Extended Abstract

Motivation Tool calls makes LLM use more expensive (latency, money, rate limits) for every call. Setting the model's budget at inference time, on the other hand, is more useful than one fixed at a single operating point during training. Thus, investigate the effect of per-prompt tool-call budgeting using a 0.5B policy and informing a budget K in the prompt. We also investigate how problems complexity relates to tool calls. We test teaching to solve the puzzle Countdown and providing an in-rollout Python interpreter.

Method We investigate this by implementing in the RLOO trainer: A Python interpreter that can test code from the model and feed the results back into the model's reasoning process, a budget K sent at prompt, a reward r = r_verifier $-$ α·max(0, ncalls $-$ K) with α = 0.8 which proioritizes correctness over tool calls, and a difficulty scorer. Our approach here is to make the tool-augmented rollout RL-correct. We mask the interpreter outputs and we keep the importance raito high, making sure that trained tokens are the sampled tokens.

Implementation We used Qwen2.5-0.5 from the SFT checkpoint and Modal H100s with vLLM.

Results Our results are unsuccesfull. they sow that the proposed approach fails at this scale. Our model fails to learn the budget K and makes it collapse to zero or one. Using instead SFT imitation teaches to obey the budget more efficacely and gives a graded control. Finally, the same graded knob can be achieved by simply forcing a budget at inference.(pass@1 0.18 to 0.41).

Discussion The negative result are due to the fact that mid-training accuracy scalars were too noisy to trust. We provide a SFT warm-start to make tool use emerge, at the expense of task accuracy (which collapses from 0.25 to 0.03 by 3000 steps).

Conclusion At this scale, deriving it from RL task reward alone fails and we obtain budget-conditioned tool use in only two ways: by teaching it (SFT) or by enforcing a cap at inference. At this scale, deriving it from RL task reward alone fails.

Budget-Conditioned Tool Use for Countdown Reasoning

Luca De Donno
Department of Mechanical Engineering
Stanford University
lucad@stanford.edu

Abstract

Tool calls makes LLM use more expensive (latency, money, rate limits) for every call. Setting the model's budget at inference time, on the other hand, is more useful than one fixed at a single operating point during training. Thus, we investigate the effect of per-prompt tool-call budgeting using a 0.5B policy and informing a budget K in the prompt. We also investigate how problems complexity relates to tool calls. We test teaching to solve the puzzle Countdown and providing an in-rollout Python interpreter. We implement in the RLOO trainer a Python interpreter that can test code from the model and feed the results back into the model's reasoning process, a budget K sent at prompt, a reward r = r_verifier $-$ $\alpha\cdot$max(0, ncalls $-$ K) with α = 0.8 which proioritizes correctness over tool calls, and a difficulty scorer. Our approach here is to make the tool-augmented rollout RL-correct: we mask the interpreter outputs and we keep the importance raito high, making sure that trained tokens are the sampled tokens. We used Qwen2.5-0.5 from the SFT checkpoint and Modal H100s with vLLM. Our results are unsuccesfull and sow that the proposed approach fails at this scale. Our model fails to learn the budget K and makes it collapse to zero or one. Using instead SFT imitation teaches to obey the budget more efficacely and gives a graded control. Finally, the same graded knob can be achieved by simply forcing a budget at inference (pass@1 0.18 to 0.41). At this scale, deriving it from RL task reward alone fails and we obtain budget-conditioned tool use in only two ways: by teaching it (SFT) or by enforcing a cap at inference.

1 Introduction

A simple way to make a reasoning model stronger is to give it a Python interpreter and let RL learn to use it. On its own this is not interesting, because the model would call the interpreter on every puzzle and the calculator would just solve the task. It gets interesting once each call has a cost. It gets more interesting still if the user can set the *budget* for those calls at inference time, instead of having it fixed at training time.

This matters because deployed LLM agents pay a real cost for every tool call: latency, money, rate limits. A model with a budget knob the user can turn at inference time is more useful than one trained at a single fixed operating point. The reason is simple: the same checkpoint can run cheaply when calls are expensive, and generously when they are not. We study this on Countdown, a multi-step arithmetic reasoning task. The model is given a set of numbers and a target, and it must produce an arithmetic expression that hits the target using each number once. Countdown is a good testbed because difficulty is analytically tractable: we can enumerate the valid solutions for a puzzle. So we can check whether the policy gives more tool calls to harder problems.

Research questions. (RQ1) Can a 0.5B policy trained with RLOO learn to obey a tool-call budget K given in the prompt? (RQ2) Does the resulting accuracy-vs-K curve behave sensibly, including for budgets *never seen during training* ($K = 4, 5$)? (RQ3) Does the policy allocate more calls to harder puzzles (difficulty-stratified allocation)? (RQ4) As a prerequisite, does a 0.5B model initialized from the course SFT checkpoint discover the tool format from RL alone, or must tool use be bootstrapped?

---

2 Related Work

Tool-call efficiency. Wang et al. ("Acting Less is Reasoning More", arXiv:2504.14870) add a reward that mixes correctness with a decreasing function of the tool-call count. They train Qwen2.5 with PPO and GRPO, and report up to 68% fewer tool calls at about the same accuracy. Their cost is a fixed penalty, not a budget set by the prompt, so the model is trained at one operating point. We instead put the budget in the prompt and sample it during training. This means a single model gives a controllable cost/accuracy trade-off.

Reward design for tool use. ToolRL (Qian et al., arXiv:2504.13958) argues that coarse answer-matching reward is not enough for tool learning, and that a finer-grained reward structure helps. We keep the Countdown verifier as the correctness signal but add a structured budget-violation term. We are also explicit about the guardrail that keeps the two terms in the right order.

Prompt-conditioned compute budgets. L1 / LCPO (Aggarwal & Welleck, arXiv:2503.04697, COLM 2025) train reasoning models to obey a length constraint given in the prompt. This gives a single model with a controllable accuracy-vs-compute knob. They condition on chain-of-thought length, not tool calls. We apply LCPO-style prompt conditioning to a discrete tool-call budget. Tool calls are the natural cost unit for an agent, so this is a different control surface from token length.

Cost-aware control beyond language. A similar trade-off shows up in robotics. Bohn et al. (arXiv:2011.13365) train an RL policy to decide when to recompute an expensive MPC controller, with a per-solve cost in the reward. The shared idea is learning when an expensive computation is worth its cost. We apply it to interpreter calls in a language model and add a user-set budget.

Tool-integrated RL. Recent work trains LLMs to call tools or search engines with RL, such as Search-R1 (Jin et al., arXiv:2503.09516) and Tool-Star (Dong et al., arXiv:2505.16410). These methods try to make tool use more effective. We ask a different question: given that the tool is used, can a prompt-set budget control how much it is used at inference time. The preference and policy-gradient machinery we build on is standard (DPO, Rafailov et al., arXiv:2305.18290; PPO, Schulman et al., arXiv:1707.06347; the role of the baseline in policy gradients, Mei et al., NeurIPS 2022), and the Countdown RL setup follows TinyZero (Pan et al., 2025).

Gap. The gap is that none of the above stuidies has combined prompt-conditioned budgets rather than a fixed cost, difficulty-stratified allocation analysis on a task where difficulty is analytically tractable, and sub-1B scale. Sub-1B scale is also where the prerequisite question becomes sharp: whether tool use is even discoverable from RL. Together with a one-shot verify tool that is not load-bearing (later overturned by the search redesign, which makes the interpreter load-bearing), this is where our negative result lands.

---

3 Method

We add four components to the course RLOO pipeline. All of them start from the same SFT checkpoint. We describe each component, then the RL-correctness design that holds them together, then the configurations and how we sample the budget.

3.1 Integrations

(1) In-rollout Python interpreter (`rloo_trainer/tool_use.py`). Vanilla RLOO sampling is single-shot: vLLM decodes a whole response in one `generate` call. We need the model to call

a tool in the middle of a rollout, so we replace this with a token-id-driven continuation loop (generate_with_tool). Each round we sample with stop=["</code>", "</answer>"]. If the model closes a </code> block, we pull out the code, run it in a sandbox (run_code), and splice the result back as \n<output>\n{result}\n</output>\n. The loop then continues from the extended sequence. If the model closes </answer>, or hits the budget, the rollout ends. ncalls is the number of code blocks we executed. The sandbox runs the snippet in an isolated subprocess (python -I) in its own process group, with a wall-clock timeout, a CPU-time rlimit, a file-size rlimit, single-threaded BLAS, and a throwaway working directory. On timeout we kill the whole process group, so no orphan grandchildren survive. (We do *not* set an address-space rlimit on purpose, because per-thread virtual arenas scale with host core count and break numpy/BLAS imports. Compute is bounded by the CPU-time limit and the wall timeout instead.)

(2) Budget *K* in the prompt (build_tool_prompt). Before sampling, we feed the tool instructions and a one-shot worked example into the user turn. For the budget-conditioned configuration, we also feed "You may use the interpreter at most *K* times." We sample *K* per prompt in rloo.py (random.randint(k_min, k_max)). We test two ways of informing of the budged: either through reward and through the a hard ceiling on tool executions in the loop (max_tool_calls in the hard_budget configuration).

(3) Reward shaping. Our reward is

> r = r_verifier − α · max(0, ncalls − K), with r_verifier ∈ {0, 0.1, 1.0} and α = 0.8.

r_verifier is the unchanged Countdown verifier (Section 4). The penalty is zero while within budget and grows linearly with the over-budget count. We use this reward in the existing leave-one-out baseline, so we do not change the RLOO update math at all. The baseline is computed from whatever reward tensor it receives. It is also worth mentioning thatt we prioritize correct responses over coprrect budget limit.

(4) Difficulty scorer (evaluation/countdown_difficulty.py). This is a module that give each puzzle a difficulty score equal to the size of its valid-solution set. We compute this by enumerating arithmetic combinations of the given numbers and counting the ones that hit the target (reusing the verifier's validate_equation/evaluate_equation). We use this only for the Phase-4 allocation analysis.

3.2 RL-correctness

If we just inster the the interpreter in the loop, RLOO gets corrupted. thus, we pay attention to two aspects. First, training tokens must equal sampled tokens. Each round feeds prompt_ids + response_so_far (token ids, not re-rendered text) to vLLM, and the response tokens we later train on are the same ids the vLLM outoutted.

Second, our interpreter tokens are environment, rather than actions. They are attended (attention_mask = 1), so the model can read them but masked out of the loss (is_response_token = 0) and excluded from the behavior log-probability mu. If we trained on them, we corrupt the policy gradient (rewarding the policy for environment tokens) and importance ratio collapses.

We make generation budget-aware per round: each round's max_tokens is the remaining budget, bounded by both the response cap and the context window. This means the response never overshoots the length cap and is never silently clipped after the fact. The result is that mu matches the update worker's sequence log-prob exactly. We verified this numerically and a rollout of 42 model tokens plus 22 interpreter tokens gives a mu equal to the sum of the 42 model-token log-probs, and a real-model check shows the masked sequence log-prob (-103.1) differs from the wrong value you would get by counting interpreter tokens (-126.2). So the mask is doing its job.

3.3 Configurations and budget sampling

All configurations start from the same SFT checkpoint and differ only in the tool/budget settings:

Config	Flags	Behavior
budget_conditioned	`--tool_enabled --budget_conditioned --alpha 0.8 --k_min 0 --k_max 3`	K sampled per prompt, in the prompt; over-budget penalty
free_tool	`--tool_enabled --alpha 0.0`	tool available, no penalty (does the model use it at all?)
fixed_cost	`--tool_enabled --alpha 0.8 --budget 0`	per-call penalty α·`ncalls`, no K in prompt (the prior-work operating point)
vanilla	reuse the no-tool RLOO checkpoint	no tool, RLOO baseline

Budget *K* sampling. During training, `K` $\sim$ `Uniform{0, 1, 2, 3}` per prompt (budget_conditioned). At evaluation we sweep `K` $\in$ `{0, 1, 2, 3, 4, 5}`. `K = 4, 5` are *unseen* during training, so they test whether the budget knob generalizes (RQ2). For fixed_cost there is no K in the prompt; for free_tool the penalty is off.

```
Algorithm: tool-augmented RLOO rollout + reward

generate_with_tool(prompt_ids, K):
    seq = prompt_ids                 # work in token-id space
    ncalls = 0
    loop:
        # sample one round, stop at a code block or the final answer
        tokens = vllm.sample(seq, stop=["</code>", "</answer>"],
                             max_tokens = remaining_budget(seq))  # budget-aware
        seq += tokens
        if ends_with(seq, "</answer>"):
            break                    # rollout done
        if ends_with(seq, "</code>"):
            code = extract_code(seq)
            result = run_code(code)              # hardened sandbox subprocess
            out = "\n<output>\n" + result + "\n</output>\n"
            seq += tokenize(out)                 # splice result back in
            mark_as_environment(out)             # attended, masked from loss + mu
            ncalls += 1
            if ncalls >= K_hard or remaining_budget(seq) <= 0:
                break                # hit the budget ceiling
    return seq, ncalls

# per-rollout reward (passed straight to the RLOO leave-one-out baseline)
r = r_verifier - alpha * max(0, ncalls - K)      # r_verifier in {0, 0.1, 1.0}, alpha = 0.8
```

3.4 The canonical methods: the search redesign and the budget variants

The first experiment showed the one-shot verify tool is not load-bearing (Section 5), so we changed the method in a few ways. These are the configurations that produce the main results in Sections 6 and 7.

Search warm-start. Instead of teaching the model to check one finished expression, we SFT it on trial-and-error search trajectories: try a candidate, print it, read the output, try another, and answer once one hits the tagret. This makes the interpreter drive the search instead of only checking a choice the model already made.

Reward variants. On top of the search warm-start we test three modes. (a) free_tool uses the verifier reward only, with no budget term. (b) budget_util adds a small bonus for each distinct, in-budget, productive call (a call that prints a valid new candidate), so extra calls are worth something when the

budget is larger, not just costly. (c) hard_budget uses no penalty, but enforces the prompted K as a hard per-prompt cap on tool calls during the rollout.

Teaching the budget by SFT (balanced-*K* and stop-at-*K*). To teach the budget by imitation we build SFT data where the number of tool calls in each trajectory is exactly equal to the prompted K. The set is balanced over K in $\{0, 1, 2, 3\}$, and it includes $K = 0$ demonstrations that use no tool at all. About 46% of the $K \geq 1$ demonstrations saturate the budget: the model uses exactly K calls failing, and then outputs the best guess. These stop-at-K demonstrations teach the model to stop at the budget instead of searching to its natural depth.

Enforcement and ablation at inference (no training). The hard-cap takes the trained free_tool policy and caps its tool calls at K at inference. The dead-interpreter ablation re-runs a policy with the interpreter replaced by a stub that returns a fixed string, isolating how much the tool contributes.

---

4 Experimental Setup

Model and initialization. We use Qwen2.5-0.5B Base, initialized from the SFT checkpoint.

Task and dataset. We use Countdown 3-to-4 (`asingh15/countdown_tasks_3to4`). Each example gives a set of numbers and a target. the model outputs an expressions using a number once and then it compares the naswer to the target.

Verifier (reward). We return 0.0 when no `<answer>` tag is present, 0.1 for a parseable answer in the right format but is incorrect, and 1.0 for valid format and corrct answer. `validate_equation` checks that we use the available numbers. `evaluate_equation` evaluates whether the expression is correct.

Compute and infrastructure. We train on Modal H100 (torch 2.9.1, vLLM 0.15.1).

Baselines and metrics. Baselines are the four configurations (vanilla = the no-tool RLOO milestone). We report accuracy as the fraction of rollouts with `r_verifier = 1.0`. We report pass@1 and pass@k, executed tool calls, and difficulty-stratified allocation on a held-out set of 50 prompts x 16 samples; all confidence intervals are 95% paired bootstrap over the 50 prompts.

Reproducibility summary. Table 1 lists every configuration we report, with its initialization, reward or budget mode, training details, seeds, and headline eval. All evaluations use the same 50 prompts x 16 samples, and all confidence intervals are 95% paired bootstrap over the 50 prompts.

Table 1. The configurations we report. "Init" is the starting checkpoint; "eval-only" means we do not train, we change inference.

Run	Init	Reward / budget mode	Training	Seeds	Headline eval
course SFT	Qwen2.5-0.5B base	SFT on Countdown	provided	1	~0.25 acc
free_tool (search)	search warm-start	verifier only ($\alpha = 0$)	RLOO	1	pass@1 0.41 / 0.48
budget_conditioned	search warm-start	verifier $-$ 0.8·max(0, ncalls$-$K), K in prompt, K ~ U[0,3]	RLOO	1	tool use $\rightarrow$ 0
hard_budget	search warm-start	verifier; K enforced as a hard per-prompt cap	RLOO	1	flat ~1.72 calls

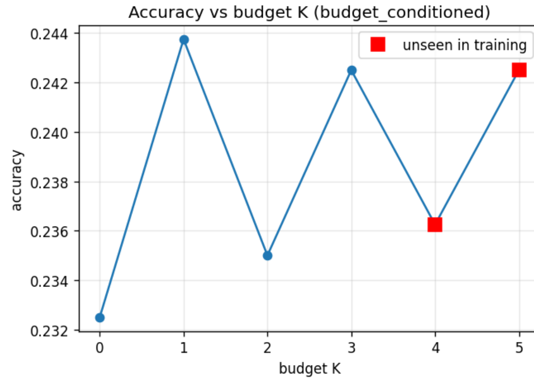

Figure 1: Accuracy vs. budget K for the budget-conditioned policy ($K = 0 \ldots 5$, with $K = 4, 5$ unseen in training). The curve is flat and `ncalls=0` at every K.

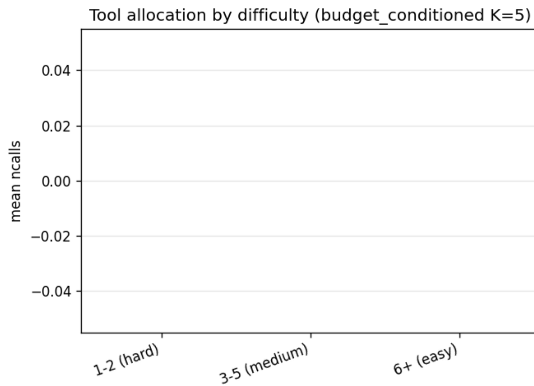

Figure 2: Mean tool calls vs. puzzle difficulty for the budget-conditioned policy. The policy never calls the tool, so allocation is zero in every bucket.

Run	Init	Reward / budget mode	Training	Seeds	Headline eval
budget_util	search warm-start	verifier + bonus for distinct in-budget calls	RLOO	1	1 ritual call
SFT-taught (balk2)	course SFT	SFT imitation of balanced-K + stop-at-K data (2400 traj)	SFT: 400 steps, lr 1e-5, batch 8	2	calls 0/1.0/1.9/2.7
hard-cap frontier	= free_tool	enforced cap K = 0. . . 4 at inference	eval-only	1	pass@1 0.18 → 0.41
dead-interpreter	= free_tool / vc_free	interpreter returns a stub	eval-only ablation	1	live − dead +0.21 / +0.27

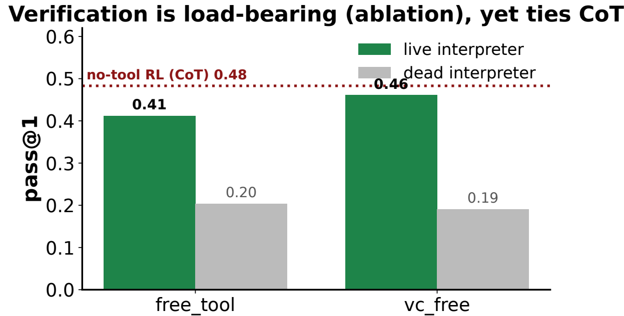

Figure 3: The interpreter is load-bearing: live vs. dead-interpreter pass@1 for the search policies, against the no-tool RL (CoT) line.

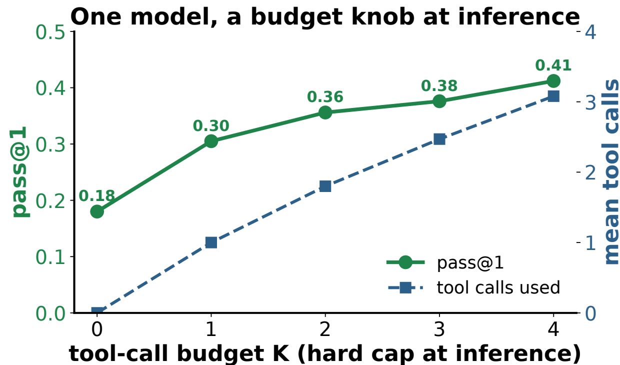

Figure 4: An enforced budget is a controllable knob: pass@1 and mean tool calls vs. the hard cap K at inference, from one checkpoint.

5 Results: verify-tool experiment

Three prerequisites set up the main results. We consider our results valid if the rollout is correct (IW≈0.90 and ≈1.0 on-policy) and interpreter tokens are masked from the loss. We analyze the effect of warm-start in 50 steps, 300 steps, and 3000. We observe that 3000 steps collapses accuracy from 0.25 to 0.03. The model forgets how to solve the task. On the other hand, 300 steps keep it at 0.19, so we run RL from the 300 steps warm-start.

The first experiment failed informatively. We train a verify-tool baseline (budget_conditioned $\alpha = 0.8$, $K \sim \{0,1,2,3\}$; free_tool $\alpha = 0$; fixed_cost) from this initialization. As expected, the use of the tool here has no effect and calls stay at zero. The one-sided penalty drives the policy to never call the tool.

---

6 Results: trial and error search

The setup. The first experiment (Section 5) showed the one-shot verify tool is useless. The hypothesis is that if we instead teach the tool a strial-and-error search it becomes useful

the re-run below is the main result.

Search warm-start result. The search warm-start works as designed. The training-set harvest is viable: 66% of puzzles (2,626 / 4,000) yield both wrong and correct model-generated answers (mean 6.3 wrong, 2.1 correct each). After the warm-start, the policy starts to learn to call the tool. The interpeter is in fact called in 100% of rollouts (mean 3.3 calls) and accuracy rises from 0.19 to 0.37. So trial-and-error search is useful.

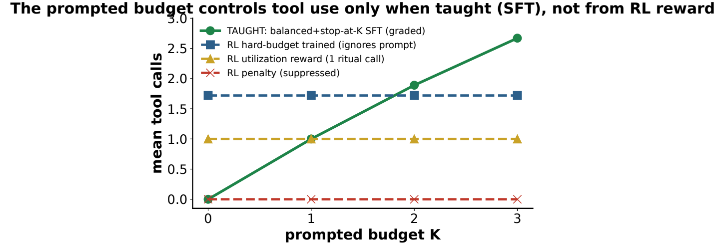

Figure 5: The prompted budget controls tool use only when taught (balanced + stop-at-K SFT, rising) and not from RL reward (flat).

RL from the search initialization. With no call penalty RLOO keeps the search and does trial-and-error (for example it tries 96−63+95=128, says "not 64, try another", then 96−95+63=64 and answers it). We also observe that when the search finds a correct candidate, it always commits to it (realization 1.000). From the same useful-tool init the one-sided over-budget penalty collapses tool use to `ncalls = 0`. This reveals that the one-sided penalty punishes over-budget search that drives accuracy.

Results. On the held-out test set (50 puzzles x 16 samples, paired bootstrap), free_tool reaches pass@1 0.41 (a replicate run reached 0.48; the gap is likely small-n run/config variance on n=50). The dead-interpreter ablation costs +0.21 to +0.27 pass@1 (both 95% CIs exclude zero; **Figure 3** shows live vs dead-interpreter pass@1 against the no-tool CoT line). About 95% of correct answers carry a verifier-valid printed expression, 0% echo-hacking and the soft over-budget penalty collapses tool use to zero.

Enforced budget The prompted budget K carries no signal (Section 5.4), but the budget mechanism does. by enforcing the budget at inference we obtain a monotone, controllable accuracy/cost frontier on the same held-out test (**Figure 4,** `fig_hardcap_frontier`, plots pass@1 and mean calls vs the hard cap):

hard cap K	0	1	2	3	4 (uncapped)
pass@1	0.180	0.305	0.356	0.376	0.412
mean tool calls	0.0	1.0	1.8	2.5	3.1
genuine-grounded / correct	0.00	0.59	0.83	0.93	0.95

The first two calls buy most of the measured gain (cap2 minus cap0 = +0.176 [+0.10, +0.26]; cap1 minus cap0 = +0.125 [+0.06, +0.19], both significant), and the curve saturates afterward (cap3 minus cap2 = +0.020, n.s.). The gain is attributed to the tool work and as the cap rises tool-verified correct rollouts rise (0, 145, 237, 281, 314) and prior-only correct guesses fall (144, 99, 48, 20, 16).

Soft vs hard budget. Training a policy with K imposed as a hard limit (no penalty) keeps the tool alive and within budget. However, under a fixed permissive limit the call count becomes constant across the prompted K (about 1.76 calls whether the prompt says K=0 or K=3) and the model ignores the instruction. The modulation is environmental and not learned from the prompt. Teaching the model to read and respect the prompted budget without enforcement requires a reward that makes extra in-budget calls valuable when K is larger (a one-sided penalty only makes calls costly). We report this experiment (a utilization reward over distinct in-budget search candidates) below.

Scope. The interpreter is significant *for the tool-trained policy* (the ablation above), but at 0.5B it does not beat a pure-CoT policy on net accuracy. An equally-RL-trained no-tool policy reaches pass@1 about 0.48 with zero code blocks, tied with free_tool, and it wins on coverage (pass@16 about 0.74 vs about 0.53). The tool does real, controllable, verifiable work rather than a net win at this 0.5B scale.

7 Main results: teaching vs. enforcing the prompted budget

The RL-trained prompted budget is ignored, but an enforced cap works, and SFT can teach a graded budget.

Teach (utilization reward) collapses to one call. A one-sided penalty only makes calls costly, never makes in-budget calls valuable, so we tried a budget_util reward that pays for distinct in-budget search candidates. It kept the tool alive but landed on the same flat collapse of 1.0 across prompted K. This held on both mixed puzzles (most solved in one call, so no budget-dependent work) and hard 4-number puzzles (so hard at 0.5B that the grounded bonus rarely fired and the signal starved).

Teach (balanced-K SFT) is what works. The RL failures pointed at the initialization: the search warm-start was SFT'd on K-imbalanced data (no K=0 demos, calls usually below the prompted K), so it ignored the budget (~4.5 calls at every K). Building the SFT data with a **balanced** K in {0,1,2,3} where the call count exactly matches the prompted K (including K=0 no-tool demos) gives the first budget response because the expert demos always solved within K and never learned to stop. Adding **stop-at-*K* (budget-exhausted) demonstrations** (use exactly K calls, then commit; ~46% of the $K \geq 1$ demos) fixes this.

prompted K	0	1	2	3
mean tool calls, seed 1 (exec = attempts)	0.00	1.00	1.89	2.67
mean tool calls, seed 2	0.00	1.00	1.87	2.64
pass@1 (seed 1)	0.212	0.098	0.186	0.295
genuine-grounded / correct	n/a	1.00	1.00	1.00

The call counts replicate across two SFT seeds (seed 2: 0.00/1.00/1.87/2.64), with tight CIs (K = 2: 1.89 [1.85, 1.92]; K = 3: 2.67 [2.57, 2.76]), so this is not a single-run artifact. **Figure 5** (`fig_budget_modulation`) plots the rising taught curve against the flat RL methods. The calls are load-bearing at K = 3 (live 0.295 vs dead-interpreter 0.147, paired +0.15 [+0.10, +0.19]) and ~100% genuine. This is the prompted-budget control we sought, by **imitation** rather than RL, at lower accuracy than unbudgeted free search. There are of course some cavaets: accuracy is non-monotonic at the low end (K=1 at 0.10 dips below K=0's 0.21, since committing after one failed search beats neither), recovering by K = 3. Also, budget boundary costs, of course, accuracy.

In-range vs extrapolation. We also tried to prompt new unseen budgets K = 4, 5 (training capped at 3). The model interestingly saturates at the trained budget using only 2.75 and 2.68 calls. It looks like it learned a monotonic mapping instead.

---

8 Discussion

Why RL did not learn the prompted budget. We attribute this to the one-sided over-budget penalty. Since the call is penalized and we can solve the puzzle also without the tool, RLOO drives calls to zero. We can make the tool survive adding a utilization term, even though the model settles on the cheapest behavior that get rewarded and ignores K. At 0.5B the interpreter only ties chain-of-thought in net value, so there is little reward signal to keep budgeted tool use. With imitation the model copies the budget-to-calls mapping directly from demonstrations.

Broader impact. A cheap inference-time tool-budget can cut deployment cost, because a single checkpoint can be run at fewer tool calls when calls are expensive and at more when they are not, and most of the gain comes from the first one or two calls. Our negative result gives us a lesson: do not assume RL learns a prompted budget on its own.

Limitations. the main limitation is that the small scale (0.5B). Results might be different for larger models.

Difficulties encountered. Beyond the bugs in Section 5.1: GPU availability (driver/kernel mismatch on both local boxes) forced the move to Modal H100; and several Modal-image and scheduling footguns (root-`.venv` reimaging, `.spawn()` detachment, warmup/LR-schedule incompatibility) had to be fixed before training would run end-to-end.

9 Conclusion

We built an RL-correct budget-conditioned tool-use extension to RLOO pipeline. We learned that we can make the tool-call budget significnt in two way: through treaching and through hard limit at inference. At this scale, the second respects the budget more than the first. Reward alone is not enough at 0.5b to innject the notion of budgeted tool call and the penalty suppresses the use tp zero calls. Re-teaching the calculator as trial-and-error search is what makes it significant in the first place (dead-interpreter ablation +0.21 to +0.27 pass@1), since a one-shot verify tool cannot help a search task.

Future directions. (1) Test at larger scale and on a different task where using the tool clearly wins. (2) Try a two-sided budget reward that pays for productive in-budget calls, to see if RL can learn what imitation does here.

---

10 Team Contributions

Solo project. All work (design, implementation of the four components, RL-correctness, analysis, debugging, training/eval infrastructure, and this report) was done by **Luca De Donno**.

References

Pranjal Aggarwal and Sean Welleck. 2025. L1: Controlling How Long A Reasoning Model Thinks With Reinforcement Learning. In *Conference on Language Modeling (COLM)*. arXiv:2503.04697.

Eivind Bøhn, Sebastien Gros, Signe Moe, and Tor Arne Johansen. 2021. Optimization of the Model Predictive Control Update Interval Using Reinforcement Learning. *IFAC-PapersOnLine* (2021). arXiv:2011.13365.

Guanting Dong, Yifei Chen, Xiaoxi Li, Jiajie Jin, Hongjin Qian, Yutao Zhu, Hangyu Mao, Guorui Zhou, Zhicheng Dou, and Ji-Rong Wen. 2025. Tool-Star: Empowering LLM-Brained Multi-Tool Reasoner via Reinforcement Learning. arXiv:2505.16410 [cs.CL]

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. arXiv:2503.09516 [cs.CL]

Jincheng Mei, Wesley Chung, Valentin Thomas, Bo Dai, Csaba Szepesvári, and Dale Schuurmans. 2022. The Role of Baselines in Policy Gradient Optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 17818–17830.

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. TinyZero. `https://github.com/Jiayi-Pan/TinyZero`.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. 2025. ToolRL: Reward is All Tool Learning Needs. arXiv:2504.13958 [cs.CL]

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems (NeurIPS)*. arXiv:2305.18290.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG]

Hongru Wang, Cheng Qian, Wanjun Zhong, Xiusi Chen, Jiahao Qiu, Shijue Huang, Bowen Jin, Mengdi Wang, Kam-Fai Wong, and Heng Ji. 2025. Acting Less is Reasoning More! Teaching Model to Act Efficiently. arXiv:2504.14870 [cs.AI]