Agnostics: Learning to Code in Any Programming Language via Reinforcement with a Universal Learning Environment

Aleksander Boruch-Gruszecki[1], **Yangtian Zi**[1], **Zixuan Wu**[1], **Tejas Oberoi**[2], **Carolyn Jane Anderson**[3], **Joydeep Biswas**[2], **Arjun Guha**[1]

[1]Northeastern University [2]University of Texas [3]Wellesley College

research@abgru.me, zi.ya@northeastern.edu, zi.wu@northeastern.edu, tejasoberoi@utexas.edu, carolyn.anderson@wellesley.edu, joydeepb@cs.utexas.edu, a.guha@northeastern.edu

Abstract

Large language models (LLMs) already excel at writing code in *high-resource* languages such as Python and JavaScript, yet stumble on *low-resource* languages that remain essential to science and engineering. Besides the obvious shortage of pre-training data, post-training itself is a bottleneck: every new language seems to require new datasets, test harnesses, and reinforcement-learning (RL) infrastructure.

We introduce Agnostics, a *language-agnostic* post-training pipeline that eliminates this per-language engineering. The key idea is to judge code solely by its externally observable behavior, so a single verifier can test solutions written in *any* language. Concretely, we (i) use an LLM to rewrite existing unit-test datasets into an I/O format, (ii) supply a short configuration that tells the verifier how to compile and run a target language, and (iii) apply reinforcement learning with verifiable rewards (RLVR) in a robust code execution environment.

Applied to five low-resource languages—Lua, Julia, R, OCaml, and Fortran—Agnostics (1) improves Qwen-3 4B to performance that rivals other 16B–70B open-weight models; (2) scales cleanly to larger and diverse model families (Qwen-3 8B, DeepSeek Coder 6.7B Instruct, Phi 4 Mini); and (3) for $\leq$16B parameter models, sets new state-of-the-art pass@1 results on MultiPL-E and a new multi-language version LiveCodeBench that we introduce.

We will release the language-agnostic training datasets (Ag-MBPP-X, Ag-Codeforces-X, Ag-LiveCodeBench-X), training code, and ready-to-use configurations, making RL post-training in *any* programming language as simple as editing a short YAML file.

1 Introduction

Large language models (LLMs) are remarkably good at programming tasks, especially when coding in *high-resource programming languages* such as Python and JavaScript. Their proficiency in *low-resource programming languages*, such as Fortran, Julia, and others, is significantly more limited. This gap appears both on benchmarks (Cassano et al. 2023) and in popular discourse. However, many low-resource languages are widely used in computational science (e.g., Fortran), medicine (e.g., Mumps), engineering (e.g., Matlab), and others sectors. For programmers in these sectors to be able to take full advantage of LLMs, we need methods to make LLMs better at the languages that they use.

The capability gap between high-resource and low-resource languages occurs for two reasons. The first one is that there is *vastly* more training data for some programming languages. For example, The Stack V2 (Lozhkov et al. 2024a), the largest public training corpus of code, has $\approx$200GB of Python but only $\approx$2GB of Julia and Fortran. Thus pretraining on code makes models significantly better at Python. A more subtle reason is the availability of post-training datasets and techniques. Contemporary LLMs are developed with an extensive post-training process that relies on (a) high-quality curated data for supervised fine-tuning, and (b) carefully designed environments for reinforcement learning, which must be able to execute and verify model-generated solutions. Both of these require significant human expertise, which is hard to find for low-resource languages.

OUr goal in this paper is to facilitate post-training LLMs on low-resource programming languages, working towards closing the resource gap. Our key idea is that for a significant class of programming tasks, correctness can be specified as a property not of individual functions or code snippets, but of *the entire program's observable behavior* (e.g., I/O). Furthermore, if its correctness can be tested with another program, we can use such a *verifier* together with appropriate problems and test cases to make *a universal reinforcement learning environment which can be instantiated for any programming language*. In fact, the language used to implement the verifier and the one being learned are entirely independent. This approach matches how some existing post-training datasets are formulated (even though they are intended for Python/C++). We can also use LLMs to reformulate language-specific datasets into this format.

Based on this insight, our approach, Agnostics, works as follows (Figure 1). 1) We use an LLM to reformulate language-specific datasets into our standard language-agnostic format. 2) We generate prompts and instantiate the verifier to target a particular language, with a small (4-5 line) configuration. 3) We apply reinforcement learning with verified rewards (RLVR) using a robust, language-agnostic execution sandbox that we develop. 4) The result is a model specialized to the target language. Agnostics particularly excels at finetuning models for low-resource languages, as it does not rely on high-quality datasets specific to a particular language.

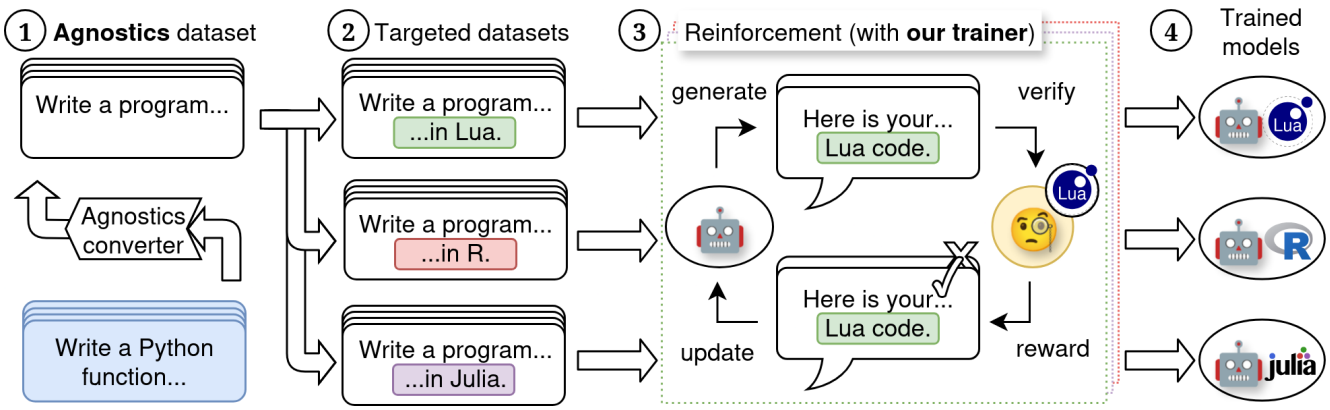

Figure 1: **Overview: Agnostics Data Preparation and Training. (1)** We reformulate existing coding datasets to our format. **(2)** We adapt the language-agnostic datasets to a particular programming language. **(3, 4)** We reinforce coding via Group Relative Policy Optimization (Shao et al. 2024; DeepSeek-AI et al. 2025), verifying the programs in our code execution sandbox.

Language	%	Language	%
JavaScript	17.04 %	Lua	0.53 %
Java	8.38 %	R	0.35 %
C++	5.41 %	Fortran	0.07 %
Python	3.56 %	Julia	0.10 %

Figure 2: High-resource languages (left) dominate the Stack v2. We are interested in low-resource languages (right).

Contributions We make the following contributions.

1. Agnostics, a post-training pipeline for coding in arbitrary programming languages, including low-resource ones;
2. The best-performing open-weights models for Lua, R, Julia, OCaml and Fortran at the $\leq$16B parameter scale;
3. 3 Agnostics datasets Ag-MBPP-X, Ag-Codeforces-X, and Ag-LiveCodeBench-X, based on MBPP (Austin et al. 2021), Open-R1 Codeforces (Penedo et al. 2025) and LiveCodeBench (Jain et al. 2024a) respectively.
4. A small and highly-optimized Agnostics training framework, implementing GRPO and including a parallel code execution sandbox, sampling, rewards computation, and model back-propagation.

Artifacts and code necessary to verify the results we present are listed at agnostics.abgru.me. The training framework will be publicly released together with the formal publication of this work.

2 Background and Related Work

Data Scarcity for Low-Resource Languages LLM have been pretraining on code for several years, both because LLMs are widely used for practical programming tasks, and because pretraining on code improves their general reasoning abilities (Ma et al. 2023). In addition to general-purpose LLMs, there are also code-specialized models that are either trained exclusively on code (e.g., Xu et al. (2022); Allal et al. (2023)) or trained on code starting from a general-purpose model checkpoint (e.g., Rozière et al. (2024)).

However, the publicly available pretraining data for code is heavily skewed toward a handful of programming languages. For example, consider The Stack V2 (Lozhkov et al. 2024b), which is the largest public code pretraining dataset, with code from GitHub and dozens of other sources. The Stack V2 dominated by a handful of programming languages: just 10 of 619 languages account for over 90% of the dataset. We are interested in developing models for low-resource languages that each account for less than 0.5% of the publicly available code (Figure 2).

We can imagine working around this data scarcity in a few ways. However, previous work shows that up-sampling low-resource languages during pretraining only leads to small benchmark improvements (Orlanski et al. 2023), while fine-tuning on the pre-training data for low-resource languages has negligible impact (Cassano et al. 2024). Thus, it is not clear how to make further gains from existing natural data.

Synthetic Data for Low-Resource Languages If natural data is not available for a task, it is possible to use LLMs to generate synthetic fine-tuning data (Wang et al. 2023), and there are many techniques for building code-centric supervised fine-tuning datasets (e.g. Luo et al. (2023); Wei et al. (2024c,b)) that work remarkably for Python. Although these approaches could in principle be applied to any programming language, Cassano et al. (2024) show that without distillation or verification (e.g., see Hu et al. (2025); Wei et al. (2024a)), synthetic tasks and code for low-resource programming languages are low-quality, and models fine-tuned on them perform poorly.

MultiPL-T (Cassano et al. 2024), similar to TransCoder-ST (Roziere et al. 2021) and CMTrans (Xie et al. 2024), couples synthetic data generation with verification using rejection sampling: it generates up n candidate programs in a target, low-resource language and only fine-tune models on generations that pass hidden unit tests that it translates from Python. However, the MultiPL-T approach has two significant limitations. (1) For the verifier to not reject all samples, the model must be able to generate a working program within n attempts. In MultiPL-T, $\approx$30% of prompts produce

a working program for $n \in \{50, 100\}$ attempts, and the rest are discarded. We train on much harder problems, and estimate that rejection sampling would require an order of magnitude more resources to achieve a comparable acceptance rate (§A). (2) For each low-resource language of interest, MultiPL-T requires the user to write a little compiler to translate test cases and function signatures from Python to the target language. MultiPL-T only supports a limited set of built-in Python types (e.g., no classes) and it dictates that all Python types and values must faithfully map to the target language. However, depending on the problem and language, the natural way to represent data may not map cleanly to Python types. This can lead to peculiar, unidiomatic translations that require deep language expertise to get right. The Agnostics approach is significantly easier to use than MultiPL-T, and only requires the user to know how to compile and run a program in the target language from the shell.

Reinforcement Learning on Coding Tasks DeepSeek R1 (DeepSeek-AI et al. 2025) popularized RL on LLMs with rule-based rewards, instead of learned reward models. R1 reports applying RL to coding tasks but does not disclose further dataset details. A number of papers apply RL to the NL to code task (Zeng et al. 2025; Gehring et al. 2024; Jain et al. 2025). These techniques, which target Python, show that reinforcement learning can improve LLM capabilities beyond what is possible with just supervised fine-tuning.

However, the key benefit of reinforcement learning is that it can train a model to do tasks for which high-quality data for supervised fine-tuning is unavailable. There are recent examples of using RL for code optimization (Du et al. 2025; Nichols et al. 2024), resolve GitHub, issue resolution (Wei et al. 2025), and iterative development (Zhou et al. 2025). These papers target tasks in high-resource languages—C++, Java, and Python—whereas Agnostics targets several low-resource languages.

3 The Agnostics Approach

Our approach comprises (1) a *data preparation* stage which reformulates language-specific programming tasks to be language-agnostic, and retargets language-agnostic datasets to a programming language of interest (1, 2 in Figure 1); and (2) the *training* stage which uses the GRPO algorithm with an efficient, language-agnostic verification framework (3, 4 in Figure 1). Our central idea is to have tasks ask for programs which behave in a particular way. The test cases are samples of this behavior, and a verifier program can check if a solution behaves according to the sample. In this paper, we limited ourselves to working with tasks asking for programs which read data from the standard input, compute a unique answer, and write it to the standard output. Hence, the datasets we prepared share the same verifier.

3.1 Dataset Preparation

Some datasets, like Open-R1 Codeforces (Penedo et al. 2025), already define tasks in the desired I/O style. More commonly, however, code datasets provide a set of unit tests. Figure 3a shows a representative item from MBPP: it has a

```
# Write a python function to identify non-
# prime numbers.
def is_not_prime(n):
    ...
```

```
assert is_not_prime(2) == False
assert is_not_prime(10) == True
```

(a) A task prompt and associated tests (in gray) from MBPP.

Instruction: Given an integer N ($N \geq 2$), determine whether it is a non-prime number. Output 'True' if the number is non-prime, 'False' otherwise. Input format: a single integer N ($N \geq 2$). Output format: a single line containing 'True' or 'False'.

Input	Output
2	`False`
10	`True`

(b) The task and tests reformulated for Agnostics.

Figure 3: For dataset preparation, we use an LLM to reformulate fine-tuning datasets with language-specific prompts and tests (above) into equivalent language-agnostic programming tasks.

natural language problem description and a Python function signature that comprise the prompt, and a suite of tests used to test model-generated code. These datasets can be easily translated into the I/O format.

To make such problems language-agnostic and compatible with our verifier, we prompt an LLM to reformulate each task so that the program communicates exclusively via plain-text standard in and standard out. We ask the model to spell out concrete I/O conventions—number of decimal places, newline versus comma separators, ordering of values, and so on—so that the expected behavior is unambiguous. Figure 3b shows the reformulated example. §B has the instruction we use to reformulate MBPP; other datasets might require small changes to the prompt.

3.2 Programming Language Preparation

To prepare a new language, we author a small configuration file that serves two purposes. First, it defines a *prompt prefix* (prepended to each problem by the trainer) which instructs the model to produce code in the target language. Second, the configuration file specifies the shell commands to install the language toolchain and run code. Figure 4 shows the configuration for the R language, with an atypically large prompt due to quirks of its I/O APIs[1].

A good prefix should help avoid the most common errors, but determining which errors are common can be time-consuming. In our experience, if the model starts from $\geq 5\%$ accuracy, then the prefix only needs to say that the solution must be in language L. (This varies with GRPO group size,

[1]There are three ways to run R, three I/O APIs, and only one portable way to read from standard in.

```
install: apt-get install -y r-cran-tidyverse
filename: snippet.R
execute: Rscript snippet.R
prompt: |
  Use R version 4.
  Use `readLines(con = file("stdin"))` to
  read input from stdin. Optionally, use
  the `n` argument to read the first `n`
  lines. For example:
  ```r
  input <- readLines(con = file("stdin"), n
    = 1)
  n <- as.integer(input)
  print(n) # print the first line of input
  ```
  Also, use `cat` to print output to stdout.
  For example:
  ```r
  cat(n)
  ```
  Please do not use `print` to print output.
```

Figure 4: An Agnostics configuration snippet for R.

see §3.3.) On the other hand, when starting from near-zero accuracy, a prefix like in Figure 4 can help prevent common mistakes. For OCaml and Fortran, we asked a capable LLM (OpenAI o3) for advice based on several faulty generations.

> *What follows are several Fortran programs. You'll see that most of them are wrong. Read them carefully and identify the Fortran programming mistakes that I'm making. Ignore algorithmic mistakes, and focus on my misconceptions about Fortran. Come up with advice on how I should program Fortran correctly. Distill this advice into 10-20 sentences.*

We used the resulting instructions verbatim (§C), but only for training and *not* to evaluate models on OCaml or Fortran.

3.3 Trainer and Code Execution

The Agnostics trainer uses the Group-Relative Policy Optimization (GRPO) reinforcement learning algorithm (Shao et al. 2024), with verifiable rewards (DeepSeek-AI et al. 2025), and some common tweaks to further improve its efficiency (Yu et al. 2025). We couple the algorithm with a language-agnostic code execution framework that is designed to be robust and efficient.

Trainer The trainer instantiates the GRPO algorithm as follows. Let $(x, \{(in_k, out_k)\}_{k=1}^K) \sim \mathcal{D}$ be a dataset of language-agnostics tasks, where x is the task prompt and $\{(in_k, out_k)\}_{k=1}^K$ is the set of I/O examples. Let L be a language configuration L (e.g., Figure 4). From the behavior policy $\pi_{\theta_{\text{old}}}$, we sample a group G of candidate responses

$$\{y_i\}_{i=1}^G \sim \pi_{\theta_{\text{old}}}(y_i \mid L.\mathsf{prompt}, x).$$

We assign each candidate a reward R_i, with $R_i = 1$ if the execution environment (described later) verifies that the extracted program's behavior is consistent with the I/O examples (in_k, out_k), and $R_i = 0$ otherwise.

We turn group rewards into sequence-level advantages

$$\hat{A}_i = \frac{R_i - \text{mean}(\{R_j\}_{j=1}^G)}{\text{std}(\{R_j\}_{j=1}^G)}.$$

We then update the policy with the objective

$$\mathcal{L}_{\text{GRPO}}(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\Big(r_{i,t}(\theta)\hat{A}_i, c_{i,t}(\theta)\hat{A}_i\Big),$$

where

$$c_{i,t}(\theta) = \text{clip}\big(r_{i,t}(\theta), 1-\varepsilon, 1+\varepsilon\big),$$
$$r_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid L.\mathsf{prompt}, x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid L.\mathsf{prompt}, x, y_{i,<t})}.$$

We omit the KL-divergence term, similar to Yu et al. (2025).

We considered and decided against a reward for a partially-correct answer. We tried to reward the model for code which runs without errors but produces wrong output or for code which only passes the public tests. In both cases the models were very likely to learn how to exploit the reward, e.g., by producing empty programs or by hard-coding the public tests (and claiming to produce a "draft answer").

Code Execution Our verifier is a language-agnostic code execution sandbox which (1) extracts a program from each candidate; (2) compiles it if needed; and (3) tests it on the I/O examples $\{(in_k, out_k)\}_{k=1}^K$.

To extract the code, we instruct the model to put it in a Markdown block, which all major instruction-tuned models do by default. Since we rely on the native format of the model, we do not need to train the model with a format reward. This guarantees that the increases in rewards that we see during training are genuine improvements and not merely the result of the model learning to format correctly.

For each language, we build and cache an OCI container using the configuration L. To build the container, we install the language compiler and runtime (the script $L.\texttt{install}$), and include a generic execution harness which runs and tests candidate programs. The execution harness runs continuously in the container, waiting to receive a triple with the candidate program, the set of input/output examples, and timeouts. The harness (1) writes the program to disk (to $L.\texttt{filename}$), (2) compiles it if needed ($L.\texttt{compile}$), (3) runs it on each received input ($L.\texttt{execute}$), and verified that it produces the expected output. The harness imposes timeouts on the compilation step and each execution, and returns reward 0 on any timeout or failed verification.

It is important to have timeouts for both compilation and execution. This prevents pathologies such as unbounded macro expansion in Julia (caught by the compile timeout) and infinite loops (caught by the execution timeout). Using containers also allows us to limit CPU, memory, and filesystem usage; no elevated privileges are granted to the generated program. Although the current datasets only specify tasks by standard I/O, the same sandbox can safely accommodate problems that read/write from network or disk.

A subtle resource limit that we impose is a limit on the size of output. Even with a short timeout such as 30 seconds, a pathological candidate program can output tens of

gigabytes of text. This can crash the verifier if it naïvely tries to read and store all output. Instead, the verifier maintains a fixed-size (5MB) read buffer and immediately kills programs which overflow it.

Overall, this design lets us keep a pool of warm containers for the duration of training: we find that spawning a fresh container is two orders of magnitude slower than re-using an existing one. In our experiments, a single training run may involve testing 150,000 programs, each on several I/O examples. Most of the generated programs are faulty and some behave badly, e.g., they either timeout, consume too much memory, or produce too much output. So containers do occasionally crash or need to be killed, and our execution environment handles this automatically.

Finally, to improve compile times, our execution environment mounts a RAM disk in each container. Compilation may be slow due to creating many intermediate files, and indeed some large C++ projects, e.g., Firefox, recommend using a RAM disk to speed up their builds (Firefox 2025).

Implementation We implement the trainer and execution environment with Ray (Moritz et al. 2018), which facilitates multiprocessing and distributed computing. In particular, the framework allows distributing the training over a network of heterogeneous nodes, which allows running the trainer on a node specialized for GPU work and the execution environment on a node specialized for CPU work. Ray also lets us easily have two separate inter-communicating processes, one for group generation and another for verification and loss computation, simply by defining two actor classes. Running the two concurrently significantly speeds up training, as we found that they take a roughly comparable amount of time. The execution environment is also implemented as an actor, with one Python `asyncio` task per container. Using tasks instead of actors helps lower how much inter-process communication and data copying is needed.

Hyperparameters For our training, we use the AdamW optimizer (Loshchilov and Hutter 2019) with a learning rate of 5×10^{-6} and a cosine decay schedule with a warmup of 0.1 epochs. We process 4 prompts in each batch, with group size 32 per prompt. We generate with temperature 0.7. We train hybrid reasoning models with reasoning disabled. (We still often observe generations with reasoning-like text, either before the answer or in comments.)

4 Evaluation

To evaluate our approach and codebase, we train and benchmark multiple models on 5 low-resource programming languages. We measure pass@1 accuracy with reasoning disabled and 20 samples per prompt at temperature 0.2.

4.1 Training datasets

Ag-Codeforces-X, the main dataset we use for training, was created based on competitive programming problems from the Open-R1 Codeforces dataset (Penedo et al. 2025). Few adjustments were necessary, since the problems already specified programs and tests using standard I/O. The train split contains 5369 problems. See §B for more details.

Table 1: Pass@1 rates on AG-LiveCodeBench-X for Qwen 3 4B models trained on five low-resource languages using Ag-Codeforces-X. Each model achieves a SOTA result for an open-weight model with $\leq$16B parameters.

Model	Ag-LiveCodeBench-X				
	X= Lua	Julia	R	OCaml	Fortran
Llama-3.3-70B-Ins	**25**	22	13	7	3
Qwen3-32B	22	**26**	**17**	2	1
DSCv2-Lite-Ins-16B	13	12	9	7	6
Qwen3-4B	11	10	10	1	0
Qwen3-4B-CF-X (Ours)	23	22	15	**7**	**15**

4.2 Benchmarks

We evaluate Agnostics with the following benchmarks.

MultiPL-E (Cassano et al. 2023) is a well-established benchmark which has been frequently used to evaluate the performance of new LLMs on a broad set of languages (e.g., Kimi Team (2025); Yang et al. (2025); Grattafiori et al. (2024); ByteDance et al. (2025)). MultiPL-E was prepared by compiling HumanEval (Chen et al. 2021) prompts and unit tests from Python to each target language. Each MultiPL-E programming language requires a $\approx$ 500 LOC prompt and test translator, considerably more effort than writing an Agnostics configuration file.

A major limitation of MultiPL-E is being too easy for frontier models. With Python, frontier models are now evaluated on solving programming contest problems (Jain et al. 2024a); no multi-language benchmarks are as challenging.

Ag-LiveCodeBench-X, a contribution of this paper, is a new multi-language benchmark derived from LiveCodeBench. LiveCodeBench 5.0 has 880 problems, of which 381 have Python starter code and test cases. The remaining 499 problems do not use starter code and instead use standard I/O to specify and test solutions. Hence we used these problems to transform LiveCodeBench into an Agnostics dataset. Accordingly, benchmarking a new programming language with Ag-LiveCodeBench-X is straightforward: we can reuse the language configurations and execution environment from our trainer (§3.3).

Importantly, the problems from LiveCodeBench and thus also Ag-LiveCodeBench-X are filtered out from our training sets. Moreover, our results show that Ag-LiveCodeBench-X is significantly harder than MultiPL-E.

4.3 Results

We now present our results. We use a few abbreviations in the tables. Ag-LCB-X stands for Ag-LiveCodeBench-X; we clarify abbreviated model names in the text. One table row may present scores of *many* models; the column for language X shows the score of the model trained on X.

SOTA small LLMs for low-resource PLs Using Agnostics, we train Qwen 3 4B on Ag-Codeforces-X specialized to Fortran, Julia, Lua, OCaml, and R. The resulting Qwen3-4B-CF-X models are, to the best of our knowledge, state-of-the

Table 2: Pass@1 rates on MultiPL-E for Qwen 3 4B models trained using Ag-Codeforces-X. The significant improvement indicates that they did not overfit to the competitive programming format.

Model	MultiPL-E		
	X= Lua	Julia	R
Qwen3-4B	61	51	36
Qwen3-4B-CF-X (Ours)	**64**	**54**	**43**

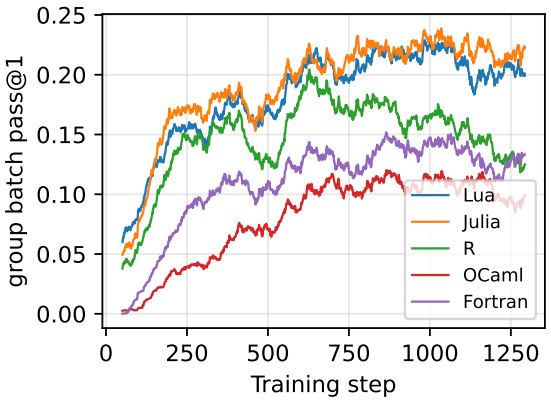

Figure 5: Training Qwen3-4B-CF-X, GRPO batch pass@1.

art low-resource programming language code models with open-weights and $\leq$16B parameters.

Benchmarking them on Ag-LiveCodeBench-X (Table 1), we see significant improvements. (i) On every language, the models match or outperform DeepSeek Coder v2 Lite Instruct (16B). (ii) On every language, our performance comes close to or even exceeds that of Qwen 3 32B and Llama 3.2 70B. (iii) Compared to the base model, Qwen 3 4B, pass@1 improves by a factor of 1.5–2x. It is safe to assume that the Qwen models, like the Llama models (Grattafiori et al. 2024), are trained on all the publicly available coding data; hence, Agnostics improves the models beyond what typical training on such data allows.

(iv) Finally, on OCaml and Fortran, we improve the performance of Qwen 3 4B from near zero to 7% and 15%. Notably, although we train the models with a prompt prefix to facilitate learning (§3.1), we do *not* use this prompt prefix for evaluation. Thus the pass@1 scores represent what the model learned, and not information provided in context.

A potential weakness of our approach is that our models are overspecialized to the competitive programming format, and can only generate programs that use standard I/O. We also evaluate with the established MultiPL-E benchmark, which does not use this format, and also find significant improvements (Table 2).[2] This indicates that our models have not overfit to the competition programming format.

Figure 5 shows the GRPO batch pass@1 rates seen when training Qwen3-4B-CF-X. All the models follow similar curves, partially due to being trained on the same data permutation. Nearly all the models slowly keep improving almost until the dataset end. We also observed the train and test split rewards to be correlated with each other (§D.2).

[2]Note that MultiPL-E does not support OCaml and Fortran.

Table 3: Pass@1 rates for larger Qwen 3 8B models trained on three languages using Ag-Codeforces-X. Agnostics scales to models of this size.

Model	MultiPL-E			Ag-LCB-X		
	X= Lua	Julia	R	X= Lua	Julia	R
Qwen3-8B	63	53	44	11	9	9
Qwen3-8B-CF-X (Ours)	**68**	**61**	**52**	**25**	**25**	**19**

Table 4: Pass@1 rates on for Qwen 3 4B models trained using MBPP. This smaller and easier dataset still results in improvements, though smaller than those in Tables 1 and 2.

Model	MultiPL-E			Ag-LCB-X		
	X= Lua	Julia	R	X= Lua	Julia	R
Qwen3-4B	51	61	36	11	10	**10**
Qwen3-4B-MBPP-X (Ours)	**51**	**62**	**41**	**15**	**15**	9

Agnostics scales to larger models To test if improvements from Agnostics training scale with model size, we train the Qwen 3 8B model on Ag-Codeforces-X specialized to Lua, Julia and R and benchmarked it on Ag-LiveCodeBench-X and MultiPL-E (Table 3). The Qwen3-8B-CF-X models improve significantly on both benchmarks, improving over their 4B counterparts.

We expect Agnostics to scale to even larger models, with appropriate computing resources. However, we found that Agnostics training with Ag-Codeforces-X does *not* improve two smaller models: Qwen 3 1.7B and Llama 3.2 3B Instruct, perhaps due to the problems being too difficult.

Agnostics works with easier problems So far, all of the models that we have trained so far have used the Open-R1 Codeforces dataset. To show that Agnostics works with other datasets, we also train models for Julia, Lua, and R using the MBPP training set. (§3.1 describes how we prepare MBPP.) Note that the MBPP problems are trivial—see Figure 3a—compared to the Open-R1 Codeforces problems. So, we cannot expect these models trained on the MBPP problems to be as good as the models we presented before.

Nevertheless, training on MBPP still improves performance on Lua and Julia (Table 4). On R, the table shows a small drop in performance on Ag-LiveCodeBench-X, but an improvement on MultiPL-E.

Agnostics works on other model families Finally, to test if Agnostics works on other models, we train both DeepSeek Coder 6.7B Instruct (Guo et al. 2024) and Phi 4 Mini Instruct (Microsoft et al. 2025) with Ag-Codeforces-X specialized to Julia, Lua, and R. Agnostics is able to improve the performance of both models on all languages, as measured by MultiPL-E and Ag-LiveCodeBench-X with one exception (Table 5). Phi 4 Mini Instruct scores zero on

Table 5: Pass@1 rates for DeepSeek Coder V1 and Phi 4 models trained using Ag-Codeforces-X. We find that Agnostics works on other model families.

Model	MultiPL-E			Ag-LCB-X		
	X=	Lua	Julia	R	X=	Lua	Julia	R
Phi4-mini-ins	40	39	34		7	7	0	
DSC-6.7B-Ins	40	54	37		8	5	7	
Phi4-mini-ins-CF-X (Ours)	41	43	35		**11**	8	0	
DSC-6.7B-Ins-CF-X (Ours)	**42**	**55**	**52**		9	**9**	**9**	

Ag-LiveCodeBench-R, and training does not improve the score. However, on the easier MultiPL-E benchmark, we see that the model is slightly better on simpler R problems ($34 \rightarrow 35$).

Note that DeepSeek Coder 6.7B is a relative old LLM, superseded by the much larger DeepSeekV2 and V3 models. Unlike Qwen 3, DeepSeek Coder is not trained with reinforcement learning, but is only an instruction-tuned model. Thus this result also shows that Agnostics can work on models that have had relatively limited post-training.

4.4 Qualitative Improvements

In this section we take a deeper look at how training with Agnostics addresses the kinds of errors that Qwen 3 4B makes on low-resource languages. First, we define a taxonomy of common bugs by prompting an LLM (o3) to classify bugs in a sample of faulty programs and then lightly editing the suggestion. §E has the full taxonomy and the prompt we used to develop it. The taxonomy spans fundamental programming errors, such as syntax errors, and more subtle mistakes such as logic flaws.

We then sample 100 problems from Ag-LiveCodeBench-X, and take five programs produced by Qwen 3 4B and the Agnostics models trained on Open-R1 Codeforces problems. Recall from Table 1 that these are significantly better than Qwen 3 4B. Using Sonnet 4, we use the taxonomy to classify the bugs in each program.

Figure 6 shows the bug distribution for OCaml. We see that the base Qwen 3 4B models makes substantially more fundamental programming mistakes in OCaml. More programs have syntax errors (55% vs 35% after training), more programs misuse builtin functions (60% vs 32%), and so on. However, we observe a small increase in logic flaws (18% vs 25% after training). Inspecting the results, the reason for this is that when a program is full of syntax errors and hallucinated functions, it is very difficult to even determine whether or not the algorithmic approach is correct. Training eliminates these shallow bugs and lets deeper issues manifest. We see the same patterns across the other four languages (§E).

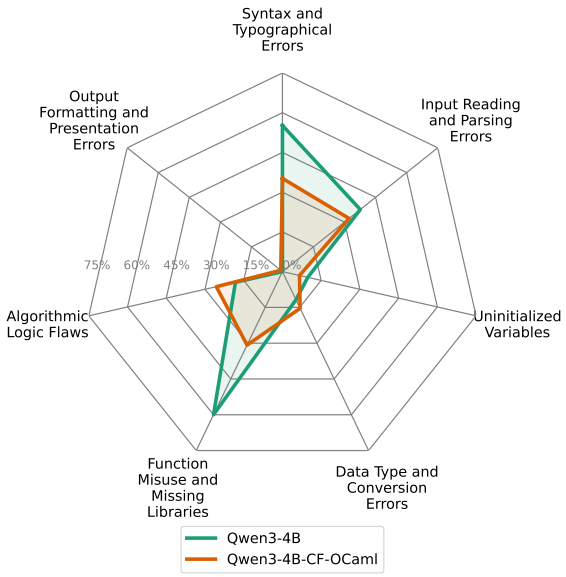

Figure 6: We classify with an LLM what bugs appear in programs produced by Qwen 3 4B and our trained model, Qwen3-4B-CF-OCaml. Each radial represents a bug category, and points along the radial indicate how many programs appear to have that bug. Qwen 3 4B makes more fundamental programming mistakes, such as syntax errors and misusing builtins, indicating its limited grasp of OCaml. Our trained model makes significantly fewer such mistakes. We see a small increase in logic flaws: the trained model makes fewer shallow mistakes, revealing deeper issues.

5 Conclusion

LLMs are strongest where pre-training and post-training data are abundant, and weakest where practitioners arguably need them most: for low-resource programming languages. This paper proposes Agnostics, a language-agnostic post-training pipeline that removes the per-language engineering tax by verifying code purely via externally observable behavior. A single verifier and a short configuration are enough to adapt the same reinforcement learning setup to new languages.

Empirically, Agnostics consistently improves small open-weight models on five low-resource languages–Lua, Julia, R, OCaml, and Fortran–without requiring language-specific test translators. Training Qwen-3 4B with Ag-Codeforces-X yields large gains on our new Ag-LiveCodeBench-X benchmark and on MultiPL-E, often rivaling or surpassing 16B–70B open-weight baselines. The method scales to larger and different model families: Qwen-3 8B benefits similarly, and we observe improvements on DeepSeek Coder 6.7B and Phi-4 Mini as well. Error-type analysis shows that the gains come from reducing fundamental language-usage mistakes.

A practical advantage of the approach is how little per-language work it requires. After the framework was in place, adding OCaml and Fortran took us less than an hour each. We expect adoption to be just as straightforward for any language with a command-line toolchain.

We see no reason to expect the approach to not work for models larger than 8B, although our experiments are limited by available compute. For scaling data, the Agnostics reformulation strategy apply to much larger problem sets as well. For instance, OpenCodeReasoning has ∼600K prob-

lems with Python solutions (Ahmad et al. 2025); converting such corpora into language-agnostic I/O tasks would provide rich RL datasets with many target languages with minimal additional engineering.

6 Acknowledgments

This work was supported by Czech Science Foundation Grant No. 23-07580X. This work is partially supported by the U.S. National Science Foundation (SES-2326174). This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award Number DESC0025613.

We used GNU parallel (Tange 2023).

References

Ahmad, W. U.; Sean Narenthiran, S. M.; Ficek, A.; Jain, S.; Huang, J.; Noroozi, V.; and Ginsburg, B. 2025. OpenCodeReasoning: Advancing Data Distillation for Competitive Coding. arXiv:2504.01943.

Allal, L. B.; Li, R.; Kocetkov, D.; Mou, C.; Akiki, C.; Ferrandis, C. M.; Muennighoff, N.; Mishra, M.; Gu, A.; Dey, M.; Umapathi, L. K.; Anderson, C. J.; Zi, Y.; Poirier, J. L.; Schoelkopf, H.; Troshin, S.; Abulkhanov, D.; Romero, M.; Lappert, M.; De Toni, F.; del Río, B. G.; Liu, Q.; Bose, S.; Bhattacharyya, U.; Zhuo, T. Y.; Yu, I.; Villegas, P.; Zocca, M.; Mangrulkar, S.; Lansky, D.; Nguyen, H.; Contractor, D.; Villa, L.; Li, J.; Bahdanau, D.; Jernite, Y.; Hughes, S.; Fried, D.; Guha, A.; de Vries, H.; and von Werra, L. 2023. SantaCoder: Don't Reach for the Stars! In *Deep Learning for Code Workshop (DL4C)*.

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. Technical Report arXiv:2108.07732, arXiv.

ByteDance; Zhang, Y.; Su, J.; Sun, Y.; Xi, C.; Xiao, X.; Zheng, S.; Zhang, A.; Liu, K.; Zan, D.; Sun, T.; Zhu, J.; Xin, S.; Huang, D.; Bai, Y.; Dong, L.; Li, C.; Chen, J.; Zhou, H.; Huang, Y.; Ning, G.; Song, X.; Chen, J.; Liu, S.; Shen, K.; Xiang, L.; and Wu, Y. 2025. Seed-Coder: Let the Code Model Curate Data for Itself. arXiv:2506.03524.

Cassano, F.; Gouwar, J.; Lucchetti, F.; Schlesinger, C.; Freeman, A.; Anderson, C. J.; Feldman, M. Q.; Greenberg, M.; Jangda, A.; and Guha, A. 2024. Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs. *Artifact: Knowledge Transfer from High-Resource to Low-Resource Programming Languages for Code LLMs*, 8(OOPSLA2): 295:677–295:708.

Cassano, F.; Gouwar, J.; Nguyen, D.; Nguyen, S.; Phipps-Costin, L.; Pinckney, D.; Yee, M.-H.; Zi, Y.; Anderson, C. J.; Feldman, M. Q.; Guha, A.; Greenberg, M.; and Jangda, A. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering (TSE)*, 49(7): 3675–3691.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. 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.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

DeepSeek-AI; Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; Zhang, X.; Yu, X.; Wu, Y.; Wu, Z. F.; Gou, Z.; Shao, Z.; Li, Z.; Gao, Z.; Liu, A.; Xue, B.; Wang, B.; Wu, B.; Feng, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; Dai, D.; Chen, D.; Ji, D.; Li, E.; Lin, F.; Dai, F.; Luo, F.; Hao, G.; Chen, G.; Li, G.; Zhang, H.; Bao, H.; Xu, H.; Wang, H.; Ding, H.; Xin, H.; Gao, H.; Qu, H.; Li, H.; Guo, J.; Li, J.; Wang, J.; Chen, J.; Yuan, J.; Qiu, J.; Li, J.; Cai, J. L.; Ni, J.; Liang, J.; Chen, J.; Dong, K.; Hu, K.; Gao, K.; Guan, K.; Huang, K.; Yu, K.; Wang, L.; Zhang, L.; Zhao, L.; Wang, L.; Zhang, L.; Xu, L.; Xia, L.; Zhang, M.; Zhang, M.; Tang, M.; Li, M.; Wang, M.; Li, M.; Tian, N.; Huang, P.; Zhang, P.; Wang, Q.; Chen, Q.; Du, Q.; Ge, R.; Zhang, R.; Pan, R.; Wang, R.; Chen, R. J.; Jin, R. L.; Chen, R.; Lu, S.; Zhou, S.; Chen, S.; Ye, S.; Wang, S.; Yu, S.; Zhou, S.; Pan, S.; Li, S. S.; Zhou, S.; Wu, S.; Ye, S.; Yun, T.; Pei, T.; Sun, T.; Wang, T.; Zeng, W.; Zhao, W.; Liu, W.; Liang, W.; Gao, W.; Yu, W.; Zhang, W.; Xiao, W. L.; An, W.; Liu, X.; Wang, X.; Chen, X.; Nie, X.; Cheng, X.; Liu, X.; Xie, X.; Liu, X.; Yang, X.; Li, X.; Su, X.; Lin, X.; Li, X. Q.; Jin, X.; Shen, X.; Chen, X.; Sun, X.; Wang, X.; Song, X.; Zhou, X.; Wang, X.; Shan, X.; Li, Y. K.; Wang, Y. Q.; Wei, Y. X.; Zhang, Y.; Xu, Y.; Li, Y.; Zhao, Y.; Sun, Y.; Wang, Y.; Yu, Y.; Zhang, Y.; Shi, Y.; Xiong, Y.; He, Y.; Piao, Y.; Wang, Y.; Tan, Y.; Ma, Y.; Liu, Y.; Guo, Y.; Ou, Y.; Wang, Y.; Gong, Y.; Zou, Y.; He, Y.; Xiong, Y.; Luo, Y.; You, Y.; Liu, Y.; Zhou, Y.; Zhu, Y. X.; Xu, Y.; Huang, Y.; Li, Y.; Zheng, Y.; Zhu, Y.; Ma, Y.; Tang, Y.; Zha, Y.; Yan, Y.; Ren, Z. Z.; Ren, Z.; Sha, Z.; Fu, Z.; Xu, Z.; Xie, Z.; Zhang, Z.; Hao, Z.; Ma, Z.; Yan, Z.; Wu, Z.; Gu, Z.; Zhu, Z.;

Liu, Z.; Li, Z.; Xie, Z.; Song, Z.; Pan, Z.; Huang, Z.; Xu, Z.; Zhang, Z.; and Zhang, Z. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. Technical Report arXiv:2501.12948, arXiv.

Du, M.; Tuan, L. A.; Liu, Y.; Qing, Y.; Huang, D.; He, X.; Liu, Q.; Ma, Z.; and Ng, S.-k. 2025. Afterburner: Reinforcement Learning Facilitates Self-Improving Code Efficiency Optimization. arXiv:2505.23387.

Firefox. 2025. Why the Build System Might be Slow. https://firefox-source-docs.mozilla.org/build/buildsystem/slow.html.

Gehring, J.; Zheng, K.; Copet, J.; Mella, V.; Cohen, T.; and Synnaeve, G. 2024. RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning. arXiv:2410.02089.

Grattafiori, A.; Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Vaughan, A.; Yang, A.; Fan, A.; Goyal, A.; Hartshorn, A.; Yang, A.; Mitra, A.; Sravankumar, A.; Korenev, A.; Hinsvark, A.; Rao, A.; Zhang, A.; Rodriguez, A.; Gregerson, A.; Spataru, A.; Roziere, B.; Biron, B.; Tang, B.; Chern, B.; Caucheteux, C.; Nayak, C.; Bi, C.; Marra, C.; McConnell, C.; Keller, C.; Touret, C.; Wu, C.; Wong, C.; Ferrer, C. C.; Nikolaidis, C.; Allonsius, D.; Song, D.; Pintz, D.; Livshits, D.; Wyatt, D.; Esiobu, D.; Choudhary, D.; Mahajan, D.; Garcia-Olano, D.; Perino, D.; Hupkes, D.; Lakomkin, E.; AlBadawy, E.; Lobanova, E.; Dinan, E.; Smith, E. M.; Radenovic, F.; Guzmán, F.; Zhang, F.; Synnaeve, G.; Lee, G.; Anderson, G. L.; Thattai, G.; Nail, G.; Mialon, G.; Pang, G.; Cucurell, G.; Nguyen, H.; Korevaar, H.; Xu, H.; Touvron, H.; Zarov, I.; Ibarra, I. A.; Kloumann, I.; Misra, I.; Evtimov, I.; Zhang, J.; Copet, J.; Lee, J.; Geffert, J.; Vranes, J.; Park, J.; Mahadeokar, J.; Shah, J.; van der Linde, J.; Billock, J.; Hong, J.; Lee, J.; Fu, J.; Chi, J.; Huang, J.; Liu, J.; Wang, J.; Yu, J.; Bitton, J.; Spisak, J.; Park, J.; Rocca, J.; Johnstun, J.; Saxe, J.; Jia, J.; Alwala, K. V.; Prasad, K.; Upasani, K.; Plawiak, K.; Li, K.; Heafield, K.; Stone, K.; El-Arini, K.; Iyer, K.; Malik, K.; Chiu, K.; Bhalla, K.; Lakhotia, K.; Rantala-Yeary, L.; van der Maaten, L.; Chen, L.; Tan, L.; Jenkins, L.; Martin, L.; Madaan, L.; Malo, L.; Blecher, L.; Landzaat, L.; de Oliveira, L.; Muzzi, M.; Pasupuleti, M.; Singh, M.; Paluri, M.; Kardas, M.; Tsimpoukelli, M.; Oldham, M.; Rita, M.; Pavlova, M.; Kambadur, M.; Lewis, M.; Si, M.; Singh, M. K.; Hassan, M.; Goyal, N.; Torabi, N.; Bashlykov, N.; Bogoychev, N.; Chatterji, N.; Zhang, N.; Duchenne, O.; Çelebi, O.; Alrassy, P.; Zhang, P.; Li, P.; Vasic, P.; Weng, P.; Bhargava, P.; Dubal, P.; Krishnan, P.; Koura, P. S.; Xu, P.; He, Q.; Dong, Q.; Srinivasan, R.; Ganapathy, R.; Calderer, R.; Cabral, R. S.; Stojnic, R.; Raileanu, R.; Maheswari, R.; Girdhar, R.; Patel, R.; Sauvestre, R.; Polidoro, R.; Sumbaly, R.; Taylor, R.; Silva, R.; Hou, R.; Wang, R.; Hosseini, S.; Chennabasappa, S.; Singh, S.; Bell, S.; Kim, S. S.; Edunov, S.; Nie, S.; Narang, S.; Raparthy, S.; Shen, S.; Wan, S.; Bhosale, S.; Zhang, S.; Vandenhende, S.; Batra, S.; Whitman, S.; Sootla, S.; Collot, S.; Gururangan, S.; Borodinsky, S.; Herman, T.; Fowler, T.; Sheasha, T.; Georgiou, T.; Scialom, T.; Speckbacher, T.; Mihaylov, T.; Xiao, T.; Karn, U.; Goswami, V.; Gupta, V.; Ramanathan, V.; Kerkez, V.; Gonguet, V.; Do, V.; Vogeti, V.; Albiero, V.; Petrovic, V.; Chu, W.; Xiong, W.; Fu, W.; Meers, W.; Martinet, X.; Wang, X.; Wang, X.; Tan, X. E.; Xia, X.; Xie, X.; Jia, X.; Wang, X.; Goldschlag, Y.; Gaur, Y.; Babaei, Y.; Wen, Y.; Song, Y.; Zhang, Y.; Li, Y.; Mao, Y.; Coudert, Z. D.; Yan, Z.; Chen, Z.; Papakipos, Z.; Singh, A.; Srivastava, A.; Jain, A.; Kelsey, A.; Shajnfeld, A.; Gangidi, A.; Victoria, A.; Goldstand, A.; Menon, A.; Sharma, A.; Boesenberg, A.; Baevski, A.; Feinstein, A.; Kallet, A.; Sangani, A.; Teo, A.; Yunus, A.; Lupu, A.; Alvarado, A.; Caples, A.; Gu, A.; Ho, A.; Poulton, A.; Ryan, A.; Ramchandani, A.; Dong, A.; Franco, A.; Goyal, A.; Saraf, A.; Chowdhury, A.; Gabriel, A.; Bharambe, A.; Eisenman, A.; Yazdan, A.; James, B.; Maurer, B.; Leonhardi, B.; Huang, B.; Loyd, B.; Paola, B. D.; Paranjape, B.; Liu, B.; Wu, B.; Ni, B.; Hancock, B.; Wasti, B.; Spence, B.; Stojkovic, B.; Gamido, B.; Montalvo, B.; Parker, C.; Burton, C.; Mejia, C.; Liu, C.; Wang, C.; Kim, C.; Zhou, C.; Hu, C.; Chu, C.-H.; Cai, C.; Tindal, C.; Feichtenhofer, C.; Gao, C.; Civin, D.; Beaty, D.; Kreymer, D.; Li, D.; Adkins, D.; Xu, D.; Testuggine, D.; David, D.; Parikh, D.; Liskovich, D.; Foss, D.; Wang, D.; Le, D.; Holland, D.; Dowling, E.; Jamil, E.; Montgomery, E.; Presani, E.; Hahn, E.; Wood, E.; Le, E.-T.; Brinkman, E.; Arcaute, E.; Dunbar, E.; Smothers, E.; Sun, F.; Kreuk, F.; Tian, F.; Kokkinos, F.; Ozgenel, F.; Caggioni, F.; Kanayet, F.; Seide, F.; Florez, G. M.; Schwarz, G.; Badeer, G.; Swee, G.; Halpern, G.; Herman, G.; Sizov, G.; Guangyi; Zhang; Lakshminarayanan, G.; Inan, H.; Shojanazeri, H.; Zou, H.; Wang, H.; Zha, H.; Habeeb, H.; Rudolph, H.; Suk, H.; Aspegren, H.; Goldman, H.; Zhan, H.; Damlaj, I.; Molybog, I.; Tufanov, I.; Leontiadis, I.; Veliche, I.-E.; Gat, I.; Weissman, J.; Geboski, J.; Kohli, J.; Lam, J.; Asher, J.; Gaya, J.-B.; Marcus, J.; Tang, J.; Chan, J.; Zhen, J.; Reizenstein, J.; Teboul, J.; Zhong, J.; Jin, J.; Yang, J.; Cummings, J.; Carvill, J.; Shepard, J.; McPhie, J.; Torres, J.; Ginsburg, J.; Wang, J.; Wu, K.; U, K. H.; Saxena, K.; Khandelwal, K.; Zand, K.; Matosich, K.; Veeraraghavan, K.; Michelena, K.; Li, K.; Jagadeesh, K.; Huang, K.; Chawla, K.; Huang, K.; Chen, L.; Garg, L.; A, L.; Silva, L.; Bell, L.; Zhang, L.; Guo, L.; Yu, L.; Moshkovich, L.; Wehrstedt, L.; Khabsa, M.; Avalani, M.; Bhatt, M.; Mankus, M.; Hasson, M.; Lennie, M.; Reso, M.; Groshev, M.; Naumov, M.; Lathi, M.; Keneally, M.; Liu, M.; Seltzer, M. L.; Valko, M.; Restrepo, M.; Patel, M.; Vyatskov, M.; Samvelyan, M.; Clark, M.; Macey, M.; Wang, M.; Hermoso, M. J.; Metanat, M.; Rastegari, M.; Bansal, M.; Santhanam, N.; Parks, N.; White, N.; Bawa, N.; Singhal, N.; Egebo, N.; Usunier, N.; Mehta, N.; Laptev, N. P.; Dong, N.; Cheng, N.; Chernoguz, O.; Hart, O.; Salpekar, O.; Kalinli, O.; Kent, P.; Parekh, P.; Saab, P.; Balaji, P.; Rittner, P.; Bontrager, P.; Roux, P.; Dollar, P.; Zvyagina, P.; Ratanchandani, P.; Yuvraj, P.; Liang, Q.; Alao, R.; Rodriguez, R.; Ayub, R.; Murthy, R.; Nayani, R.; Mitra, R.; Parthasarathy, R.; Li, R.; Hogan, R.; Battey, R.; Wang, R.; Howes, R.; Rinott, R.; Mehta, S.; Siby, S.; Bondu, S. J.; Datta, S.; Chugh, S.; Hunt, S.; Dhillon, S.; Sidorov, S.; Pan, S.; Mahajan, S.; Verma, S.; Yamamoto, S.; Ramaswamy, S.; Lindsay, S.; Lindsay, S.; Feng, S.; Lin, S.; Zha, S. C.; Patil, S.; Shankar, S.; Zhang,

S.; Zhang, S.; Wang, S.; Agarwal, S.; Sajuyigbe, S.; Chintala, S.; Max, S.; Chen, S.; Kehoe, S.; Satterfield, S.; Govindaprasad, S.; Gupta, S.; Deng, S.; Cho, S.; Virk, S.; Subramanian, S.; Choudhury, S.; Goldman, S.; Remez, T.; Glaser, T.; Best, T.; Koehler, T.; Robinson, T.; Li, T.; Zhang, T.; Matthews, T.; Chou, T.; Shaked, T.; Vontimitta, V.; Ajayi, V.; Montanez, V.; Mohan, V.; Kumar, V. S.; Mangla, V.; Ionescu, V.; Poenaru, V.; Mihailescu, V. T.; Ivanov, V.; Li, W.; Wang, W.; Jiang, W.; Bouaziz, W.; Constable, W.; Tang, X.; Wu, X.; Wang, X.; Wu, X.; Gao, X.; Kleinman, Y.; Chen, Y.; Hu, Y.; Jia, Y.; Qi, Y.; Li, Y.; Zhang, Y.; Zhang, Y.; Adi, Y.; Nam, Y.; Yu; Wang; Zhao, Y.; Hao, Y.; Qian, Y.; Li, Y.; He, Y.; Rait, Z.; DeVito, Z.; Rosnbrick, Z.; Wen, Z.; Yang, Z.; Zhao, Z.; and Ma, Z. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

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.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196.

Hu, Z.; Li, J. J.; Guha, A.; and Biswas, J. 2025. Robo-Instruct: Simulator-Augmented Instruction Alignment For Finetuning Code LLMs. arXiv:2405.20179.

Jain, A. K.; Gonzalez-Pumariega, G.; Chen, W.; Rush, A. M.; Zhao, W.; and Choudhury, S. 2025. Multi-Turn Code Generation Through Single-Step Rewards. arXiv:2502.20380.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2024a. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. In *International Conference on Learning Representationsm (ICLR)*.

Jain, N.; Han, K.; Gu, A.; Li, W.-D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; and Stoica, I. 2024b. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. Technical Report arXiv:2403.07974, arXiv.

Kimi Team. 2025. Kimi K2: Open Agentic Intelligence.

Loshchilov, I.; and Hutter, F. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; Liu, T.; Tian, M.; Kocetkov, D.; Zucker, A.; Belkada, Y.; Wang, Z.; Liu, Q.; Abulkhanov, D.; Paul, I.; Li, Z.; Li, W.-D.; Risdal, M.; Li, J.; Zhu, J.; Zhuo, T. Y.; Zheltonozhskii, E.; Dade, N. O. O.; Yu, W.; Krauß, L.; Jain, N.; Su, Y.; He, X.; Dey, M.; Abati, E.; Chai, Y.; Muennighoff, N.; Tang, X.; Oblokulov, M.; Akiki, C.; Marone, M.; Mou, C.; Mishra, M.; Gu, A.; Hui, B.; Dao, T.; Zebaze, A.; Dehaene, O.; Patry, N.; Xu, C.; McAuley, J.; Hu, H.; Scholak, T.; Paquet, S.; Robinson, J.; Anderson, C. J.; Chapados, N.; Patwary, M.; Tajbakhsh, N.; Jernite, Y.; Ferrandis, C. M.; Zhang, L.; Hughes, S.; Wolf, T.; Guha, A.; von Werra, L.; and de Vries, H. 2024a. StarCoder 2 and The Stack v2: The Next Generation. Technical Report arXiv:2402.19173, arXiv.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; Liu, T.; Tian, M.; Kocetkov, D.; Zucker, A.; Belkada, Y.; Wang, Z.; Liu, Q.; Abulkhanov, D.; Paul, I.; Li, Z.; Li, W.-D.; Risdal, M.; Li, J.; Zhu, J.; Zhuo, T. Y.; Zheltonozhskii, E.; Dade, N. O. O.; Yu, W.; Krauß, L.; Jain, N.; Su, Y.; He, X.; Dey, M.; Abati, E.; Chai, Y.; Muennighoff, N.; Tang, X.; Oblokulov, M.; Akiki, C.; Marone, M.; Mou, C.; Mishra, M.; Gu, A.; Hui, B.; Dao, T.; Zebaze, A.; Dehaene, O.; Patry, N.; Xu, C.; McAuley, J.; Hu, H.; Scholak, T.; Paquet, S.; Robinson, J.; Anderson, C. J.; Chapados, N.; Patwary, M.; Tajbakhsh, N.; Jernite, Y.; Ferrandis, C. M.; Zhang, L.; Hughes, S.; Wolf, T.; Guha, A.; von Werra, L.; and de Vries, H. 2024b. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. arXiv:2306.08568.

Ma, Y.; Liu, Y.; Yu, Y.; Zhang, Y.; Jiang, Y.; Wang, C.; and Li, S. 2023. At Which Training Stage Does Code Data Help LLMs Reasoning? In *The Twelfth International Conference on Learning Representations*.

Microsoft; :; Abouelenin, A.; Ashfaq, A.; Atkinson, A.; Awadalla, H.; Bach, N.; Bao, J.; Benhaim, A.; Cai, M.; Chaudhary, V.; Chen, C.; Chen, D.; Chen, D.; Chen, J.; Chen, W.; Chen, Y.-C.; ling Chen, Y.; Dai, Q.; Dai, X.; Fan, R.; Gao, M.; Gao, M.; Garg, A.; Goswami, A.; Hao, J.; Hendy, A.; Hu, Y.; Jin, X.; Khademi, M.; Kim, D.; Kim, Y. J.; Lee, G.; Li, J.; Li, Y.; Liang, C.; Lin, X.; Lin, Z.; Liu, M.; Liu, Y.; Lopez, G.; Luo, C.; Madan, P.; Mazalov, V.; Mitra, A.; Mousavi, A.; Nguyen, A.; Pan, J.; Perez-Becker, D.; Platin, J.; Portet, T.; Qiu, K.; Ren, B.; Ren, L.; Roy, S.; Shang, N.; Shen, Y.; Singhal, S.; Som, S.; Song, X.; Sych, T.; Vaddamanu, P.; Wang, S.; Wang, Y.; Wang, Z.; Wu, H.; Xu, H.; Xu, W.; Yang, Y.; Yang, Z.; Yu, D.; Zabir, I.; Zhang, J.; Zhang, L. L.; Zhang, Y.; and Zhou, X. 2025. Phi-4-Mini Technical Report: Compact yet Powerful Multimodal Language Models via Mixture-of-LoRAs. arXiv:2503.01743.

Moritz, P.; Nishihara, R.; Wang, S.; Tumanov, A.; Liaw, R.; Liang, E.; Elibol, M.; Yang, Z.; Paul, W.; Jordan, M. I.; and Stoica, I. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, 561–577. USA: USENIX Association. ISBN 9781931971478.

Nichols, D.; Polasam, P.; Menon, H.; Marathe, A.; Gamblin, T.; and Bhatele, A. 2024. Performance-Aligned LLMs for Generating Fast Code. arXiv:2404.18864.

Orlanski, G.; Xiao, K.; Garcia, X.; Hui, J.; Howland, J.; Malmaud, J.; Austin, J.; Singh, R.; and Catasta, M. 2023. Measuring The Impact Of Programming Language Distribution. arXiv:2302.01973.

Penedo, G.; Lozhkov, A.; Kydlíček, H.; Allal, L. B.; Beeching, E.; Lajarín, A. P.; Gallouédec, Q.; Habib, N.; Tunstall, L.; and von Werra, L. 2025. Open-R1 Codeforces. https://huggingface.co/datasets/open-r1/codeforces.

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.; Azhar, F.; Touvron, H.; Martin, L.; Usunier, N.; Scialom, T.; and Synnaeve, G. 2024. Code Llama: Open Foundation Models for Code. arXiv:2308.12950.

Roziere, B.; Zhang, J.; Charton, F.; Harman, M.; Synnaeve, G.; and Lample, G. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. In *International Conference on Learning Representations (ICLR)*.

Shao, Z.; Wang, P.; Zhu, Q.; Xu, R.; Song, J.; Bi, X.; Zhang, H.; Zhang, M.; Li, Y. K.; Wu, Y.; and Guo, D. 2024. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. Technical Report arXiv:2402.03300, arXiv.

Tange, O. 2023. GNU Parallel 20231122 ('Grindavík'). Zenodo.

Wang, Y.; Kordi, Y.; Mishra, S.; Liu, A.; Smith, N. A.; Khashabi, D.; and Hajishirzi, H. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In Rogers, A.; Boyd-Graber, J.; and Okazaki, N., eds., *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 13484–13508. Toronto, Canada: Association for Computational Linguistics.

Wei, Y.; Cassano, F.; Liu, J.; Ding, Y.; Jain, N.; Mueller, Z.; de Vries, H.; von Werra, L.; Guha, A.; and Zhang, L. 2024a. SelfCodeAlign: Self-Alignment for Code Generation. *CoRR*, abs/2410.24198.

Wei, Y.; Cassano, F.; Liu, J.; Ding, Y.; Jain, N.; Mueller, Z.; de Vries, H.; Werra, L. V.; Guha, A.; and Zhang, L. 2024b. Fully Transparent Self-Alignment for Code Generation. In *Neural Information Processing Systems (NeurIPS)*.

Wei, Y.; Duchenne, O.; Copet, J.; Carbonneaux, Q.; Zhang, L.; Fried, D.; Synnaeve, G.; Singh, R.; and Wang, S. I. 2025. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. arXiv:2502.18449.

Wei, Y.; Wang, Z.; Liu, J.; Ding, Y.; and Zhang, L. 2024c. Magicoder: Empowering Code Generation with OSS-Instruct. In *International Conference on Machine Learning (ICML)*.

Xie, Y.; Naik, A.; Fried, D.; and Rose, C. 2024. Data Augmentation for Code Translation with Comparable Corpora and Multiple References. In *Findings of EMNLP*.

Xu, F. F.; Alon, U.; Neubig, G.; and Hellendoorn, V. J. 2022. A Systematic Evaluation of Large Language Models of Code. In *Deep Learning for Code Workshop (DL4C)*.

Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; Zheng, C.; Liu, D.; Zhou, F.; Huang, F.; Hu, F.; Ge, H.; Wei, H.; Lin, H.; Tang, J.; Yang, J.; Tu, J.; Zhang, J.; Yang, J.; Yang, J.; Zhou, J.; Zhou, J.; Lin, J.; Dang, K.; Bao, K.; Yang, K.; Yu, L.; Deng, L.; Li, M.; Xue, M.; Li, M.; Zhang, P.; Wang, P.; Zhu, Q.; Men, R.; Gao, R.; Liu, S.; Luo, S.; Li, T.; Tang, T.; Yin, W.; Ren, X.; Wang, X.; Zhang, X.; Ren, X.; Fan, Y.; Su, Y.; Zhang, Y.; Zhang, Y.; Wan, Y.; Liu, Y.; Wang, Z.; Cui, Z.; Zhang, Z.; Zhou, Z.; and Qiu, Z. 2025. Qwen3 Technical Report. arXiv:2505.09388.

Yu, Q.; Zhang, Z.; Zhu, R.; Yuan, Y.; Zuo, X.; Yue, Y.; Dai, W.; Fan, T.; Liu, G.; Liu, L.; Liu, X.; Lin, H.; Lin, Z.; Ma, B.; Sheng, G.; Tong, Y.; Zhang, C.; Zhang, M.; Zhang, W.; Zhu, H.; Zhu, J.; Chen, J.; Chen, J.; Wang, C.; Yu, H.; Song, Y.; Wei, X.; Zhou, H.; Liu, J.; Ma, W.-Y.; Zhang, Y.-Q.; Yan, L.; Qiao, M.; Wu, Y.; and Wang, M. 2025. DAPO: An Open-Source LLM Reinforcement Learning System at Scale. Technical Report arXiv:2503.14476, arXiv.

Zeng, H.; Jiang, D.; Wang, H.; Nie, P.; Chen, X.; and Chen, W. 2025. ACECODER: Acing Coder RL via Automated Test-Case Synthesis. arXiv:2502.01718.

Zhou, Y.; Jiang, S.; Tian, Y.; Weston, J.; Levine, S.; Sukhbaatar, S.; and Li, X. 2025. SWEET-RL: Training Multi-Turn LLM Agents on Collaborative Reasoning Tasks. arXiv:2503.15478.

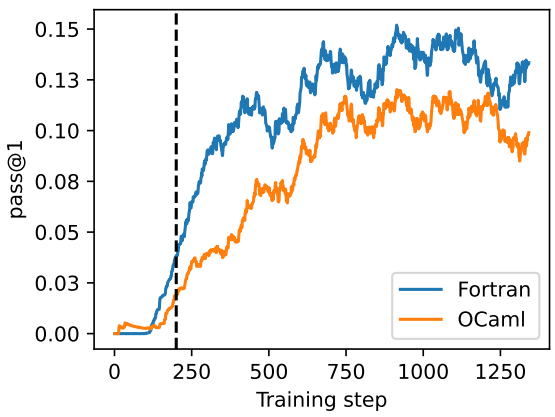

Figure 7: Rewards each step from training Qwen 3 4B on Fortran and OCaml. These are very low-resource languages, and rewards are zero for the first several steps.

A Efficiency of Rejection Sampling

An alternative to reinforcement learning is to use rejection sampling with supervised fine-tuning: prompt the model to synthesize n solutions to each task, reject solutions that fail tests, and fine-tune on the task-solution pairs that pass tests. If the model is weak at a task, which is the case with low-resource languages, then it is possible for all solutions to fail for a given task. Cassano et al. (2024) use this approach to get solutions for $\approx 30\%$ of the tasks in their dataset. They use $n = 50$ for Lua and Julia, but increase to $n = 100$ for OCaml.

The efficiency of this approach depends on the hardness of the task and the capabilities of the model. In this paper, we work with newer models that are marginally better at low-resource languages (based on MultiPL-E benchmark results). The task in Cassano et al. (2024) is to translate a simple, self-contained Python function from the model's pretraining data into an equivalent function in another programming language. This task is significantly easier than the Agnostics task, which is to solve a competitive programming problem in a low-resource language, without any reference code or tests.

We can estimate the efficiency of rejection sampling from the rewards assigned in the early steps of reinforcement learning, when the policy is closest to the initial policy. Figure 7 shows the pass1 rates while training Qwen 3 4B on Fortran and OCaml. In the first 200 steps, the average pass rate for Fortran and OCaml are 1.9% and 1.1% respectively–considerably less than the 30% success rate reported by Cassano et al. (2024) on their code translation task, making the rejection sampling approach prohibitively expensive for the low-resource language setting.

B Preparing Datasets for Agnostics

Ag-MBPP-X is a dataset of Mostly Basic Programming Problems, transformed from the MBPP (Austin et al. 2021) dataset. The source dataset contains problems asking to complete a Python function from on its signature and a docstring, where the docstring specifies what result the function should return; our dataset contains equivalent programs which read/write the same data from/to standard input/output. Out of 974 problems, we are able to translate 776 problems into Ag-MBPP-X (348 problems are in the sanitized subset of MBPP).

Processing the dataset with Qwen3-32B took less than 1 hour using 2 H100 GPUs. Figures 3a and 3b in the main body of the paper shows a sample problem from the dataset, before and after the reformulation. We used the following prompt.

```
You are a competitive programming expert.
You are given a problem that asks you to
    implement a function.
Your task is to translate the description of
     the problem into a form that accepts
    one set of function arguments as inputs
    and return the function return value as
    output.

Use programming competition style input and
    outputs -- that is, priorize the use of
    spaces and newlines to separate inputs
    and outputs over using commas and
    parentheses (or other delimiters).
    Specifically, for 2d lists, you should
    print them as a list of lists, where the
     outer lists elements are separated by
    newlines and the elements of the inner
    lists are separated by spaces.
For example, a 2d list like [[1, 2], [3, 4]]
     should be printed as:
1 2
3 4
Do not use any other delimiters.

If there are multiple 2d lists, you should
    use 2 newlines to separate them.
for example, a 2d list like [[1, 2], [3, 4]]
     and [[5, 6], [7, 8]] should be printed
    as:
1 2
3 4

5 6
7 8


If the problem requires outputing decimal
    numbers, make sure the output format
    specifies to round all decimal numbers
    to 4 decimal places. In this case, you
    should also round all the numbers in the
     output to 4 decimal places.

Do not forget to specify the input and
    output format in the description.

Here is the problem description:
{original mbpp problem description}

Here are the test cases:
{original mbpp test cases}
```

```
You should return a json object with the
    following fields:
- "description": the description of the
    problem
- "input_format": a string describing the
    input format
- "output_format": a string describing the
    output format
- "tests": a list of test cases, each test
    case is a json object with the following
     fields:
  - "input": a string that represents the
    input of the test case, in the same
    format as the input format in the
    description
  - "output": a string that represents the
    output of the test case, in the same
    format as the output format in the
    description

Place your response in a single ```json ```
    block. Do not include any other text in
    your response.
```

Ag-Codeforces-X is a dataset of competitive programming problems, created from Codeforces problems in the `open-r1/codeforces` dataset. The source problems already specified programs by their I/O behavior, hence only very minor changes were needed to build language-universal Agnostics problems out of the fields in the dataset: we only skipped the time and memory restrictions present in the original problems. To be precise, we used data from the `open-r1/codeforces-cots` dataset, `solutions_py_decontaminated` subset, which contains problems decontaminated using 8-gram overlap against multiple benchmarks, in particular LiveCodeBench. We used the auxilliary `checker_interactor` subset to only keep the problems which admit a simple verifier for their solutions, i.e., problems where a single output is correct for each input and where the solution only needs to read data from its input, compute the result, and write it to the standard output. We prepared both a train and a test split. The former contains 5369 problems and the latter contains 107 problems we held out from the source dataset, 7 selected manually and 100 randomly. The train split features randomized prompts, which we found help with generalizing the results of training on the dataset to other benchmarks. The prompts were randomly split into a number of types. 30% of the prompts use standard Markdown headings to start different sections of the prompt, 35% use bold text instead, and the remaining 35% simply concatenate the prompt sections together. Half of the prompts of the final type also do not include an I/O sample in the prompt.

Ag-LiveCodeBench-X is also a dataset of competitive programming problems, created from a subset of the LiveCodeBench dataset (Jain et al. 2024b). LiveCodeBench 5.0 has 880 problems, of which 381 have Python starter code and test cases. The remaining 499 problems do not use starter code and instead use standard I/O to specify and test solutions. Hence we used these problems to transform LiveCodeBench into an Agnostics dataset. Ag-LiveCodeBench-X only has a train split, like its source dataset.

C Agnostics Configurations

In this section, we list the configurations that we use for our target languages. The configuration files use YAML. The prompts for OCaml and Fortran have instructions generated by OpenAI o3.

The Lua configuration:

```
prompt: Use Lua 5.1, targeting LuaJIT.
install: apt-get install -y luajit
filename: snippet.lua
execute: luajit snippet.lua
```

The Julia configuration:

```
prompt: Use Julia 1.11.
container:
  base-image: "julia:1.11.3"
  type: debian
filename: snippet.jl
execute: julia snippet.jl
```

The R configuration (the same as Figure 4, repeated here for convenience):

```
install: apt-get install -y r-cran-tidyverse
filename: snippet.R
execute: Rscript snippet.R
prompt: |
  Use R version 4.
  Use `readLines(con = file("stdin"))` to
  read input from stdin. Optionally, use
  the `n` argument to read the first `n`
  lines. For example:
  ```r
  input <- readLines(con = file("stdin"), n
    = 1)
  n <- as.integer(input)
  print(n) # print the first line of input
  ```
  Also, use `cat` to print output to stdout.
  For example:
  ```r
  cat(n)
  ```
  Please do not use `print` to print output.
```

The OCaml configuration:

```
prompt: |
  Use OCaml 5.

  Numbers:   + - * / mod   vs.   +. -. *. /.
     **    (add dots!)
  Casts:     float_of_int   int_of_float
    int_of_string
  Mutation:  refs (:= !) or pass new values
    recursively
  Strings:   split_on_char, String.get =>
    char, use Printf "%c"
  Lists:     avoid List.nth; prefer pattern-
    match / folds / arrays
```

```
container:
  base-image: "docker.io/ocaml/opam:ubuntu
    -22.04-ocaml-5.0"
  type: debian
install:
  container-instructions: |
    RUN opam install base stdio utop
    ENV OPAM_SWITCH_PREFIX='/home/opam/.opam
    /5.0'
    ENV CAML_LD_LIBRARY_PATH='/home/opam/.
    opam/5.0/lib/stublibs:/home/opam/.opam
    /5.0/lib/ocaml/stublibs:/home/opam/.opam
    /5.0/lib/ocaml'
    ENV OCAML_TOPLEVEL_PATH='/home/opam/.
    opam/5.0/lib/toplevel'
    ENV MANPATH=':/home/opam/.opam/5.0/man'
    ENV PATH='/home/opam/.opam/5.0/bin:/usr/
    local/sbin:/usr/local/bin:/usr/sbin:/usr
    /bin:/sbin:/bin'
filename: snippet.ml
execute: utop -require base -require stdio
    snippet.ml
```

The Fortran configuration:

```
prompt: |
  Use Fortran 90. Some tips:

  Always begin each scope with implicit none
    , pick explicit kinds via
  selected_*_kind, and declare proper
    lengths-character(len=*) is legal only
    for
  dummy arguments, not locals.  Strings are
    blank-padded: call len_trim before
  iterating, and store dynamic text in
    deferred-length allocatables
  (character(len=:), allocatable :: s).
    List-directed read(*,*) arr does not
  auto-size arrays; read a count first, then
     allocate and read, or tokenize a
  line manually.  When translating 0-based
    formulas (heaps, bit positions)
  remember Fortran arrays default to 1-based
    ; if you want 0-based, declare lower
  bounds.  Use real literals (2.0d0, 1.0_rk)
     to avoid silent integer division,
  and guard against overflow when
    exponentiating integers.  For frequency
    tables,
  allocate an array or use findloc; Fortran
    lacks native dicts/sets, so you must
  implement search yourself.  Prefer array
    intrinsics (sum, count, pack) over
  hand-rolled loops, and keep helper
    procedures inside a contains section or
  module so interfaces are explicit.  return
     inside the main program is
  non-idiomatic; use structured blocks or
    stop.  Never print interactive prompts
  in batch solutions; just read, compute,
    and write.
install: apt-get install -y gfortran
filename: snippet.f90
```

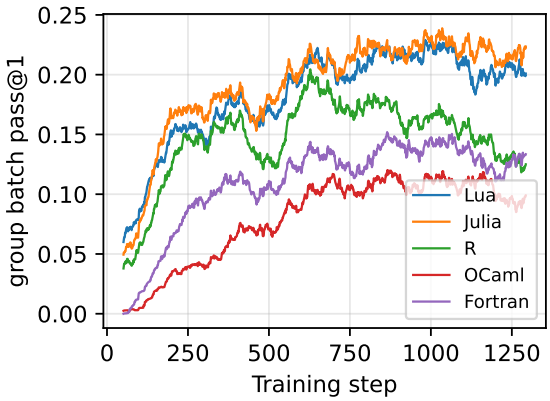

Figure 8: Training Qwen3-4B-CF-X, GRPO group batch pass@1.

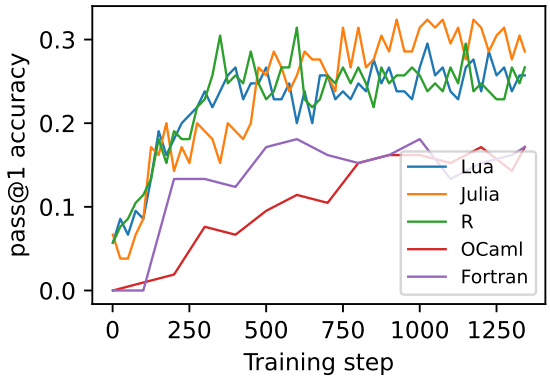

Figure 9: Training Qwen3-4B-CF-X, test split pass@1.

```
compile: gfortran -o snippet.out snippet.f90
execute: ./snippet.out
```

D Training and Results

D.1 Choosing hyperparameters

Before picking the hyperparameters described in §3.3, we investigated other values by training the Qwen 3 4B model on a previous version of Ag-Codeforces-X. We trained two models for each of Lua, Julia and R, using a linear learning rate schedule with the same learning rate. We decided against it since some of the runs degraded the model's capabilities, unlike any of the runs we did with a cosine decay schedule. We also compared between GRPO group sizes of 16, 32 and 64 by training the same model on Lua. We observed that in some runs with group size 16, the model improved significantly less than at higher group sizes. Since we found no significant difference between group sizes 32 and 64, we chose the smaller one due to limited resources.

D.2 Training dynamics

In this section we discuss the measurements we took while training the models. Figure 8 shows the GRPO group batch

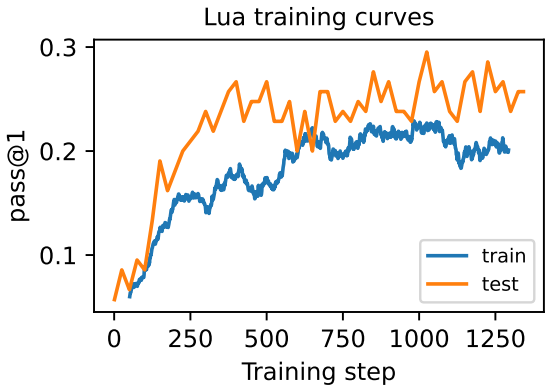

Figure 10: Training Qwen3-4B-CF-Lua, GRPO group batch and test split pass@1.

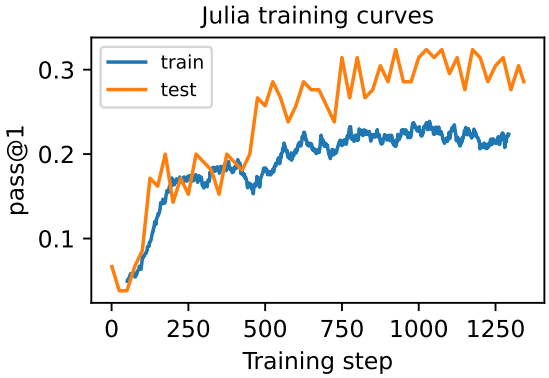

Figure 11: Training Qwen3-4B-CF-Julia, GRPO group batch and test split pass@1.

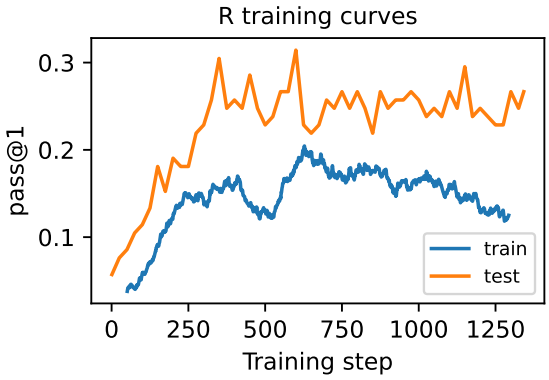

Figure 12: Training Qwen3-4B-CF-R, GRPO group batch and test split pass@1.

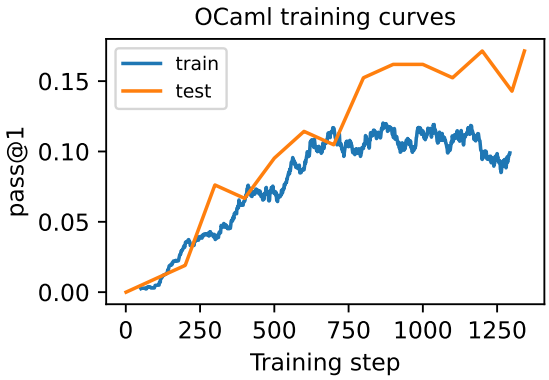

Figure 13: Training Qwen3-4B-CF-OCaml, GRPO group batch and test split pass@1.

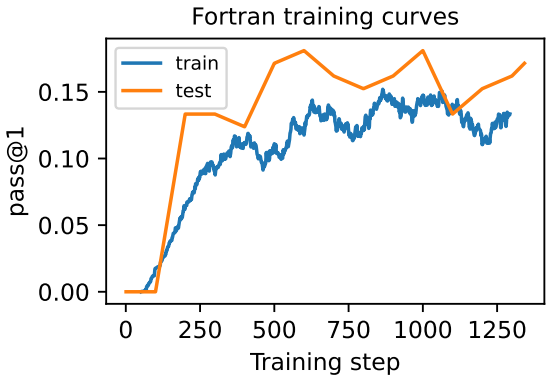

Figure 14: Training Qwen3-4B-CF-Fortran, GRPO group batch and test split pass@1.

pass@1 while training the Qwen3-4B-CF-X models. The scores of all the models are broadly correlated with one another, in part due to training them on the same permutation of the training data. Figures 10 to 14 compare the GRPO group pass@1 scores with the pass@1 scores on the test split. We see that the scores on the test split is broadly correlated with the train split rewards. In most cases, we see that the train scores keep increasing until the end of the epoch, together with the test split pass@1 scores, indicating that the model keeps improving until the end of the dataset.

D.3 Hardware and Software Used

We used three machines while working on this paper: B, R1 and R2. R2 was only used to generate completions of trained models for evaluation, while B and R1 were used to train models. Upon publication of the paper, we will publicly release Wandb records of our training runs, which include the duration and the machine used.

B has 2 Intel Xeon Gold 6342 CPUs @ 2.80GHz, 1008 GB of RAM, 4 NVIDIA H100 80GB, and uses Ubuntu 22.04.5 LTS.

R1 has 2 AMD EPYC 9454 48-Core CPUs, 8 NVIDIA H100 80GB (with NVLink connections), 2268 GB of RAM, and uses Ubuntu 22.04.5 LTS.

R2 has 2 Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz, 10 NVIDIA RTX A600, 504 GB of RAM, and uses Ubuntu 22.04.5 LTS.

When developing the Agnostics framework, we used the following major Python libraries: `ray v2.46.0`, `torch v2.6.0`, `transformers v4.54.1`, `vllm v0.8.5.post1`, `datasets v3.4.1`, `wandb 0.19.11`.

E Bug Taxonomy

E.1 Prompt for Generating Taxonomy

We used the following instructions to generate the bug taxonomy, followed by a list of faulty R programs.

> **Input:** The attached file contains multiple failed R programs (Version 4) with their:
>
> - Source code
> - Expected output
> - Actual standard output
> - Error messages (where applicable)
>
> **Objective:** Analyze these program failures systematically to create a comprehensive taxonomy of 10-12 bug themes that categorize the underlying causes of failure.
>
> **Instructions:**
>
> 1. **Initial Analysis**
>   - Read through ALL program examples carefully
>   - For each failure, identify the root cause (not just the symptom)
>   - Note any patterns or commonalities across failures
> 2. **Taxonomy Development**
>   - Create 10-12 distinct bug themes that collectively cover all observed failures
>   - Each theme should represent a fundamental type of programming error or misconception
>   - Themes should be mutually exclusive when possible, but comprehensive in coverage
>   - Order themes from most to least frequent (or by logical grouping)
> 3. **For Each Bug Theme, Provide:**
>   - Theme Name: A concise, descriptive title
>   - Description: 2-3 sentences explaining the nature of this bug type
>   - Common Symptoms: How these bugs typically manifest (error messages, incorrect output, etc.)
>   - Root Causes: The underlying programming mistakes or misconceptions
>   - Examples: Reference 2-3 specific programs from the file that exhibit this theme
>   - Prevention Tips: Brief advice on how to avoid this type of bug
> 4. **Constraints:**
>   - Focus on R-specific issues as well as general programming errors
>   - Base your taxonomy ONLY on the provided examples
>   - You may search online ONLY to understand specific R error messages or function behavior, not for existing bug taxonomies
>   - Ensure every failed program in the file can be classified under at least one theme
> 5. **Deliverable Format:** Present your taxonomy as a numbered list with clear formatting and comprehensive coverage of all observed failure patterns. Supply a short explanation for each theme in your taxonomy.

The prompt produced a taxonomy of 11 bug categories. We edited these categories and selected 7 categories relevant to us, shown in §E.2.

E.2 Bug Taxonomy Used For Analysis

The following categories represent the prevalent themes of programming errors we use in our analysis of bugs in model-generated code. They cover the full spectrum of parse, runtime, and logical failures typically encountered in programming. The themes are not mutually exclusive; we allow a program to have more than one themes.

1. **Syntax and Typographical Errors**: Missing commas, mismatched parentheses, or other typos that cause compile-time parse errors.
2. **Input Reading and Parsing Errors**: Mis-reading or mis-parsing input, leading to empty or malformed variables and subsequent failures.
3. **Uninitialized Variables**: References to variables never defined, causing undefined behavior or runtime faults.

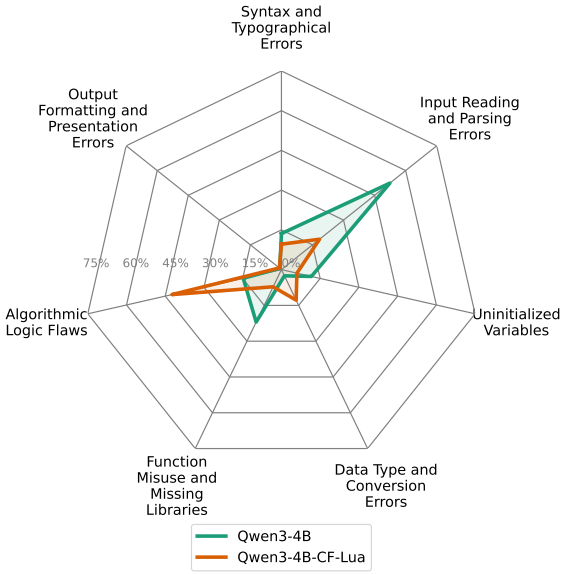

Figure 15: Radar chart of Lua error themes for Qwen3-4B and Qwen3-4B-CF-Lua.

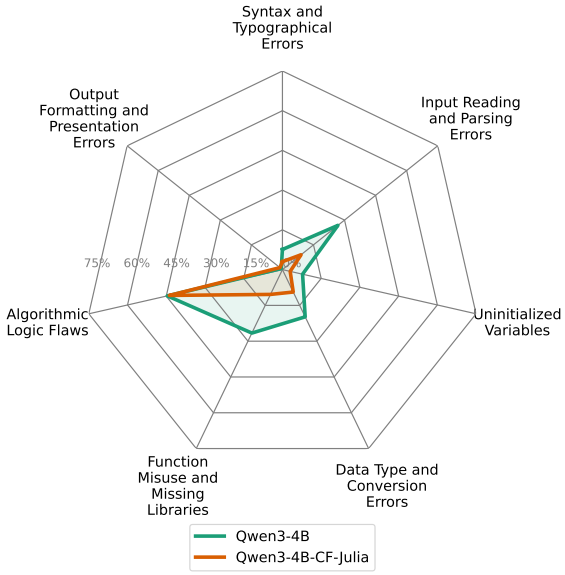

Figure 16: Radar chart of Julia error themes for Qwen3-4B and Qwen3-4B-CF-Julia.

4. **Data Type and Conversion Errors**: Incorrect casting or type misuse that triggers type errors, warnings, or incorrect results.
5. **Function Misuse and Missing Libraries**: Invocations of non-existent or mis-parameterized functions, or missing imports/libraries, causing errors.
6. **Algorithmic Logic Flaws**: Programs that compile and run but produce wrong answers due to faulty logic or conditions.
7. **Output Formatting and Presentation Errors**: Correct computational results, but incorrect due to formatting issues (e.g. missing newlines/spaces or output spec violations).

E.3 Radar Charts for All Programming Languages

Figures 15, 16, 17, 18, 19 show the error theme charts for all the programming languages we trained a model on. Figure 18 is the same as Figure 6 from the main body; we repeated it here for convenience.

F Reproducibility Checklist

This paper:

- Includes a conceptual outline and/or pseudocode description of AI methods introduced: yes
- Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results: yes
- Provides well marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper: yes

Does this paper make theoretical contributions? No.
Does this paper rely on one or more datasets? Yes.

- A motivation is given for why the experiments are conducted on the selected datasets: yes.
- All novel datasets introduced in this paper are included in a data appendix: no.
- All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes: yes.
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations: yes.
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available: yes.
- All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisfying: N/A.

Does this paper include computational experiments? Yes.

- This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting: yes.
- Any code required for pre-processing data is included in the appendix: no.
- All source code required for conducting and analyzing the experiments is included in a code appendix: no.
- All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes: yes.

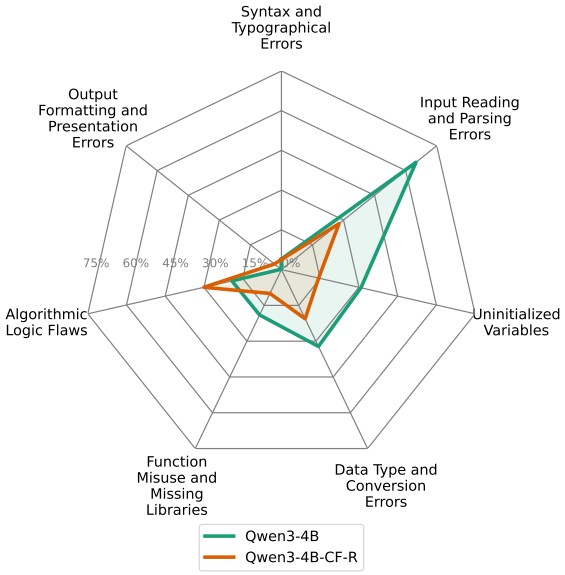

Figure 17: Radar chart of R error themes for Qwen3-4B and Qwen3-4B-CF-R.

- All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from: no.
- If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results: yes.
- This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks: yes.
- This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics: yes.
- This paper states the number of algorithm runs used to compute each reported result: yes.
- Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information: no.
- The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank): no.
- This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments: yes.

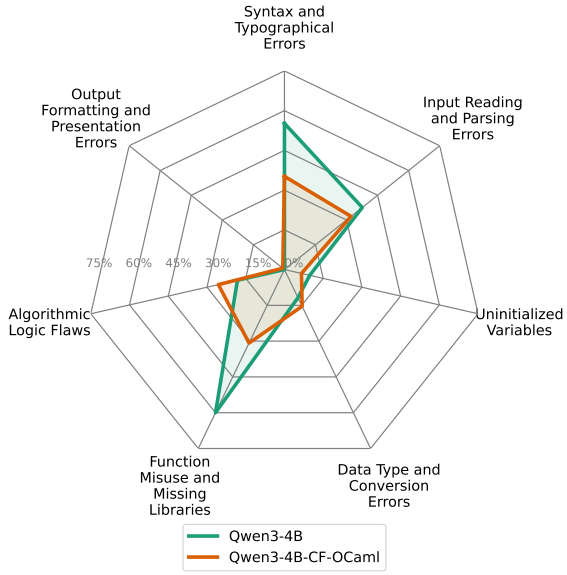

Figure 18: Radar chart of OCaml error themes for Qwen3-4B and Qwen3-4B-CF-OCaml.

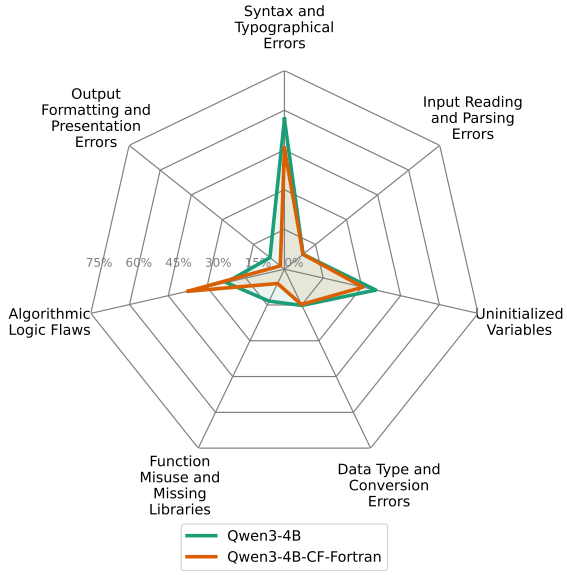

Figure 19: Radar chart of Fortran error themes for Qwen3-4B and Qwen3-4B-CF-Fortran.