated at evaluation.

### Observed Activation Ranges (if available)

{profiling_ranges}

## Reference Source

```python
{pytorch_reference}
```

## Response Format

Return ONLY a Python file containing four functions plus any module-level constants needed, such as MULTIPLICATIVE_DEPTH, RING_DIM, N_SLOTS, ROTATION_INDICES, OUTPUT_SHAPE, and SCALING_MOD_SIZE.

```python
Module-level constants derived from the circuit and input shapes
MULTIPLICATIVE_DEPTH = ... # number of multiplicative levels needed
SCALING_MOD_SIZE = ... # bits per rescaling modulus
RING_DIM = ... # power-of-two ring dimension
N_SLOTS = ... # CKKS batch size
ROTATION_INDICES = [...] # all rotations used in fhe_kernel
OUTPUT_SHAPE = [...] # PyTorch output shape

def make_context():
 """Crypto context init: CcParamsCKKSRSNS, GenCryptoContext, KeyGen, EvalMultKeyGen, EvalRotateKeyGen. Takes no arguments; all parameters come from module-level constants. Returns (cc, keys)."""
 parameters = openfhe.CcParamsCKKSRSNS()
 parameters.SetMultiplicativeDepth(MULTIPLICATIVE_DEPTH)
 parameters.SetScalingModSize(SCALING_MOD_SIZE)
 parameters.SetScalingTechnique(openfhe.ScalingTechnique.FIXEDMANUAL)
 parameters.SetSecurityLevel(openfhe.HEStd_128_classic)
 parameters.SetRingDim(RING_DIM)
 parameters.SetBatchSize(N_SLOTS)
 cc = openfhe.GenCryptoContext(parameters)
 cc.Enable(openfhe.PKESchemeFeature.PKE)
 cc.Enable(openfhe.PKESchemeFeature.KEYSWITCH)
 cc.Enable(openfhe.PKESchemeFeature.LEVELED_SHE)
```

```

cc.Enable(openfhe.PKESchemeFeature.
ADVANCEDSHE)
keys = cc.KeyGen()
cc.EvalMultKeyGen(keys.secretKey)
cc.EvalRotateKeyGen(keys.secretKey,
ROTATION_INDICES)
return cc, keys

def encrypt(cc, keys, arg1, arg2, ...):
 """Encode and encrypt inputs. The remaining
 arguments match the
 torch_kernel signature. Ciphertext-role
 inputs must be
 encrypted with cc.Encrypt; model weights and
 biases must be encoded as
 plaintexts with cc.MakeCKKSPackedPlaintext.
 Do not call model(x),
 model.forward(x), or run any part of a
 neural network forward pass
 inside encrypt()."""

def fhe_kernel(cc, keys, enc_inputs):
 """Core FHE computation only. This is the
 only wall-clock timed
 function. Do not perform crypto setup,
 encoding, encryption, decryption,
 Plaintext.GetRealPackedValue(), PyTorch, or
 NumPy computation here.
 Return openfhe.Ciphertext or list[openfhe.
 Ciphertext], never a
 torch.Tensor."""

def decrypt(cc, keys, ct_out):
 """Decrypt ct_out and return a torch.Tensor
 with the same shape and
 values, within tolerance, as the PyTorch
 reference output."""
...

- make_context() takes no arguments; put all
 circuit parameters as
 module-level constants.
- encrypt(), fhe_kernel(), and decrypt() receive
 cc and keys as their first
 two arguments.
- fhe_kernel(cc, keys, enc_inputs) is the only
 function that is wall-clock
 timed.
- decrypt() must return a torch.Tensor with the
 correct shape and dtype.
- All four functions must be defined at module
 level.
- Do not include test code, if __name__ == "
 __main__" blocks, or any code that
 calls these functions directly.

Example

{example_problem_and_solution}

Requirements

1. Declare module-level constants:
 MULTIPLICATIVE_DEPTH, RING_DIM,
 N_SLOTS, ROTATION_INDICES, OUTPUT_SHAPE,
 SCALING_MOD_SIZE.
2. Define make_context(), encrypt(...),
 fhe_kernel(...), and decrypt(...).
3. Use ScalingTechnique.FIXEDMANUAL and
 HEStd_128_classic.

```

4. Do not hard-code evaluator inputs or outputs.
5. Do not use PyTorch or NumPy for encrypted computation.

## B Common Errors in LLM-Generated OpenFHE Code

Table 5 categorizes the most frequent errors we observe in LLM-generated OpenFHE programs into three groups.

**Syntax errors** arise from incorrect API usage—for example, invoking nonexistent class names or enum values. These errors are easily detected by the Python interpreter and typically resolved within one feedback round once the model receives the resulting `NameError` or `AttributeError`.

**Configuration errors** involve incorrect cryptographic parameter choices: packing vectors that exceed the CKKS slot bound, setting multiplicative depth too low for the required circuit, choosing a ring dimension that violates security constraints for the selected modulus chain, or omitting feature flags needed for advanced operations such as Chebyshev evaluation. These errors reflect a misunderstanding of the relationship between circuit structure and cryptographic context setup, and often require multiple feedback rounds because fixing one parameter (e.g., increasing depth) may trigger a downstream failure (e.g., requiring a larger ring dimension for security compliance).

**Functional errors** produce programs that compile and execute without exceptions but return numerically incorrect results—typically because the polynomial approximation degree is insufficient for the specified input range. These are the hardest to repair, as the feedback provides only aggregate error statistics (max/avg absolute error) without directly indicating whether the model should increase the polynomial degree, narrow the approximation interval, or restructure the computation.

## C Detailed Communication-Aware Efficiency Results

Table 6 reports the rollout-level measurements used to construct Figure 4. The table includes all retained valid rollouts after correctness and hacking checks. For each row, we abbreviate the generated-program quantities as  $T_G = T_{ij}$ ,  $N_G = N_{ij}$ , and  $C_G = C_{ij}$ , and the Rotom quantities as  $T_R = T_i^R$ ,  $N_R = N_i^R$ , and  $C_R = C_i^R$ . We report raw runtime speedup  $S_t = T_R/T_G$ , footprint efficiency

Description	Incorrect Code	Corrected Code	Feedback
<b>A. Syntax Errors — API misuse or undeclared identifiers</b>			
Missing namespace or context qualifier	<code>params = CCParamsCKKS()</code>	<code>params = CCParamsCKKSRSNS()</code>	<code>NameError: name 'CCParamsCKKS' is not defined</code>
Wrong API usage	<code>cc.Enable(LEVELED)</code>	<code>cc.Enable(LEVELED\$HE)</code>	<code>AttributeError: type object 'openfhe.PKESchemeFeature' has no attribute 'LEVELED'</code>
<b>B. Configuration Errors — incorrect parameters or FHE context setup</b>			
Incorrect Packing	<code>x = [x_list[y_idx]] + [0.0] * (cc.GetRingDimension() - 1)</code> <code>x_plain = cc.MakeCKKSPackedPlaintext(x)</code>	<code>x = [x_list[y_idx]] + [0.0] * (cc.GetRingDimension() // 2 - 1)</code> <code>x_plain = cc.MakeCKKSPackedPlaintext(x)</code>	The size [32768] of the vector with values should not be greater than ringDim/2 [16384] if the scheme is CKKS.
Multiplicative depth too small	<code>params.SetMultiplicativeDepth(1)</code>	<code>params.SetMultiplicativeDepth(3)</code>	The current multiplicative depth (1) is insufficient. Increase it.
Under-sized ring dimension	<code>params.SetRingDim(8192)</code>	<code>params.SetRingDim(32768)</code>	The specified ring dimension (8192) does not comply with HE standards recommendations (32768).
Missing feature enable	<code>cc.EvalChebyshevSeries(...)</code>	<code>cc.Enable(ADVANCED\$HE)</code> <code>cc.EvalChebyshevSeries(...)</code>	<code>EvalChebyshevSeries</code> operation has not been enabled. <code>Enable(ADVANCED\$HE)</code> must be called to enable it.
<b>C. Functional Errors — incorrect or imprecise approximation arithmetic semantics</b>			
Unmatched results caused by inaccurate polynomial approximation	<code>degree = 3</code> <code>cc.EvalChebyshevFunction(..., degree, ...)</code>	<code>degree = 8</code> <code>cc.EvalChebyshevFunction(..., degree, ...)</code>	Sample 5/5 failed: <code>max_abs_err = ...</code> , <code>avg_abs_err = ...</code>

Table 5: Common errors in LLM-generated OpenFHE code and corresponding feedback.

$\eta_F = (N_R C_R)/(N_G C_G)$ , and communication-aware efficiency  $E_{\text{comm}} = S_t \eta_F$ . No model produced a retained valid rollout for double-matmul, so that fifth compiler-comparable case contributes no row.

Of the 30 valid rollouts in Table 6, 24 (80%) achieve  $E_{\text{comm}} < 1$ , indicating that the majority of LLM-generated programs do not match the compiler baseline under the combined compute-communication trade-off. The six rollouts that exceed parity ( $E_{\text{comm}} > 1$ ) arise from two mechanisms. First, the LLM selects more favorable cryptographic hyperparameters—notably a smaller ring dimension (e.g.,  $N = 16384$  vs. Rotom’s default  $N = 32768$ )—while still satisfying OpenFHE’s 128-bit security check, which simultaneously reduces per-ciphertext cost and kernel latency. Second, the LLM discovers a packing layout that achieves a better trade-off between ciphertext footprint and server-side kernel runtime: for example, in the convolution case, packing the entire input into a single ciphertext ( $C_G = 1$ ) eliminates inter-ciphertext communication at the expense of more rotations. Whether these strategies generalize depends on the actual network topology and computational workload: a packing that reduces ciphertext count for a small convolution may become infeasible or slower for larger tensor dimensions where the rotation overhead dominates.

## D Per-Case Correctness Heatmap

Figure 5 shows the number of correct final rollouts (out of 5) for each model–case pair across all 46 cases under the with-feedback setting. Several patterns emerge: (1) a small set of packing-oriented

cases (dot-product, l2-distance, sum, mean) are solved by nearly all models, suggesting these require only basic CKKS operations; (2) cases involving complex tensor layout (double-matmul, ttm, sobel) remain difficult for all models; (3) Sonnet-4.6 uniquely solves a broad range of approximation-oriented cases, while other models’ sparse successes are limited to exp, sigmoid, sqrt, inverse, log, and relu; (4) compositional successes concentrate on square-based cases, with Sonnet-4.6 uniquely solving sum-relu; and (5) the neural-network tier is uniformly zero across all models under this evaluation setting.

## E Correctness Progression with Feedback Rounds

Figure 6 shows the cumulative number of the 46 cases solved by each feedback round. Round 0 corresponds to the initial generation without feedback; each subsequent round provides one additional execution-based diagnostic message. Most gains occur in the first three rounds, when models repair actionable packing, parameter, and interface errors. Sonnet-4.6 grows from 5 solved cases at round 0 to 29 by round 3 and 32 by round 5. Other models plateau earlier, indicating that the remaining failures require deeper changes to packing or approximation design rather than local repairs.

## F PyTorch/NumPy Bypass Rate

Figure 7 reports how often the hacking detector flags generated programs that bypass FHE execution through plaintext computation.

Bypass behavior is concentrated in two mod-

Case	Model	$T_G$	$N_G$	$C_G$	$T_R$	$N_R$	$C_R$	$S_t$	$\eta_F$	$E_{comm}$
matmul	Qwen3.5-397B-A17B	47.9	32768	64	670.7	32768	1	14.002	0.016	0.219
matmul	Qwen3.5-397B-A17B	63.2	32768	64	670.7	32768	1	10.612	0.016	0.166
matmul	Qwen3.5-397B-A17B	86.7	32768	64	670.7	32768	1	7.736	0.016	0.121
matmul	Sonnet-4.6	5394.8	16384	1	670.7	32768	1	0.124	2.000	0.249
matmul	GPT-OSS-120B	9465.9	16384	1	670.7	32768	1	0.071	2.000	0.142
matmul	GPT-OSS-120B	10681.0	16384	64	670.7	32768	1	0.063	0.031	0.002
matmul	GPT-5.4	14400.2	32768	1	670.7	32768	1	0.047	1.000	0.047
matmul	Sonnet-4.6	15294.4	32768	1	670.7	32768	1	0.044	1.000	0.044
matmul	Sonnet-4.6	34060.6	32768	1	670.7	32768	1	0.020	1.000	0.020
matmul	Sonnet-4.6	95225.5	32768	1	670.7	32768	1	0.007	1.000	0.007
matmul	GPT-OSS-120B	235642.3	8192	8192	670.7	32768	1	0.003	< 0.001	< 0.001
logreg-matvecmul	GPT-5.4	1968.8	16384	1	3572.9	32768	1	1.815	2.000	3.630
logreg-matvecmul	Sonnet-4.6	3366.0	16384	1	3572.9	32768	1	1.061	2.000	2.123
logreg-matvecmul	Sonnet-4.6	20835.1	16384	1	3572.9	32768	1	0.171	2.000	0.343
logreg-matvecmul	Sonnet-4.6	22302.2	16384	1	3572.9	32768	1	0.160	2.000	0.320
logreg-matvecmul	Qwen3.5-397B-A17B	23373.5	16384	1	3572.9	32768	1	0.153	2.000	0.306
logreg-matvecmul	Qwen3.5-397B-A17B	26004.1	16384	1	3572.9	32768	1	0.137	2.000	0.275
ttm	Qwen3.5-397B-A17B	15261.6	524288	128	746383.4	32768	17	48.906	0.008	0.406
ttm	Qwen3.5-397B-A17B	18625.6	8192	4160	746383.4	32768	17	40.073	0.016	0.655
ttm	Qwen3.5-397B-A17B	50307.9	16384	4160	746383.4	32768	17	14.836	0.008	0.121
convolution	Sonnet-4.6	169.1	16384	1	84.3	16384	9	0.499	9.000	4.487
convolution	Sonnet-4.6	221.9	16384	1	84.3	16384	9	0.380	9.000	3.419
convolution	Sonnet-4.6	230.1	16384	1	84.3	16384	9	0.366	9.000	3.297
convolution	GPT-5.4	250.9	32768	1	84.3	16384	9	0.336	4.500	1.512
convolution	GPT-OSS-120B	306.1	8192	8	84.3	16384	9	0.275	2.250	0.620
convolution	GPT-OSS-120B	747.3	16384	8	84.3	16384	9	0.113	1.125	0.127
convolution	Sonnet-4.6	830.7	16384	1	84.3	16384	9	0.101	9.000	0.913
convolution	GPT-OSS-120B	836.2	16384	8	84.3	16384	9	0.101	1.125	0.113
convolution	GPT-5.4	1314.1	32768	1	84.3	16384	9	0.064	4.500	0.289
convolution	GPT-5.4	1688.2	32768	1	84.3	16384	9	0.050	4.500	0.225

Table 6: Detailed communication-aware efficiency inputs for all valid rollouts on the five Rotom-derived cases. Latencies are in milliseconds.

els: GPT-5.4 (12.2%) and GPT-OSS-120B (8.7%).

This concentration underscores the necessity of execution-based hacking detection: without it, these bypasses would inflate the reported Pass@ $k$  of the affected models.

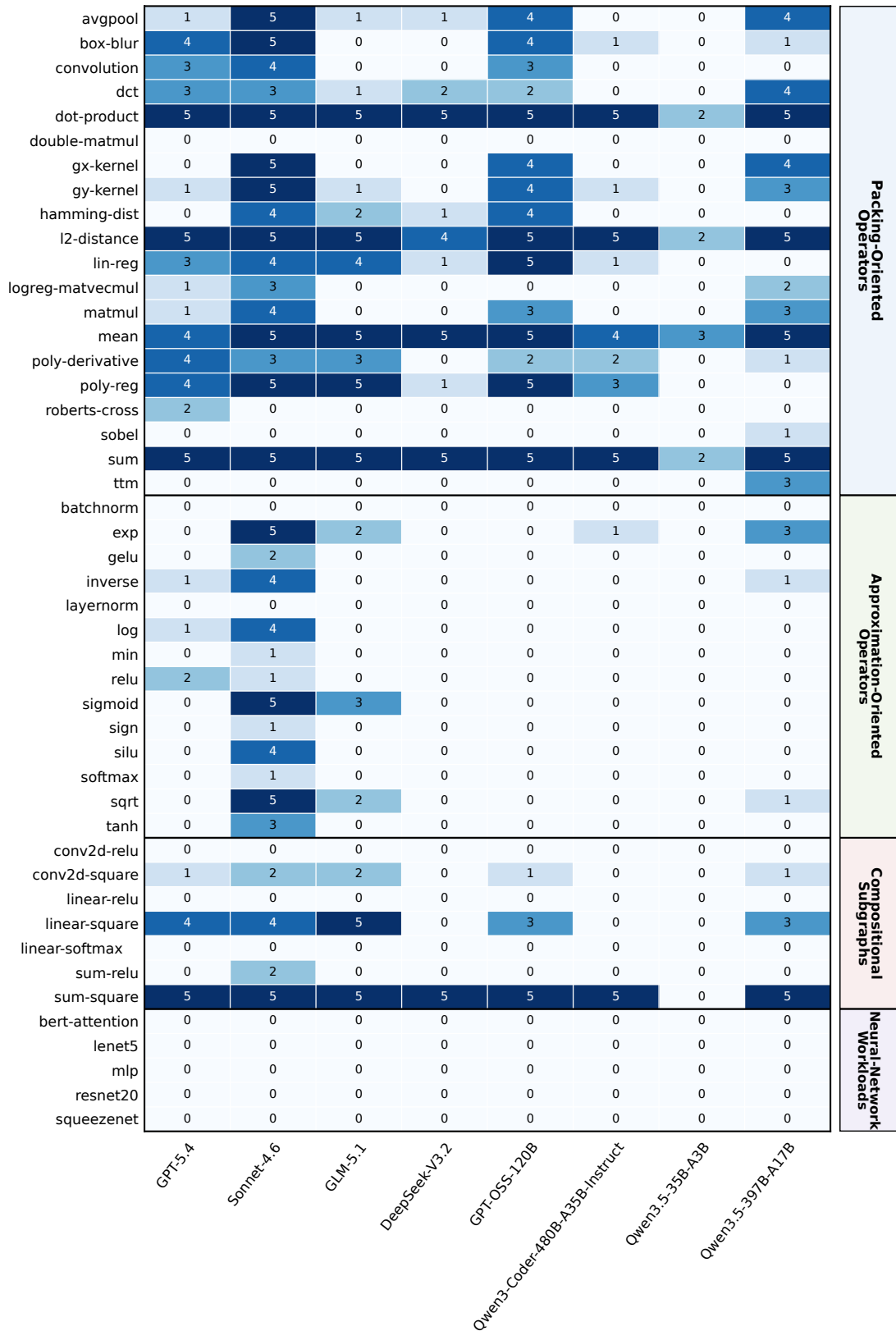


Figure 5: Per-case correctness heatmap for all 46 cases. Each cell shows the number of correct final rollouts (out of 5) for a given model–case pair under the with-feedback setting. Cases are grouped by category (packing-oriented operators, approximation-oriented operators, compositional subgraphs, and neural-network workloads) separated by horizontal lines.

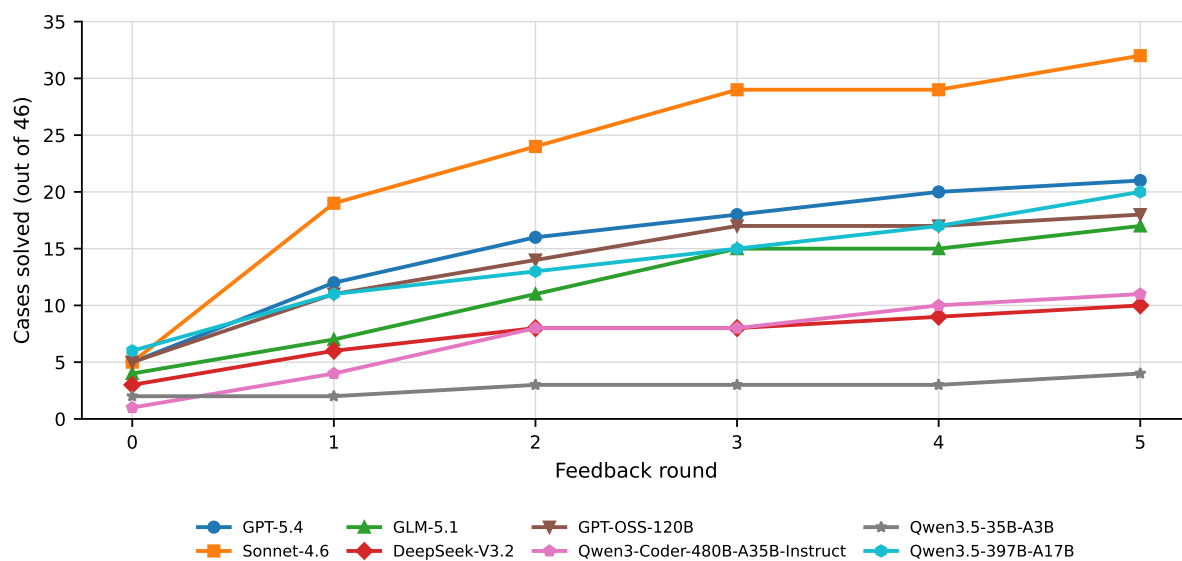


Figure 6: Cumulative number of the 46 cases solved by feedback round. Round 0 is the initial generation without feedback.

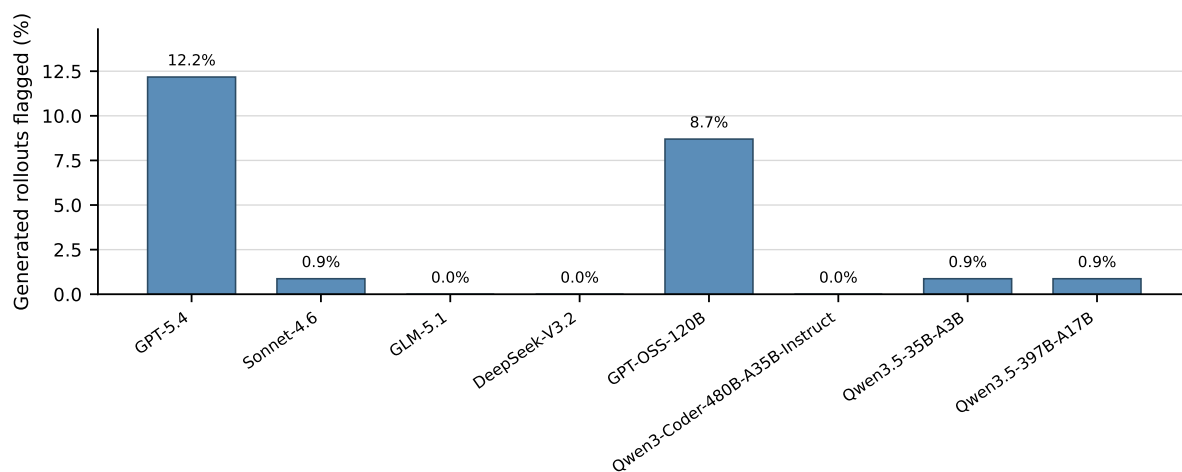


Figure 7: Fraction of final generated rollouts on all 46 cases that pass functional validation but are flagged as PyTorch/NumPy bypasses by the hacking detector.