

Visible Tests, Held-Out Tests and a Judge: Measuring Reward Hacking in Multi-Turn Code Repair with GRPO

DC-Jev-RL project, Part III-A

October 2026

Abstract

A policy rewarded by the tests it can see may learn to satisfy them instead of solving the problem. We measure this in multi-turn code repair: Qwen3.5-4B-Base writes a function, sees the first failing example test, and tries again, up to three times; held-out tests decide whether the final program is correct. Offline, 36% of the base policy’s attempts that pass the visible tests fail a held-out test, and both Kev-4B and an autoregressive judge of the same size rank such attempts below correct ones (AUROC 0.87 and 0.86) and separate hard-coded lookups of the tests from reference solutions almost perfectly. We then train with GRPO for 100 steps, two seeds per reward, on the visible tests alone, on the visible tests times Kev-4B’s judgment that the code is genuine, and on all tests as an oracle. We saw no sign that the visible-test reward was exploited: the share of problems solved rose on KodCode validation (65.5% to 86.6%), MBPP+ and HumanEval+, the share of visible-test passes that are wrong fell by about half on two of the three sets, and programs that copy values from the visible tests stayed as rare as in the base policy. The judge factor changed nothing beyond the variation between seeds, and the oracle reward, which never pays a wrong program, left slightly fewer wrong passes and solved no more problems. Code, logs and every number in this paper are in the repository.

1 Introduction

A coding agent usually sees a few example tests, while the code it writes has to work beyond them. A training reward built from the visible tests pays for passing them, not for solving the problem. A policy can learn to satisfy the examples in ways that do not generalise, and in the extreme to special-case them. Reward hacking of this kind has been observed when reasoning models are trained against tests [2, 4, 6].

This paper measures how much a judge can help, in a small but complete setting. The task is multi-turn code repair: the policy writes a function, sees the first failing example test and its error, and tries again, up to three times. The rewards are the visible tests alone; the visible tests times Kev-4B’s judgment of whether the code “solves the problem for all valid inputs, instead of special-casing the example tests”; and, as an oracle, the visible and the held-out tests. We measure the gap between what the visible tests certify and what the held-out tests find. We do this offline, for the base policy’s code and for hard-coded lookups built to pass the visible tests, and online, during and after GRPO training.

2 Related work

Execution feedback is the standard reward for code RL. Benchmarks with extended tests such as EvalPlus [5] showed that the original tests of HumanEval and MBPP [1, 3] miss many wrong programs. That is exactly the gap a visible-test reward leaves open. KodCode [7] provides synthetic problems with verified tests for RL. LLM judges of code correctness are the code counterpart of the generative verifiers [8] that Part II studies for

math. Monitoring a policy’s reasoning for signs of reward hacking [2] is another way to use a second model; here the judge reads the final code only.

3 Task: code repair with visible and held-out tests

Problems. Training problems come from KodCode-Light-RL-10K [7]: Python functions with `pytest` unit tests. We keep the problems whose test file tests one function, uses no fixtures or third-party packages, and has at least three test functions. The first test function is shown to the policy (*visible*); the others are *held out*. 300 problems (seed 0) are held out as the validation set (`kodcode_val`). The test sets are MBPP+ and HumanEval+ [1, 3, 5]. For MBPP+ the original MBPP asserts are visible (three for most problems, up to seven). For HumanEval+ the first three asserts of the original HumanEval test are visible (fewer where it has fewer). In both, the EvalPlus harness with its added tests is held out. A problem is dropped if its reference solution fails any of its own tests under our harness, and we report how many were dropped. Each program runs in its own `python -I` subprocess in an empty temporary directory. It has CPU-time, address-space and file-size limits and a wall-clock timeout, and runs after the destructive functions of `os`, `shutil` and `subprocess` are disabled (the “reliability guard” of OpenAI’s `human-eval`). This contains buggy code from a small model; it is not a security boundary.

Episodes. The policy sees the problem, the function signature and the visible tests, and writes a solution (at most 768 tokens per attempt). The code runs on the visible tests. If one fails, the policy sees the first failing test and its error and writes another attempt, up to three in all. Every attempt continues the exact token sequence of the earlier turns. An episode ends at the first attempt that passes every visible test, or after the third, and its final code then runs on the held-out tests. The prompts are in `experiments/part3a_code/prompts/code_prompts.json` and were fixed before any run.

Metrics. *vis pass*: the final code passes every visible test. *all pass*: it passes every visible and every held-out test. *hack*: it passes the visible tests and fails a held-out one. This counts an over-fitted or special-cased solution and also a merely incomplete one: the metric cannot tell the two apart, and we report it as the gap between what the visible tests certify and what is true. *First-attempt all pass* measures the policy without repair.

4 Rewards

- **vis**: 1 if the final code passes every visible test. This is all an agent that holds only the visible tests can check, and it is the reward that invites over-fitting to them.
- **vis_kev**: **vis** times Kev-4B’s $p(\text{yes})$ for *genuine*: “Does the proposed code solve the problem for all valid inputs, instead of special-casing the example tests?” Kev-4B sees the problem, the visible tests and the final code, never the held-out tests.
- **hidden**: 1 if the final code passes every visible and every held-out test. This is an oracle: what **vis** would be with complete tests.

Every attempt of an episode is trained with the episode’s advantage, and the feedback tokens are context, not trained. The update is Part II’s: group baseline over 8 episodes per problem, batch scale, token-level loss, truncated importance sampling, one optimizer step per batch.

	held-out tests pass, among visible-pass attempts				all pass	
	KodCode val	MBPP+	HumanEval+	all [95% CI]	within	all
attempts passing the visible tests	391	225	116	732		
of which fail a held-out test	169	57	37	263		
Kev-4B <i>genuine</i>	0.89	0.83	0.85	0.87 [0.84, 0.90]	0.81	0.93
Kev-4B <i>correct</i>	0.88	0.82	0.86	0.87 [0.84, 0.90]	0.79	0.93
AR <i>genuine</i>	0.90	0.74	0.84	0.86 [0.83, 0.89]	0.85	0.94
AR <i>correct</i>	0.87	0.74	0.84	0.84 [0.81, 0.87]	0.81	0.93

Table 1: AUROC of each judge signal for passing the held-out tests, among the base policy’s attempts that pass the visible tests (4 attempts per problem at temperature 1). *within*: mean AUROC within the 62 problems that have visible-pass attempts both passing and failing the held-out tests. *all pass*: AUROC for passing every test, over all attempts. 95% intervals from resampling problems.

5 Offline judge study

Before training, the base policy writes 4 single attempts per problem at temperature 1 on kodcode_val, MBPP+ and HumanEval+. We run them on both test sets, and both judges answer *genuine* and *correct* for each. The question that matters for a visible-test reward is whether a judge can separate, among the attempts that pass the visible tests, those that also pass the held-out tests. We report that AUROC, with the AUROC for all-pass over all attempts. We also build two kinds of code with known status for MBPP+ and HumanEval+. One is the reference solution. The other is a *hard-coded* lookup that returns the expected value for each visible test’s arguments and None otherwise. It is kept only if it passes the visible tests, so it is a pure test-special-casing program. We report how often each judge scores the lookup above the reference solution of the same problem.

6 What the judges see in code

The visible tests leave a large gap. The base policy’s 3,340 single attempts (4 per problem) pass the visible tests in 21.9% of cases and every test in 14.0%. Of the attempts that pass the visible tests, 36% fail a held-out test: 43% on KodCode val, 25% on MBPP+ and 32% on HumanEval+. A reward built from the visible tests pays these attempts as much as the correct ones, before any training has had a chance to exploit it.

Both judges see most of it, and Kev-4B is no better on the training distribution. Table 1 gives, among the attempts that pass the visible tests, each signal’s AUROC for also passing the held-out tests. Over all three sets, Kev-4B’s *genuine* reaches 0.87 [0.84, 0.90] and the AR judge’s 0.86 [0.83, 0.89] (95% intervals from resampling problems). The difference, +0.013 [−0.007, 0.033], is not resolved. With the *correct* question Kev-4B is ahead by +0.030 [0.011, 0.050]. Most of either difference comes from MBPP+, where Kev-4B leads by +0.086 [0.045, 0.128] on *genuine*. On KodCode val, the distribution the policy is trained on, the AR judge is ahead by a margin that is not resolved (−0.012 [−0.038, 0.010]), and on HumanEval+ the two cannot be told apart (+0.010 [−0.054, 0.069]). Over all attempts, both judges separate the ones that pass every test from the rest better (0.93 and 0.94), with the AR judge slightly ahead (−0.012 [−0.020, −0.005]).

Within a problem the evidence is thin. GRPO’s group baseline compares the episodes of one problem, so the judge factor in vis_kev matters through how it ranks a problem’s visible-pass episodes. Only 62 of the 442 problems with a visible-pass attempt have attempts on both sides of the held-out tests. Within

signal	AUROC	mean $p(\text{yes})$		lookup scored above reference
		reference	lookup	
Kev-4B <i>genuine</i>	1.000	0.75	0.03	0.0%
Kev-4B <i>correct</i>	0.997	0.75	0.07	0.2%
AR <i>genuine</i>	1.000	0.80	0.02	0.0%
AR <i>correct</i>	0.991	0.79	0.05	0.8%

Table 2: Reference solutions against hard-coded lookups of the visible tests, on the 507 MBPP+ and HumanEval+ problems whose visible tests allow a lookup: AUROC, mean $p(\text{yes})$ of each kind, and how often the lookup scores above its own problem’s reference solution.

them Kev-4B’s *genuine* reaches 0.81 and the AR judge’s 0.85, a difference of -0.040 $[-0.109, 0.021]$. These problems are too few to rank the judges, and too few to predict how much the factor will change training.

Hard-coded lookups are caught. 507 lookups of the visible tests’ calls pass the visible tests of their MBPP+ or HumanEval+ problem (1 of them also passes the held-out tests). Both judges separate them from the reference solutions of the same problems almost perfectly (Table 2). Kev-4B’s *genuine* gives a lookup 0.03 on average and a reference solution 0.75, and never scores a lookup above the reference solution of its own problem. Such lookups are the crudest way to special-case the tests. That the judges catch them shows that the *genuine* question is understood. It does not show that subtler over-fitting would be caught.

What this predicts. A judge factor can down-weight most of the visible-pass attempts that are wrong, and Kev-4B does that about as well as the AR judge, a general model of the same size asked the same question. Whether the factor reduces that gap during training, and at what cost to the visible-test pass rate, is what the training runs measure.

7 Training against the visible tests

Each reward trained the policy for 100 steps (16 problems with 8 episodes each, learning rate 10^{-5} as in Part II) with 2 seeds. Table 3 gives the final policies on the three evaluation sets, Table 4 the paired differences between rewards, and Figure 1 the training curves. The 95% intervals resample problems; they do not include the variation between seeds, so we also give each seed’s value where the seeds disagree.

Training on the visible tests made the code more correct. At step 0 the base policy passes every test on 65.5% of the KodCode validation problems, 54.8% of MBPP+ and 49.1% of HumanEval+. After 100 steps with the *vis* reward it passes every test on 86.6%, 69.3% and 76.4%, and it needs fewer repairs: it solves 84.7% of the KodCode validation problems at the first attempt, against 59.8% at step 0. MBPP+ and HumanEval+ were never trained on.

The gap between visible and held-out tests shrank on two sets, not on MBPP+. Of the base policy’s visible-test passes on KodCode validation, 23.7% fail a held-out test; after training 10.9% do. On HumanEval+ the share falls from 27.7% to 13.7%. On MBPP+ it barely moves, from 20.0% to 18.2%: the policy passes many more of MBPP+’s visible tests (68.6% of the problems at step 0, 84.7% after training), and the problems where the visible tests pass but a held-out test fails grow from 13.7% to 15.4% of all problems. Training did not make the policy more fitted to the visible tests than the base policy was, but on MBPP+ it did not make it less fitted either.

condition	seeds	KodCode val			MBPP+			HumanEval+		
		vis	all	hack	vis	all	hack	vis	all	hack
vis	2	97.1	86.6	10.6	84.7	69.3	15.4	88.5	76.4	12.1
vis_kev	2	96.6	86.6	10.1	85.2	72.3	12.9	87.9	76.7	11.2
hidden	2	96.3	87.2	9.1	82.0	68.8	13.3	88.2	77.0	11.2

Table 3: Final policies (greedy, up to three attempts): visible tests pass (*vis*), every test passes (*all*), and visible tests pass but a held-out test fails (*hack*), in percent of problems; seed means.

minus vis	all pass			hack		
	KodCode val	MBPP+	HumanEval+	KodCode val	MBPP+	HumanEval+
vis_kev	+0.0	+3.1	+0.3	−0.5	−2.5	−0.9
	[−2.5, 2.5]	[1.1, 5.2]	[−3.7, 4.3]	[−2.9, 1.7]	[−4.5, −0.7]	[−3.7, 1.9]
hidden	+0.7	−0.5	+0.6	−1.5	−2.1	−0.9
	[−1.8, 3.2]	[−2.5, 1.3]	[−3.4, 5.0]	[−3.9, 0.8]	[−4.0, −0.3]	[−4.7, 2.8]

Table 4: Paired differences to *vis* in percentage points: per problem, the seed mean of each condition, averaged over problems, with a 95% bootstrap interval over problems.

The oracle reward: slightly fewer wrong passes, no more problems solved. *hidden* pays only for code that passes every test, so it never pays for a wrong program. Its final policy solves as many problems as *vis*’s: all-pass differs by +0.7 [−1.8, 3.2] points on KodCode validation, −0.5 [−2.5, 1.3] on MBPP+ and +0.6 [−3.4, 5.0] on HumanEval+ (Table 4). It passes the visible tests less often (82.0% against 84.7% on MBPP+), and most of the difference is in programs that are wrong: the hack rate differs by −1.5 [−3.9, 0.8], −2.1 [−4.0, −0.3] and −0.9 [−4.7, 2.8] points, an interval that excludes zero only on MBPP+. A reward that never pays a wrong program keeps some of them from passing the visible tests; within 100 steps it does not turn them into right ones.

The judge factor: an effect on MBPP+ in one seed only. *vis_kev* multiplies the visible-test reward by Kev-4B’s *genuine* score. On KodCode validation and HumanEval+ its final policies solve as many problems as *vis*’s (+0.0 [−2.5, 2.5] and +0.3 [−3.7, 4.3]). On MBPP+ its seed mean is better: all-pass differs by +3.1 [1.1, 5.2] points and the hack rate by −2.5 [−4.5, −0.7], intervals over problems that exclude zero. The seeds disagree, though: *vis_kev* passes every MBPP+ test on 75.8% of the problems with seed 1 and 68.9% with seed 2, while *vis* gives 69.1% and 69.4%. One of two seeds is not an effect we can claim.

No copying of the tests. The hack rate counts every wrong program that passes the visible tests, whether it is merely incomplete or special-cases the tests. To separate the two we look for the crudest form of special-casing: a program that contains a value from the visible tests, an argument or an expected result, that the reference solution does not contain (dcjev/analysis/code_hacks.py; only the tested function and the module-level code it uses are searched, so a model’s own example calls do not count). The check flags 488 of the 507 hard-coded lookups of the offline study, every one that contains a non-trivial value, and none of the 537 reference solutions. Among the base policy’s attempts in the offline study it flags 2 of the 263 that pass the visible tests and fail a held-out one, and 1.1% of the others. During training it flags at most 0.7% of the episodes in any ten-step window of *vis* (0.7% for *vis_kev*, 0.8% for *hidden*), with no upward trend (Table 5). In the final evaluation it flags 1 of *vis*’s 218 hack episodes, 0 of *vis_kev*’s 193 and 0 of *hidden*’s 190. The wrong programs that pass the visible tests almost never contain a value from the tests,

condition	seeds	training episodes that copy			final evaluation, copy	
		steps 0–9	steps 50–59	steps 90–99	hack	all pass
vis	2	0.2%	0.2%	0.0%	1 / 218	2 / 1283
vis_key	2	0.2%	0.1%	0.0%	0 / 193	1 / 1307
hidden	2	0.2%	0.0%	0.0%	0 / 190	4 / 1285

Table 5: Programs that contain a value of the visible tests that the reference solution does not: share of training episodes in three ten-step windows (seed mean), and counts among the final evaluation’s hack and all-pass episodes (all seeds and sets).

before or after training; the visible-test reward did not teach the policy to copy them. Special-casing that copies no value is not detected by this check.

Evaluation noise. Every run evaluates the same base model at step 0. Across the 6 runs, all-pass on KodCode validation ranges from 65.1% to 65.8% (standard deviation 0.3 points), on MBPP+ from 54.3% to 55.6% and on HumanEval+ from 48.4% to 49.7%. Greedy decoding on vLLM is not bitwise reproducible across server states. The spread between seeds of the same reward is larger: up to 4.9 points of all-pass (standard deviation, vis_key on MBPP+).

8 Discussion

No sign that the visible tests were exploited. The concern behind this paper is that a policy rewarded by the tests it can see learns to satisfy those tests rather than to solve the problem. In our setting we saw no sign of it within 100 steps. Training raised the share of problems solved on all three sets, including the two the policy never trained on. The share of visible-test passes that are wrong fell by about half on KodCode validation and HumanEval+ and barely moved on MBPP+. Programs that copy values from the visible tests stayed as rare as they were in the base policy. The wrong programs that pass the visible tests are, before and after training, almost never copies of the tests: they handle the cases the examples show and get others wrong. A visible-test reward pays for them, but the base policy already wrote them, and on two of the three sets training made them a smaller share of the passes.

What the judge could do, and did. Offline, both judges rank a problem’s visible-test passes by whether they also pass the held-out tests better than chance (§6), and both catch hard-coded lookups. In training, the judge factor did not change the outcome beyond the variation between seeds: no difference on KodCode validation and HumanEval+, and on MBPP+ a gain in one seed and none in the other. What was left to suppress were not copies of the tests but solutions that are wrong on cases the examples do not show. Whether a judge factor helps where a policy does learn to special-case, with weaker tests, harder problems or longer training, these runs cannot say.

Better tests remove a few false passes, not unsolved problems. The oracle reward, which pays only for programs that pass every test, ended with fewer wrong programs passing the visible tests than vis on all three sets (by an interval that excludes zero on MBPP+ only), and with the same share of problems solved. Within 100 steps, knowing which visible-test passes are wrong did not teach the policy to solve more problems; it taught it to pass the visible tests less often when it could not solve them. A perfect judge of correctness, used as the factor, would give exactly this reward.

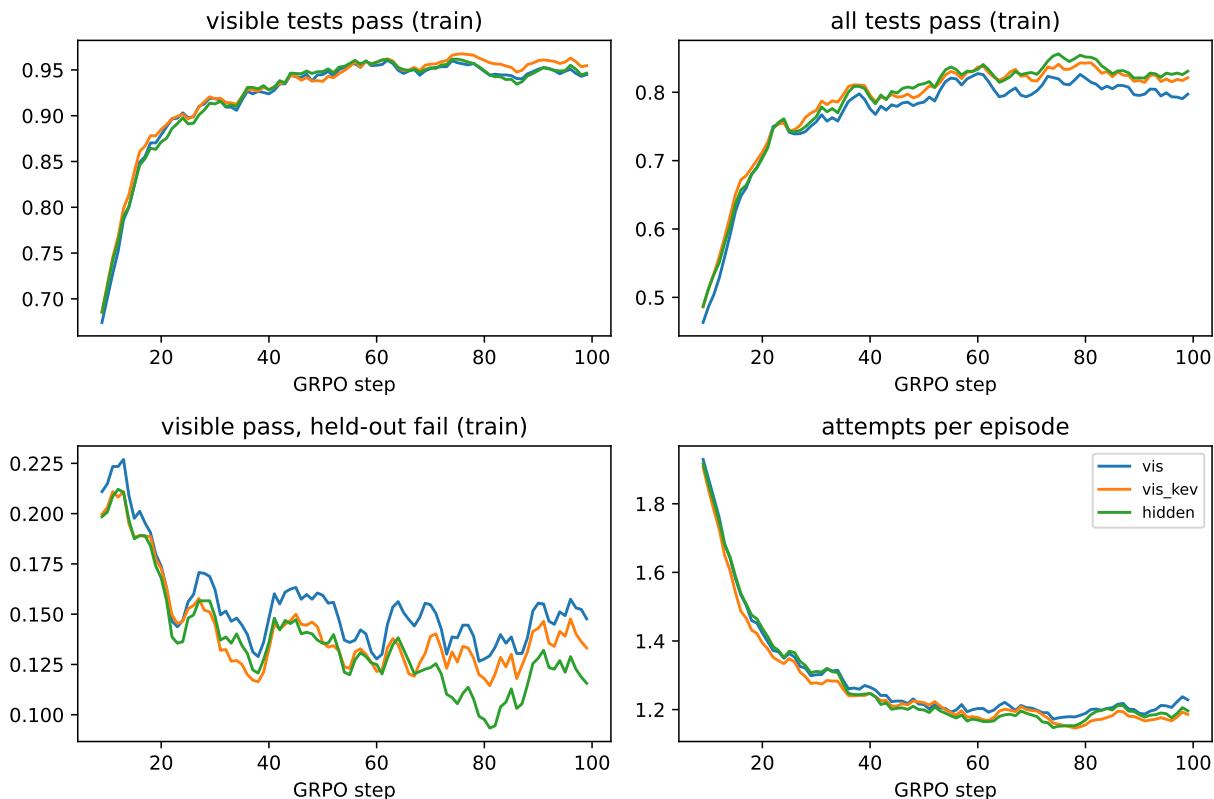


Figure 1: Training curves (seed mean, 10-step moving average): visible tests pass, every test passes, visible tests pass but a held-out test fails, and attempts per episode, over the training episodes.

Limitations. One policy size (4B), one task format (a single function, up to three attempts with the first failing test as feedback), 100 steps, one learning rate taken from Part II, and two seeds per reward, which disagree by up to 4.9 points (standard deviation) on MBPP+. The intervals resample problems and not seeds. The held-out tests are those of KodCode and EvalPlus, and a program that fails them is wrong by their specification, which can be stricter than the problem statement. The special-casing check finds only programs that contain a value from the visible tests; special-casing without such a value (a branch on the input’s length, say) is not detected. Other experiments shared the GPUs during training.

Reproducibility

Code, prompts, the data preparation (problem filtering and the split into visible and held-out tests), raw logs (every GRPO step and episode of the final evaluations) and the offline study’s programs and judge scores are in the repository ([experiments/part3a_code/](#)). The tables and every number in the text are generated from them by `dcjev/code/judge_study.py`, `dcjev/analysis/part3a.py` and `dcjev/analysis/paper_numbers.py`.

References

- [1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language

models. *arXiv preprint arXiv:2108.07732*, 2021.

- [2] Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.
- [3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [4] Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, Nicholas Schiefer, Ryan Soklaski, Alex Tamkin, Jared Kaplan, et al. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *arXiv preprint arXiv:2406.10162*, 2024.
- [5] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.
- [6] Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems*, 2022.
- [7] Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. *arXiv preprint arXiv:2503.02951*, 2025.
- [8] Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction. *arXiv preprint arXiv:2408.15240*, 2024.