REWARDS FOR BUDGET-AWARE MULTI-TURN AGENTS THAT ADAPT TO ARBITRARY TEST-TIME CONSTRAINTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model agents may be given an inference budget and asked to maximise success within it. Budget-aware training has largely focused on controlling the cost of a single model generation at a single budget setting. However, multi-turn agents face a different problem, as a shared budget must be allocated across multiple environment interactions. This allocation is often managed by an external harness, adding inference-time complexity while leaving open whether the agent itself can learn to manage its own resources. We ask whether agents can be trained to reason about user-specified budgets and adapt when and how much they spend to maximise task success within them.We adapt existing length penalties to the multi-turn setting, propose new alternatives, and systematically study which objectives best teach agents to manage their resources effectively. We train Qwen3.5 models at 4B and 9B on AppWorld with budgets over tokens, API calls, and environment interactions, and evaluate generalisation to unseen budgets, to distribution shift, and to a second agentic benchmark. Training substantially improves both within-budget success and budget compliance over untrained models, and a single policy generalises this behaviour zero-shot to budgets, shifted distributions, and environments it never trained on, adjusting how much it spends to both the stated budget and the difficulty of the task. Simple gating, which withholds reward past the budget without pricing overspend magnitude, is highly competitive, suggesting that precise reward-shaping is not required for controlling overspend severity. Budgeting across multi-turn interactions can therefore be learned by the policy itself, without the external harnesses, predictors, or routers that prior approaches rely on.

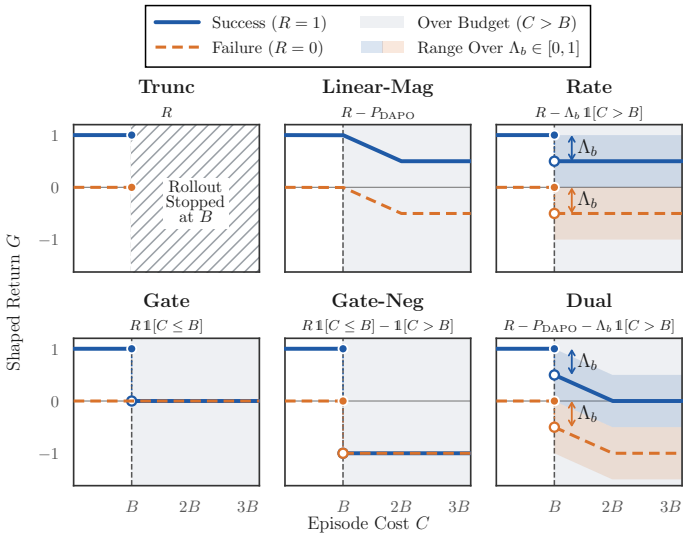
1 INTRODUCTION

Agents can solve harder tasks by spending more at test time, thinking for longer and taking more actions in an environment (Snell et al., 2024; Guo et al., 2025; Yu et al., 2025). Current frontier agents give users some control over the effort in terms of generated tokens, tool calls, or environment actions (Liu et al., 2025b). To respond to different budgets, an agent must learn different strategies to solve a task with maximum success rate, conditional on the allowance it receives.

Budget-aware training has largely focused on controlling the cost of a single model generation (Aggarwal and Welleck, 2025; Liu et al., 2026; Wen et al., 2025). However, multi-turn agents face a different problem. A shared budget must be allocated across multiple turns. At each turn, the policy must decide when and how much to spend given what remains (Liu et al., 2025b). Such allocation is often handled by external inference-time machinery, including planning, harnesses, or routers, rather than learned by the policy itself (Zhang et al., 2026; Liu et al., 2025b). This adds inference-time complexity and leaves open whether resource allocation can be learned by the policy itself, for more direct resources like steps, or more abstract ones like tokens. Different constraints may have different optimal behaviours, after all. This raises our first question: *can we train a multi-turn agent to maximise task success under dynamically specified budgets, across different inference resources?*

Length penalties, subtracted from the reward, penalise exceeding the budget while still allowing to collect overlong trajectories that may provide useful information about how to solve the task, particularly when within-budget successes are rare. Some current frontier models apply linear penalties (Yu et al., 2025; DeepSeek-AI, 2026), while others optimise under step functions (Kimi Team,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



Reward	Truncated	Step	Linear	Prices failure	Adaptive
`Trunc`	✓	✗	✗	✗	✗
`Linear-Mag`	✗	✗	✓	✓	✗
`Rate`	✗	✓	✗	✓	✓
`Gate`	✗	✓	✗	✗	✗
`Gate-Neg`	✗	✓	✗	✓	✗
`Dual`	✗	✓	✓	✓	✓

Figure 1: **Cost-conditioned rewards.** [Left] Shaped return G as a function of episode cost C for successful (R=1) and failed (R=0) trajectories. Shading marks costs beyond the budget B. For `Rate` and `Dual`, the band spans the range of the learned price Λ_b. [Right] Reward properties: *Truncated* stops the rollout at the budget. *Step* and *Linear* describe the penalty shape at and beyond the budget. *Prices failure* ranks an over-budget failure below a compliant one. *Adaptive* learns a separate penalty for each budget.

2026) and variable step functions inspired from constrained optimisation (Zhang et al., 2026; Altman, 1999; Stooke et al., 2020). It is currently unclear a priori which penalty mechanisms are useful, especially when training across dynamically specified budgets in multi-turn tasks. This raises our second question: *which training objectives best teach multi-turn agents to maximise task success while complying with dynamically specified budgets?*

We introduce a framework for training a single policy across dynamically specified episode-level budgets. The total and remaining budget are provided to the policy in natural language throughout each interaction, allowing it to learn how to allocate its resources as the episode unfolds. Within this framework, we adapt existing objectives that penalise resource use or budget violations to the multi-turn, multi-budget setting and propose new alternatives, including simply optimising success within budget without explicitly penalising a model for a lack of compliance. This enables a systematic study of which training mechanisms best induce budget-aware behaviour. We train and evaluate on AppWorld (Trivedi et al., 2024) with budgets over generated tokens, environment interactions, or tool calls. We evaluate generalisation to budgets unseen during training and under distribution shift to ASTRA (Xiu et al., 2026) and the AppWorld Test-Challenge.

Across budget types, training substantially improves both within-budget success and budget compliance over untrained models. A single policy learns genuine conditional resource allocation, adjusting how much it spends to both the stated budget and the difficulty of the task, and retains this behaviour across unseen budgets and distribution shifts. Strong budget awareness does not require directly penalising the magnitude of overspend, since simple gating is highly competitive without.

Our contributions are:

- To the best of our knowledge, we are the first to consider the problem of training a single multi-turn agent across dynamically specified resource-budgets and resource types. We define a general training framework to study this problem, including specifying relevant evaluation metrics.
- We show that multi-turn agents can learn to allocate their resources under dynamically specified budgets, improving task success and compliance across tokens, environment turns, and tool calls, and that this behaviour generalises zero-shot to unseen budgets, under distribution shift, and to a second agentic benchmark out-of-distribution.

- We survey and generalise cost-conditioned rewards popular in literature, and propose new ones.
- We find that a well-shaped magnitude penalty is not required for controlling overspend severity, since a simple objective that withholds reward past the budget without pricing overspend magnitude is highly competitive.

2 Related Work

Token budgets. On-policy budget-aware without external harness training has primarily considered controlling the length of a single model generation, rather than allocating resources across a multi-turn episode. Methods optimise adherence to token budgets (Aggarwal and Welleck, 2025; Liu et al., 2026; Han et al., 2025) or learn levels of reasoning effort (Arora and Zanette, 2025), largely in mathematical reasoning. In these settings, resource use is determined within one generation and tokens are the quantity being controlled. Multi-turn agents face a different problem: a shared allowance must be allocated across successive generations and environment interactions while solving the task, and the constrained resource may instead be tool calls or environment steps. Wen et al. (2025) trains across a curriculum of token budgets, but through inserted control tokens and decoder-side truncation rather than a policy that regulates its own spending. DAPO (Yu et al., 2025) introduces an overspend penalty for a fixed generation-length budget, while Kimi K3 (Kimi Team, 2026) considers multiple effort levels in agentic domains by training separate experts before distillation using a step function reward shaping. We instead study how a single policy can be trained across dynamically specified, episode-level budgets over multiple resource types.

Budget feedback and allocation. Budget-aware behaviour can also be introduced without training it into the task-performing policy. Liu et al. (2025b) show that increasing a search agent's tool-call budget alone does not yield corresponding gains, while explicitly reporting the total and remaining budget can improve scaling for large models when combined with planning and verification strategies. We find that smaller models respond much less to the same prompt-based remaining-budget signal as shown through evaluation of untrained models in Figure 2. Other approaches predict how much compute a task requires (Damani et al., 2025) or route between cheaper and more expensive models under a budget (Zhang et al., 2026). At inference time alone, Li et al. (2026) search a value tree on multi-hop question answering, scaling node values by the remaining resource ratio so that search shifts from exploration to exploitation as the budget depletes. These methods place resource allocation in inference-time planning, harnesses, or routers. Our framework instead trains the task-performing policy itself to condition its actions on the total and remaining budget innately.

3 Budget-Aware Policy Optimisation

3.1 Preliminaries

We write $\mathcal{P}(\mathcal{Y})$ for the set of probability distributions over a set $\mathcal{Y}$, and model a task as a POMDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \Omega, \mathcal{O}, R, \gamma, \mu_0 \rangle$. Here $\mathcal{S}$, $\mathcal{A}$ and Ω are the state, action and observation spaces. The transition and observation kernels are $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\mathcal{S})$ and $\mathcal{O} : \mathcal{S} \times \mathcal{A} \to \mathcal{P}(\Omega)$, with R the reward, which Section 3.2 specialises to a binary terminal reward, $\gamma \in (0, 1]$ the discount factor and $\mu_0 \in \mathcal{P}(\mathcal{S})$ the initial state distribution. The agent acts on the observation history $h_t := (o_1, a_1, \ldots, a_{t-1}, o_t)$, writing $\mathcal{H}$ for the set of all such histories, through a policy $\pi : \mathcal{X} \times \mathcal{H} \to \mathcal{P}(\mathcal{A})$, with tasks $x \sim p_{\mathcal{X}} \in \mathcal{P}(\mathcal{X})$ drawn from a distribution over the task set $\mathcal{X}$.

We optimise policies with Dr. GRPO (Liu et al., 2025c), an unbiased variant of group relative policy optimisation (GRPO, Shao et al., 2024) that baselines each rollout against its group mean and drops the normalisers that otherwise inflate the length of failed rollouts, which in our setting are largely the rollouts that overspend, see Appendix E.1 for more detail.

3.2 Budget-Augmented POMDP

Budget augmentation. We equip the task POMDP $\mathcal{M}$ of Section 3.1 with a cost map $\kappa : \mathcal{A} \to \mathbb{R}_{\geq 0}$ pricing each action in a user-chosen unit, whether environment steps, tool calls, or emitted tokens. An action is the whole string the policy emits at a turn, including thinking tokens. We lift

$\mathcal{M}$ to the *budget-augmented* POMDP $\tilde{\mathcal{M}} = \langle \tilde{\mathcal{S}}, \mathcal{A}, \tilde{\mathcal{T}}, \tilde{\Omega}, \tilde{\mathcal{O}}, R, \gamma, \tilde{\mu}_0 \rangle$ over $\tilde{\mathcal{S}} := \mathcal{S} \times \mathbb{R}$, whose states $\tilde{s}_t = (s_t, b_t)$ pair the environment state, s_t with a scalar remaining budget, b_t, the same augmentation used by budgeted MDPs in a different setting (Carrara et al., 2019). The augmented transition kernel factorises as

$$\tilde{\mathcal{T}}\big(s_{t+1}, b_{t+1} \mid s_t, b_t, a_t\big) = \mathcal{T}\big(s_{t+1} \mid s_t, a_t\big)\, \mathbf{1}\big[\, b_{t+1} = b_t - \kappa(a_t)\,\big], \tag{1}$$

so the environment dynamics are untouched and the budget evolves deterministically.

Budget-conditioned policy. Both B_0 and the remaining budget b_t are written into the text of the observations the agent sees (Appendix F.4), so augmented observations live in $\tilde{\Omega} := \Omega \times \mathbb{R}$, with

$$\tilde{\mathcal{O}}\big(o_{t+1}, b \mid s_{t+1}, b_{t+1}, a_t\big) = \mathcal{O}\big(o_{t+1} \mid s_{t+1}, a_t\big)\, \mathbf{1}\big[\, b = b_{t+1}\,\big], \tag{2}$$

which leaves the task observation untouched and reveals the budget exactly. An augmented history splits into the history h_t of $\mathcal{M}$ and the observed budgets, so the *budget-conditioned* policy

$$\pi(\cdot \mid x, h_t, b_t, B_0) \in \mathcal{P}(\mathcal{A}), \qquad \pi : \mathcal{X} \times \mathcal{H} \times \mathbb{R} \times \mathbb{R}_{>0} \to \mathcal{P}(\mathcal{A}), \tag{3}$$

can gauge both the absolute allowance left and the fraction b_t/B_0 still available against what the task demands. We write Π for the set of all such policies. The budget is therefore an explicit input to the policy throughout post-training rather than an external cap. Since Equation (3) conditions directly on B_0 and b_t, inserted in environment observations, a budget outside the training grid needs no change to the policy or the inference procedure, and Section 4 shows that budget-aware behaviour generalises to such unseenbudgets.

Episode termination. Each episode draws a task $x \sim p_{\mathcal{X}}$ and, independently of it, a budget $B_0 \sim p_B$ from the budget prior defined below, initialising the episode at $b_1 = B_0$ with the opening observation o_1 the prompt stating the task and its budget (see Appendix F.4 for the prompts in full). Most methods permit $b_t < 0$, so that the agent can overspend and receive learning signal from this, but might be penalised. A special `complete_task` action $a_\perp \in \mathcal{A}$ terminates the episode and, like any action, costs $\kappa(a_\perp) > 0$, so stopping also spends budget. With a truncation horizon $T_{\max} \in \mathbb{Z}_{>0}$, the episode ends at the stopping time $\tau \in \{1, \ldots, T_{\max}\}$ given by $\tau := \min\big(\inf\{t \geq 1 : a_t = a_\perp\},\, T_{\max}\big)$.

Reward and budget penalties. We score the completed trajectory $\zeta := (o_1, a_1, \ldots, o_\tau, a_\tau, o_{\tau+1})$ by a trajectory-level reward $R(\zeta) \in \{0, 1\}$ and the goal of the agent is to maximise its expected shaped reward, $\tilde{R}(\zeta, b) \in \mathbb{R}$, defined as

$$\max_{\pi \in \Pi} \; \mathbb{E}_{x \sim p_{\mathcal{X}},\, B_0 \sim p_B, \zeta \sim \pi(\cdot \mid x, B_0)}\big[\, \tilde{R}(\zeta, B_0)\,\big]\,. \tag{4}$$

We do not discount, so any pressure to finish early comes from the budget. We fix a finite set of $M \in \mathbb{Z}_{>0}$ *training* budgets $\mathcal{B} = \{b^{(1)} < \cdots < b^{(M)}\} \subset \mathbb{R}_{>0}$ and draw budgets uniformly at random during training. Spent budget, or cost C, in this work is always the budget spent over the entire, multi-step trajectory, as a fraction of the prompted budget B_0, and rewards are shaped as a function of the cost

$$C(\zeta, B_0) := \frac{1}{B_0} \sum_{t \leq \tau} \kappa(a_t) \in \mathbb{R}_{\geq 0}, \tag{5}$$

with $C \leq 1$ within budget and $C > 1$ over budget.

3.3 Cost-Conditioned Rewards

We study a range of rewards that incentivise budget adherence, drawn both from the literature and from new approaches we propose, motivated by heuristics, common practice in the field, and constrained optimisation. Figure 1 gives an overview. The rewards differ primarily in (a) whether they apply a step penalty at B_0 or a continuous one, and (b) whether they assign negative reward to overspending. We describe them below, with details in Appendix B.

`Trunc` (Liu et al., 2025a) is a baseline that does not permit any overspending. Every rollout is terminated at B_0, and the reward is the binary task reward obtained within B_0.

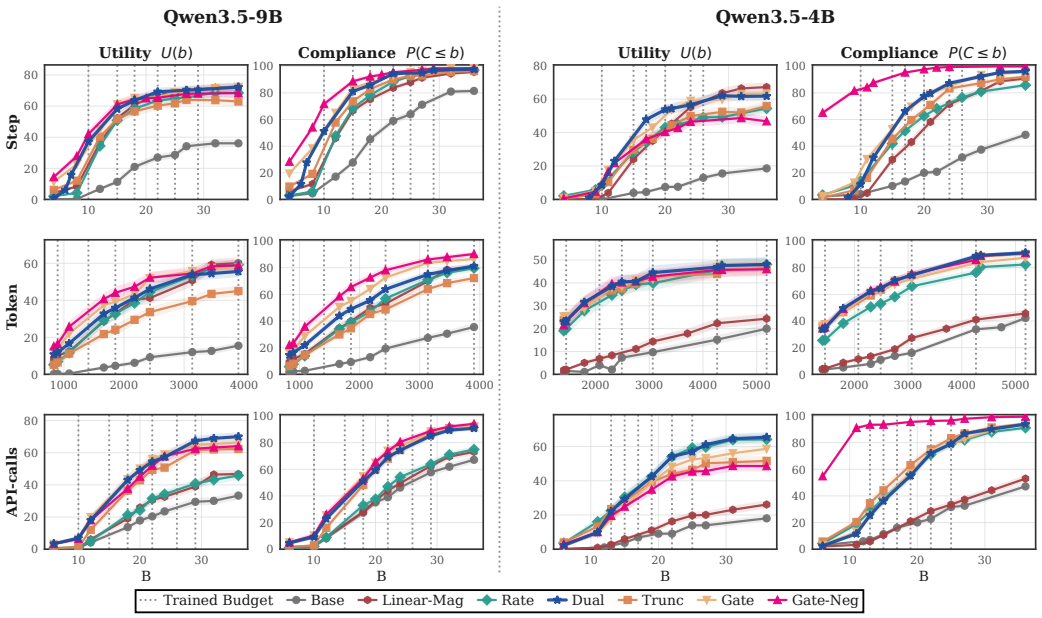


Figure 2: **In-domain budget utility and compliance.** Budget Utility $\mathcal{U}(b)$ and natural compliance $\mathrm{Comp}(b)$ against stated budget B on AppWorld for Qwen3.5-9B and 4B, under step, token, and API-call budgets. Each method is trained and evaluated in each budgeting unit separately. (Appendix F.3) The datapoints not on top of vertical dashed lines evaluate budget-generalization of the policy.

`Gate` relaxes this constraint. The policy may overspend, but any rollout that exceeds B_0 receives zero reward regardless of task success. We are not aware of this reward being used in the literature for training an agentic policy.

`Gate-Neg` (Kimi Team, 2026) assigns a reward of -1 whenever the policy exceeds B_0, regardless of task success. Unlike `Gate`, it therefore prefers a failure within budget to a failure over budget.

`Rate` tracks the overspend rate at each budget and applies a larger penalty the more often the policy exceeds that budget. We derive it from a Lagrangian perspective in Appendix D. Its price is capped at 0.5, so an overspending success always scores above any failure.

`Linear-Mag` has no step at B_0 and instead reduces the reward linearly beyond it. It was popularised by DAPO (Yu et al., 2025), and we generalise it to multiple budgets in Appendix C.1.

`Dual` combines a step penalty with a linear one. A learned Lagrangian step penalty prices *how often* rollouts at the given budget overspend, and a linear penalty prices *how much* the current rollout overspends. Each term subtracts at most 0.5. We study the dual optimisation in Appendix E and why the two penalties are complementary in Appendix D.5.

4 Experiments

Our experiments answer three questions. Section 4.1: can a single policy learn to adapt across a family of dynamically specified budgets and cost units? Section 4.2: which training objectives best teach this behaviour? Section 4.3: does the learned behaviour generalise to budgets, distributions, and environments the policy never trained on?

Experimental setup. We train Qwen3.5 models at 4B and 9B (Qwen Team, 2026), continuing training from the post-trained checkpoints (Appendices E.3 and F.7). Every method fine-tunes with Dr. GRPO (Liu et al., 2025c) from identical prompts, grids, seeds, differing only in the training return. `Base` is the untrained model. We train on AppWorld (Trivedi et al., 2024), an interactive coding-agent benchmark whose agent completes user queries in multi-turn Python code blocks. We evaluate on unseen AppWorld `test-normal` tasks, under distribution shift on App-

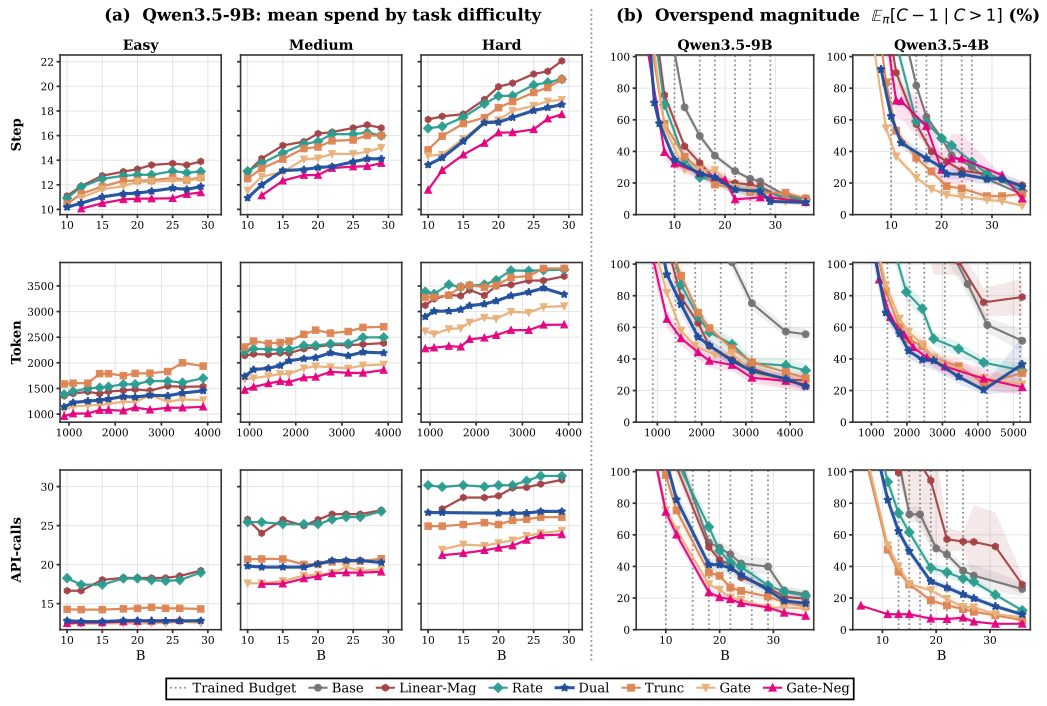


Figure 3: **Spend by task difficulty and overspend magnitude. (a)** Mean spend of the Qwen3.5-9B policies on AppWorld `test-normal` against budget B, split by task difficulty, under step, token, and API-call budgets. **(b)** Overspend magnitude, i.e. excess spend $\mathbb{E}_\pi[(C-1)_+ \mid C > 1]$ conditional on overspending, against stated budget B on AppWorld for Qwen3.5-9B and 4B, as a percentage of the budget and clipped at 100%.

World `test-challenge`, and out-of-domain on ASTRA (Xiu et al., 2026), a multi-turn personal-assistant benchmark (Appendix F). We consider three budget types through the cost map κ: *steps* (each agent turn), *tool calls* (each API call), and *tokens* (all generated tokens, including reasoning). At each budget b we measure *budget utility* $\mathcal{U}(b)$, the probability of solving a task within budget, and *compliance* $\mathrm{Comp}(b)$, how often the agent self-terminates a rollout within budget if not cut off:

$$\mathcal{U}(b) := \mathbb{E}_{x\sim p_{\mathcal{X}_{\text{test}}},\zeta\sim\pi(\cdot|x,b)}[R(\zeta)\,\mathbf{1}[C(\zeta,b)\le 1]]\,, \tag{6}$$

$$\mathrm{Comp}(b) := \mathbb{P}_{x\sim p_{\mathcal{X}_{\text{test}}},\zeta\sim\pi(\cdot|x,b)}(C(\zeta,b)\le 1)\,, \tag{7}$$

averaged over held-out tasks $\mathcal{X}_{\text{test}}$, with rollouts cut off at B_0 for $\mathcal{U}(b)$ and left uncut for $\mathrm{Comp}(b)$.

4.1 Agents Learn to Adapt Across Dynamically Specified Budgets

Training substantially improves success within budget. Figure 2 shows results for Qwen3.5-9B and 4B on AppWorld `test-normal`. At the budget of 36, the 9B and 4B `Base` models achieve only 37% and 19% budget utility respectively, and for tokens the 9B `Base` complies on fewer than half of episodes across the evaluated budgets. The limited performance can be explained by not having trained on Appworld, but not adapting to prompted budgets is an issue. Training raises budget utility throughout the budget range, including at unseen budgets both inside and beyond the training grid. Note that a single policy amortises training across all budgets in the training grid. Compliance remains below 100% when budgets are tight, but perfect compliance is not necessarily desirable: terminating early can increase compliance by giving up on tasks that additional resources might still solve.

Agents allocate resources according to budget and task difficulty. The utility and compliance curves in Figure 2 suggest that policies trained with budget penalties respond to the stated budget, which is further supported by Figure 3(a). Under step and token budgets, mean spend increases smoothly with the stated allowance rather than remaining at a fixed level, showing that training does

not simply produce a globally spend-conservative policy. Spend also increases with task difficulty, most clearly on hard tasks, indicating that agents learn to allocate additional resources where they are most needed. API-call budgets show a different pattern. Mean spend increases substantially with task difficulty but varies comparatively little with the stated allowance, suggesting that agents adapt API-call use to task demands but less readily to the available call budget.

Step penalties induce more conservative resource use and higher compliance. The choice of reward also changes how aggressively agents spend their budgets. `Gate-Neg` is the most spend-conservative policy across much of Figure 3(a), while still adapting its resource use to both the stated budget and task difficulty. Its stronger penalty for crossing the budget therefore compresses overall resource use without eliminating task-adaptive budget allocation. More generally, the step-penalty objectives (`Gate-Neg`, `Gate`, `Dual` specifically) achieve high natural compliance in Figure 2, indicating that directly distinguishing compliant from over-budget trajectories is particularly effective for learning budget adherence across resource types.

Tight budgets change how agents act. We can see agents adapting to specified budgets in alternative ways as shown in Table 1, which shows that voluntarily terminating episodes issue more API calls per step as the step budget tightens for both `Dual` and `Gate`, rising to 1.89 calls per step at a budget of 10 from around 1.5 to 1.6 at a budget of 32. Since this excludes hard cap-hitting episodes that would truncate rollouts and does not condition on success, it isolates a change in strategy rather than an artefact of truncation. Agents therefore respond to tight budgets of turns by batching more work into each interaction, rather than merely stopping earlier.

Table 1: **Mean API calls per step by budget.** Restricted to voluntarily ended episodes.

	$B = 10$	$B = 20$	$B = 32$
`Base`	1.71 ± 0.70	1.42 ± 0.38	1.45 ± 0.41
`Dual`	1.89 ± 0.60	1.68 ± 0.42	1.61 ± 0.38
`Gate`	1.89 ± 0.71	1.60 ± 0.43	1.53 ± 0.39

4.2 Which Rewards Best Teach Budget-Aware Behaviour?

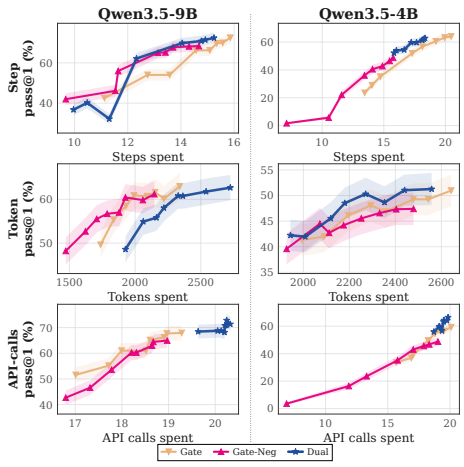


Figure 4: **Mean success against spend.** AppWorld `test-normal`.

`Gate-Neg`, `Gate`, and `Dual` are the strongest contenders. All three achieve high utility across most units and scales in Figure 2. `Linear-Mag` and `Rate` are less reliable, with each performing poorly in different settings. Combining both of their individual mechanisms in `Dual` avoids these failures. On the other hand, `Gate` and `Gate-Neg` preserve simplicity of design, while achieving generally high performance. `Gate-Neg` achieves particularly strong compliance results compared to other methods. `Trunc` is also simple and performs quite reliably, but tends to underperform the three strongest contenders.

Shorter overruns do not imply greater resource efficiency. While `Gate` generally produces shorter overruns in Figure 3(b), this does not translate into uniformly better use of resources for maximising success. Figure 4 compares task success (`pass@1` without cutting off at budget) against average spend when prompted at different budgets. At comparable spend, `Dual` achieves higher success particularly under step budgets and remains competitive under tokens, while the methods follow similar trends under API-call budgets. Thus, `Gate`'s shorter overruns partly reflect more conservative spending rather than a uniformly better success–spend trade-off.

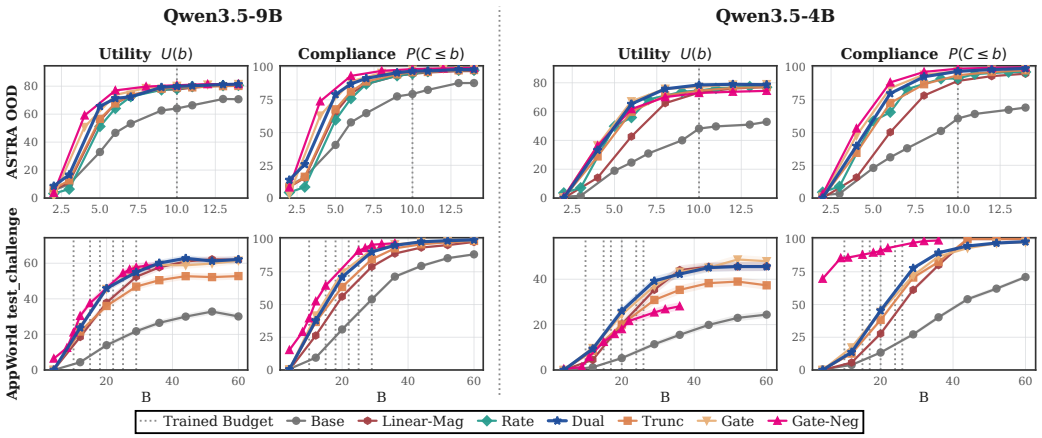


Figure 5: **Out-of-domain budget utility and compliance.** Budget Utility $\mathcal{U}(b)$ and natural compliance $\mathrm{Comp}(b)$ against stated budget B under distribution shift on AppWorld `test-challenge` and out-of-domain on ASTRA for Qwen3.5-9B and 4B. Both evaluations use step budgets.

4.3 Budgeting Generalises Beyond the Training Distribution

Budget behaviour generalises to unseen budgets. The results of Section 4.1 already demonstrate generalisation along the budget axis. A single policy trained on six budgets per unit can achieve consistently high budget utility at budgets that they have never encountered during training, both between training budgets and outside of them. One policy can therefore serve a family of inference budgets without requiring a separate policy for each.

Budget-aware behaviour transfers zero-shot, with the three strongest contenders (`Gate-Neg`, `Gate`, and `Dual`) robust under distribution shift. Figure 5 evaluates AppWorld-trained policies without retraining on AppWorld `test-challenge` and ASTRA, using steps as the budgeting quantity. Budget-aware behaviour persists under both shifts, with budget utility continuing to respond to the stated allowance. On `test-challenge`, budget utility continues rising beyond the largest training budget of 29. Together, these results suggest that the simple cost-conditioned reward functions like `Gate-Neg` and `Gate` can achieve high performance and good generalization.

5 Conclusion

We study whether multi-turn agents can learn to maximise task success under dynamically specified inference budgets, and which training objectives best teach this behaviour. Across budgets over steps, tokens, and tool calls on AppWorld, we find that a single policy can learn conditional resource allocation, adjusting its spend to both the stated budget and task difficulty rather than simply learning to spend less. This behaviour generalises to unseen budgets and transfers zero-shot under distribution shift to AppWorld `test-challenge` and ASTRA, showing that budget management can be learned by the task-performing policy itself rather than delegated to external inference-time machinery. Our comparison of training objectives further shows that explicit reward shaping of the overspend penalties are not required for strong budget awareness. Simply withholding the success reward when the budget is exceeded is highly competitive and produces short overruns despite receiving no signal for their magnitude. Together, these results show that multi-turn agents can learn to ration their own resources across user-specified budgets, while clarifying when additional penalty shaping is useful for learning this behaviour.

AI DISCLOSURE

In this work, we have used generative AI tools to help implement and debug our methods and the code for our experiments. We have not used generative AI tools to generate data sets, develop the theoretical framework, formulate or prove mathematical claims, propose hypotheses, design the experiments, or interpret results, and translation and dataset cleaning are not applicable to this work. Additionally, we used generative AI tools to create and edit the code for our figures, to draft parts of the paper, and to edit the paper for readability. We have reviewed all AI-assisted work. All AI-assisted code was checked and tested by the authors, and all AI-assisted text was read and revised by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 22–31. PMLR, 2017. URL http://proceedings.mlr.press/v70/achiam17a.html.

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning, 2025. URL https://arxiv.org/abs/2503.04697.

Eitan Altman. *Constrained Markov Decision Processes*. Chapman and Hall/CRC, 1999.

Daman Arora and Andrea Zanette. Training language models to reason efficiently. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/579b5b84311e122584dedab1d3b7613f-Abstract-Conference.html.

Nicolas Carrara, Edouard Leurent, Romain Laroche, Tanguy Urvoy, Odalric-Ambrym Maillard, and Olivier Pietquin. Budgeted reinforcement learning in continuous state space. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 9295–9305, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/4fe5149039b52765bde64beb9f674940-Abstract.html.

Yinlam Chow. *Risk-sensitive and data-driven sequential decision making*. Stanford University, 2017.

Mehul Damani, Idan Shenfeld, Andi Peng, Andreea Bobu, and Jacob Andreas. Learning how hard to think: Input-adaptive allocation of LM computation. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=6qUUgw9bAZ.

DeepSeek-AI. DeepSeek-V4.1-Flash: Pushing the limits of KV cache compression. https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash/blob/main/DeepSeek_V41_Tech_Report.pdf, 2026. Accessed: 2026-09-25.

Peter Geibel and Fritz Wysotzki. Risk-sensitive reinforcement learning applied to control under constraints. *Journal of Artificial Intelligence Research*, 24:81–108, 2005.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 24842–24855, Vienna, Austria, 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.1274. URL https://aclanthology.org/2025.findings-acl.1274/.

William B Haskell and Rahul Jain. A convex analytic approach to risk-aware markov decision processes. *SIAM Journal on Control and Optimization*, 53(3):1569–1598, 2015.

Kimi Team. Kimi k3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Yushu Li, Wenlong Deng, Jiajin Li, and Xiaoxiao Li. Spend less, reason better: Budget-aware value tree search for llm agents, 2026. URL https://arxiv.org/abs/2603.12634.

Chang Liu, Yiran Zhao, Lawrence Liu, Yaoqi Ye, Csaba Szepesvári, and Lin F Yang. Laconic: Length-aware constrained reinforcement learning for llm, 2026. URL https://arxiv.org/abs/2602.14468.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Dler: Doing length penalty right - incentivizing more intelligence per token via reinforcement learning, 2025a. URL https://arxiv.org/abs/2510.15110.

Tengxiao Liu, Zifeng Wang, Jin Miao, I Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, et al. Budget-aware tool-use enables effective agent scaling, 2025b. URL https://arxiv.org/abs/2511.17006.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025c. URL https://arxiv.org/abs/2503.20783.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. URL https://qwen.ai/blog?id=qwen3.5.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Adam Stooke, Joshua Achiam, and Pieter Abbeel. Responsive safety in reinforcement learning by PID lagrangian methods. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 9133–9143. PMLR, 2020. URL http://proceedings.mlr.press/v119/stooke20a.html.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.850. URL https://aclanthology.org/2024.acl-long.850/.

Hao Wen, Xinrui Wu, Yi Sun, Feifei Zhang, Liye Chen, Jie Wang, Yunxin Liu, Yunhao Liu, Ya-Qin Zhang, and Yuanchun Li. Budgetthinker: Empowering budget-aware llm reasoning with control tokens, 2025. URL https://arxiv.org/abs/2508.17196.

Zidi Xiu, David Q Sun, Kevin Cheng, Maitrik Patel, Yizhe Zhang, Jiarui Lu, Omar Attia, Raviteja Vemulapalli, Oncel Tuzel, Meng Cao, et al. Astra-bench: Evaluating tool-use agent reasoning and action planning with personal user context, 2026. URL https://arxiv.org/abs/2603.01357.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: an open-source LLM reinforcement learning system at scale. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla, editors, *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/a4277440d50f1f15d2cb4c14f7e0c0d2-Abstract-Conference.html.

Caiqi Zhang, Menglin Xia, Xuchao Zhang, Daniel Madrigal, Ankur Mallick, Samuel Kessler, Victor Ruehle, and Saravan Rajmohan. Budget-aware agentic routing via boundary-guided training, 2026. URL https://arxiv.org/abs/2602.21227.

APPENDIX CONTENTS

A Notation **13**

B Budget-Conditioned Rewards in Full **13**

C Background **14**

C.1 DAPO's Soft-Overlong Penalty 15

D Theoretical Results **15**

D.1 Background 16

D.2 Setup 16

D.3 Rearrangement of the Lagrangian 16

D.4 Gradients of the Lagrangian 16

D.5 Complementary Roles of the Two Penalties 17

E Optimising the `Dual` Objective **18**

E.1 Group-Relative Policy Optimisation 18

E.2 Policy and Dual Updates 19

E.3 Training Algorithm 19

F Experimental Setup **21**

F.1 AppWorld 21

F.2 ASTRA 21

F.3 Cost Maps and Budgeted Quantities 21

F.4 Budget Conditioning and Prompt Templates 22

F.5 Models and Dr. GRPO Settings 26

F.6 Budget and Constraint Settings 26

F.7 Compute 26

G Limitations **27**

A NOTATION

Table 2 collects the symbols used throughout the paper, grouped by where they are introduced.

Task and environment (Sections 3.1 and 3.2)	
$\mathcal{S}, \mathcal{A}, \Omega$	State, action, and observation spaces
$\mathcal{T}, \mathcal{O}$	Transition and observation kernels
$\mathcal{M}, \tilde{\mathcal{M}}$	Task POMDP and budget-augmented POMDP
γ, μ_0	Discount factor and initial state distribution
$x, p_{\mathcal{X}}, \mathcal{X}$	Task, task distribution, and task set
$h_t, \mathcal{H}$	Observation history at step t, and set of all histories
π, Π	Policy, and set of policies
κ	Cost map pricing each action
$a_\perp$	`complete_task` action that ends an episode
$T_{\max}, \tau$	Truncation horizon, and stopping time
ζ	Completed trajectory
$R(\zeta)$	Binary terminal reward
Budgets and cost (Section 3.2 and Appendix B)	
B_0, b_t	Episode budget, and remaining budget at step t
$\mathcal{B}, M$	Finite set of training budgets, and its size $M = \lvert\mathcal{B}\rvert$
p_B	Budget prior over $\mathcal{B}$
$C(\zeta, B_0)$	Normalised trajectory cost
δ	Permitted overspend rate
Objectives and rewards (Section 3.3 and Appendix D)	
P_{DAPO}	Soft-overlong penalty, adapted from DAPO
α	Coefficient of P_{DAPO}, fixed at $1/2$
λ_b	Lagrange multiplier at budget b
Λ_b	Effective price at budget b, $\Lambda_b = \lambda_b / p_B(b)$
$\mathcal{L}(\pi, \{\lambda_b\})$	Lagrangian of the constrained objective
$\tilde{R}(\zeta, B_0)$	Shaped return
$\tilde{R}_{\mathrm{gate}}, \tilde{R}_{\mathrm{trunc}}$	Gated and truncated within-budget success rewards
$\tilde{R}_{\mathrm{DAPO}}, \tilde{R}_{\mathrm{rate}}$	Rewards with only the overspend magnitude or only the overspend rate penalty
$J(\pi), \tilde{J}(\pi)$	Success objective, and its shaped counterpart
$g_b(\pi)$	Constraint functional at budget b
$V^\star$	Optimal value of the constrained objective
Policy optimisation (Appendices E.1 and E.2)	
θ, π_θ	Policy parameters, and the policy they induce
K, G	Number of task-budget groups per iteration, and rollouts per group
$A^{(g,i)}$	Dr. GRPO advantage of rollout i in group g
σ	Group reward standard deviation (GRPO normaliser dropped by Dr. GRPO)
ϵ	PPO clip range
$r_{t,l}^{(g,i)}(\theta)$	Token-level importance ratio
β	Dual step size
$\hat{p}_b, \mathcal{I}_b$	Batch overspend rate at budget b, and the rollouts it is estimated from
Evaluation (Section 4)	
$\mathcal{U}(b), \mathrm{Comp}(b)$	Budget utility and compliance at budget b
$\mathcal{X}_{\mathrm{test}}$	Held-out task set used for evaluation
AUBUC, AUBCC	Area under the budget-utility and compliance curves

Table 2: **Notation.** Symbols used throughout the paper.

B BUDGET-CONDITIONED REWARDS IN FULL

This appendix defines every reward of Figure 1 and derives the per-budget price of `Rate` and `Dual`. All rewards are written in the normalised cost $C = C(\zeta, B_0)$ of Equation (5), so that the budget sits at $C = 1$.

Baselines from prior work. `Trunc` enforces the budget in the environment. It halts the episode before executing any action that would overdraw the budget, at $\tau_{B_0} := \min\big(\tau,\ \inf\{t \geq 1 : b_t - \kappa(a_t) < 0\} - 1\big)$, and scores the truncated trajectory by the task reward alone, $\tilde{R}_{\mathrm{trunc}}(\zeta, B_0) := R(\zeta_{\leq \tau_{B_0}})$. Every training trajectory then complies by construction, but the policy never observes $b_t < 0$, so it is never trained to stop on its own. `Gate-Neg` (Kimi Team, 2026) extends `Gate` with a fixed penalty for overspending, $R\,\mathbf{1}[C \leq 1] - \mathbf{1}[C > 1]$. `Linear-Cost` (DeepSeek-AI, 2026) penalises cost linearly from the first action, $R - \min(1, C/2)$, so it applies pressure before the budget is reached. `Exp-Mag` decays a success exponentially in the overspend, $R\,e^{-2(C-1)_+}$, and leaves every failure at 0.

Magnitude shaping. We adapt the DAPO shaping Equation (12) to the normalised cost so that one form covers every budget in every cost unit,

$$P_{\mathrm{DAPO}}(\zeta, B_0) := \alpha\,\mathrm{clip}\big(\,C(\zeta, B_0) - 1,\ 0,\ 1\,\big), \tag{8}$$

where $\alpha := 1/2$ is the weight DAPO's reward scale fixes rather than a tuned hyperparameter (derived in Appendix C.1). Its reward $\tilde{R}_{\mathrm{DAPO}} := R - P_{\mathrm{DAPO}}$ extends DAPO from one cap fixed before training to a budget sampled each episode. Its fixed coefficient penalises how far a policy overspends but not how often.

A per-budget constrained objective. Penalising cost fixes a trade-off between success and spending but does not control how often the policy overspends. We therefore maximise success subject to a constraint on the overspend rate. Following the chance-constrained framing of Section 2, we bound the probability of overspending rather than the expected cost, and impose one constraint per budget rather than pooling them, giving the constrained objective

$$\max_{\pi \in \Pi}\ \mathbb{E}_{\substack{x \sim p_{\mathcal{X}},\ B_0 \sim p_B \\ \zeta \sim \pi(\cdot|x, B_0)}}\big[\,R(\zeta)\,\big] \quad \text{s.t.} \quad \mathbb{P}_{\substack{x \sim p_{\mathcal{X}} \\ \zeta \sim \pi(\cdot|x,b)}}\big(\,C(\zeta, b) > 1\,\big) \leq \delta, \quad \forall\, b \in \mathcal{B}, \tag{9}$$

where $\delta \in (0, 1)$ is the permitted overspend rate. The objective averages over tasks and budgets together, whereas each constraint conditions on the budget and averages over tasks alone, so it is the overspend rate a user would measure at budget b across the tasks they send. Slack may therefore move between tasks but never between budgets. Subtracting P_{DAPO} from the objective of Equation (9), with the constraints unchanged, gives our `Dual` *training objective*,

$$\max_{\pi \in \Pi}\ \mathbb{E}_{\substack{x \sim p_{\mathcal{X}},\ B_0 \sim p_B \\ \zeta \sim \pi(\cdot|x, B_0)}}\big[\,R(\zeta) - P_{\mathrm{DAPO}}(\zeta, B_0)\,\big] \quad \text{s.t.} \quad \mathbb{P}_{\substack{x \sim p_{\mathcal{X}} \\ \zeta \sim \pi(\cdot|x,b)}}\big(\,C(\zeta, b) > 1\,\big) \leq \delta, \quad \forall\, b \in \mathcal{B}. \tag{10}$$

Lagrangian relaxation. Introducing one non-negative multiplier $\lambda_b \in \mathbb{R}_{\geq 0}$ per budget $b \in \mathcal{B}$ and folding the resulting sum of per-budget penalties into a single expectation (see Appendix D.3 for more detail) relaxes Equation (10) to the Lagrangian objective

$$\mathcal{L}\big(\pi, \{\lambda_b\}\big) = \mathbb{E}_{\substack{x \sim p_{\mathcal{X}},\ B_0 \sim p_B \\ \zeta \sim \pi(\cdot|x, B_0)}}\big[\,R(\zeta) - P_{\mathrm{DAPO}}(\zeta, B_0) - \Lambda_{B_0}\,\mathbf{1}[C(\zeta, B_0) > 1]\,\big] + \delta \sum_{b \in \mathcal{B}} \lambda_b, \tag{11}$$

where $\Lambda_b := \lambda_b / p_B(b) \in \mathbb{R}_{\geq 0}$ is the *effective price* at budget b, its weight $1/p_B(b)$ correcting for how often that budget is drawn. The trailing term is constant in π, leaving the per-trajectory integrand, which is the shaped return of `Dual` in Figure 1. `Dual` trains on this return together with the dual update on $\{\lambda_b\}$ that calibrates it (Appendix E), and `Rate` drops the P_{DAPO} term, relaxing Equation (9) to the return of `Rate` in Figure 1. Each multiplier prices only the overspend event at its own budget, so the policy can still use its whole budget. The two penalties are complementary. The dual sets *how often* the policy may overspend, calibrated per budget to δ (Appendix D.4), while P_{DAPO} orders overspending trajectories by *how far* they ran over, which a rate term cannot do. Appendix D.5 formalises this.

C Background

We first give DAPO's original soft-overlong penalty and show how we adapt it from a single fixed token cap to a family of budgets (Appendix C.1).

C.1 DAPO's Soft-Overlong Penalty

DAPO (Yu et al., 2025) adds a soft-overlong penalty to the task reward that penalises response length as it approaches and exceeds a fixed token cap. Given a maximum response length $L_{\max} \in \mathbb{Z}_{>0}$ and a soft window $L_{\mathrm{s}} \in \mathbb{Z}_{>0}$ with $L_{\mathrm{s}} \leq L_{\max}$, it shapes the reward according to the response length $L(o) \in \mathbb{Z}_{\geq 0}$ as

$$R_{\mathrm{len}}(o) = \begin{cases} 0, & L(o) \leq L_{\max} - L_{\mathrm{s}}, \\ \dfrac{(L_{\max} - L_{\mathrm{s}}) - L(o)}{L_{\mathrm{s}}}, & L_{\max} - L_{\mathrm{s}} < L(o) \leq L_{\max}, \\ -1, & L(o) > L_{\max}. \end{cases} \tag{12}$$

The penalty is zero up to the soft threshold $L_{\max} - L_{\mathrm{s}}$, decreases linearly to -1 over a window of width L_{s}, and remains at -1 beyond $L_{\max}$. It therefore penalises the magnitude of overspend smoothly rather than imposing a hard cut-off.

This construction was designed for a single token-length cap fixed before training. Our setting instead requires one policy to work across a family of budgets B_0, with the budget varying between episodes, and with budgeted quantities other than tokens, such as steps or tool calls. We adapt DAPO's penalty to this setting by expressing spend relative to the budget of the current episode.

Concretely, set $L_{\mathrm{s}} = B_0$ and $L_{\max} = 2B_0$ in Equation (12). The soft threshold $L_{\max} - L_{\mathrm{s}}$ is then exactly B_0, and the middle branch becomes

$$\frac{(2B_0 - B_0) - L(o)}{B_0} = \frac{B_0 - L(o)}{B_0} = -\left(\frac{L(o)}{B_0} - 1\right). \tag{13}$$

Thus, rather than applying the ramp to the raw token count $L(o)$ relative to one fixed cap, we apply it to the normalised trajectory cost $C(\zeta, B_0)$ of Equation (5). Reversing the sign so that it is written as a positive penalty to be subtracted from the return gives the same ramp in normalised-cost units, $\mathrm{clip}(C(\zeta, B_0) - 1,\, 0,\, 1)$. The cost map κ determines the quantity being budgeted, while normalisation by B_0 makes the soft threshold $C(\zeta, B_0) = 1$ for every budget. The same DAPO ramp therefore applies to any budget and to any cost unit.

It remains to match the scale of DAPO's original reward. DAPO pairs a task reward $R_{\mathrm{DAPO}} \in \{-1, +1\}$ with a penalty $R_{\mathrm{len}} \in [-1, 0]$, whereas our terminal reward $R(\zeta) \in \{0, 1\}$. The map $x \mapsto (x+1)/2$ takes DAPO's task-reward scale to ours. Applying the same map to DAPO's shaped return gives

$$\frac{R_{\mathrm{DAPO}} + R_{\mathrm{len}} + 1}{2} = \underbrace{\frac{R_{\mathrm{DAPO}} + 1}{2}}_{R(\zeta)} + \tfrac{1}{2} R_{\mathrm{len}}, \tag{14}$$

so preserving DAPO's relative weighting between task success and the soft-overlong penalty introduces a factor of $\frac{1}{2}$. Combining this rescaling with Appendix C.1 gives our penalty

$$P_{\mathrm{DAPO}}(\zeta, B_0) = \tfrac{1}{2}\,\mathrm{clip}(C(\zeta, B_0) - 1,\, 0,\, 1)\,, \tag{15}$$

which is Equation (8) with $\alpha = \frac{1}{2}$. We therefore set $\alpha = 0.5$ throughout (Appendix F.6). In raw spend, the penalty has slope $1/(2B_0)$ within the linear overspending region, so a fixed absolute overspend carries a larger penalty under a tighter budget than under a looser budget.

The rescaling itself does not otherwise change the Dr. GRPO update. The Dr. GRPO advantage centres each trajectory's return within its group (Appendix E.2), so an affine transformation $\tilde{R} \mapsto a\tilde{R} + c$ with $a > 0$ removes the additive constant under centring and multiplies every advantage by a. It therefore preserves their signs and relative magnitudes and only rescales the resulting gradient, which can be absorbed into the learning rate.

D Theoretical Results

This appendix collects the results of Section 3. Appendix D.2 fixes the policy class and constraint functionals, Appendix D.3 verifies the rearrangement of the Lagrangian Equation (11), Appendix D.4 derives the policy and multiplier updates of Appendix E from it, and Appendix D.5 shows why `Dual` needs both of its penalties.

D.1 Background

Constrained MDPs (cMDPs) typically bound expected cumulative cost for a single fixed threshold through Lagrangian relaxation (Altman, 1999; Achiam et al., 2017; Stooke et al., 2020). Budgeted MDPs (Carrara et al., 2019) extend cMDPs to families of budgets by conditioning value functions on the remaining allowance, but still constrain *expected* discounted cost return via fitted Q-iteration. This does not directly express budget compliance at the level of individual requests, where an expected-cost constraint can hide frequent individual violations. Risk-aware and chance-constrained formulations instead bound the probability of undesirable events (Geibel and Wysotzki, 2005; Haskell and Jain, 2015), and Chow (2017) studies trajectory-level probabilistic constraints under Lagrangian relaxation.

D.2 Setup

Throughout we work with the trajectory law induced by a policy π on the budget-augmented POMDP $\tilde{\mathcal{M}}$ of Section 3.2. We write $\mathbb{P}_\pi$ for that law, i.e. $\mathbb{P}_\pi(\cdot) = \mathbb{P}_{x\sim p_{\mathcal{X}},\, B_0\sim p_B,\, \zeta\sim\pi(\cdot|x,B_0)}(\cdot)$ and likewise $\mathbb{E}_\pi$ for expectation under it, matching the sampling made explicit in Equations (9) and (11). The task draw is folded into the law instead of being tracked separately, since every functional below averages over it, and conditioning on a fixed budget, written $\mathbb{P}_\pi(\cdot \mid B_0 = b)$ below, is the task-marginal law $\pi(\cdot \mid b)$ of Section 3.2. We define the objective and per-budget constraint functionals

$$J(\pi) := \mathbb{E}_\pi[\, R(\zeta)\,], \qquad g_b(\pi) := \mathbb{P}_\pi(\, C(\zeta, B_0) > 1 \,|\, B_0 = b\,) - \delta, \qquad b \in \mathcal{B}, \tag{16}$$

so that the constrained objective Equation (9) reads $\max_{\pi\in\Pi} J(\pi)$ subject to $g_b(\pi) \le 0$ for every $b \in \mathcal{B}$. The conditioning in Equation (16) is well-defined because p_B has full support on the finite set $\mathcal{B}$. Equation (10) keeps these constraints and shapes the objective,

$$\tilde{J}(\pi) := \mathbb{E}_\pi[\, R(\zeta) - P_{\mathrm{DAPO}}(\zeta, B_0)\,], \tag{17}$$

so that it reads $\max_{\pi\in\Pi} \tilde{J}(\pi)$ subject to the same $g_b(\pi) \le 0$, with $\mathcal{L}$ of Equation (11) its Lagrangian. Results below are stated for $\tilde{J}$, and setting $\alpha = 0$ recovers J and Equation (9).

D.3 Rearrangement of the Lagrangian

Relaxing Equation (10) prices each budget with its own multiplier, through a penalty $\sum_{b\in\mathcal{B}} \lambda_b\big(\mathbb{P}_\pi(C(\zeta, B_0) > 1 \mid B_0 = b) - \delta\big)$. Folding that sum into a single expectation produces the compact Lagrangian Equation (11) and its per-trajectory integrand, the shaped return of `Dual` (Figure 1). Writing each overspend probability as an expectation and splitting the outer expectation over the prior $B_0 \sim p_B$,

$$\begin{aligned}
\sum_{b\in\mathcal{B}} \lambda_b\, \mathbb{P}_\pi(C(\zeta, B_0) > 1 \,|\, B_0 = b) &= \sum_{b\in\mathcal{B}} \lambda_b\, \mathbb{E}_\pi[\, \mathbf{1}[C(\zeta, B_0) > 1] \,|\, B_0 = b\,] \\
&= \sum_{b\in\mathcal{B}} p_B(b)\, \frac{\lambda_b}{p_B(b)}\, \mathbb{E}_\pi[\, \mathbf{1}[C(\zeta, B_0) > 1] \,|\, B_0 = b\,] \\
&= \mathbb{E}_\pi\left[\, \frac{\lambda_{B_0}}{p_B(B_0)}\, \mathbf{1}[C(\zeta, B_0) > 1]\,\right],
\end{aligned} \tag{18}$$

where the second line divides and multiplies by $p_B(b) > 0$ and the third recognises the resulting sum as an expectation over $B_0 \sim p_B$. The importance weight $1/p_B(B_0)$ therefore corrects for the frequency with which each budget is sampled, so that a budget contributes to the penalty in proportion to λ_b alone and not to how often it is drawn. The remaining term $\delta \sum_b \lambda_b$ is constant in π and plays no part in the policy update. Under the uniform prior used in our experiments, $1/p_B(b) = |\mathcal{B}|$ for every budget, so the effective price appearing in the shaped return is $\Lambda_b = |\mathcal{B}|\, \lambda_b$.

D.4 Gradients of the Lagrangian

This section derives the policy and multiplier updates of Appendix E from the Lagrangian Equation (11), whose per-trajectory integrand is the shaped return $\tilde{R}$ of `Dual` (Figure 1) and whose trailing term $\delta \sum_{b\in\mathcal{B}} \lambda_b$ is constant in π.

Policy gradient. Fix $\{\lambda_b\}$ and parameterise the policy by π_θ as in Equation (3). In the budget-augmented POMDP $\tilde{\mathcal{M}}$ of Section 3.2, neither the kernels $\tilde{\mathcal{T}}, \tilde{\mathcal{O}}$ nor the budget recursion $b_{t+1} = b_t - \kappa(a_t)$ depends on θ, so their log-derivatives vanish and the trajectory score reduces to a sum over the policy's own factors,

$$s_\theta(\zeta, x, B_0) := \nabla_\theta \log p_\theta(\zeta \mid x, B_0) = \sum_{t \le \tau} \nabla_\theta \log \pi_\theta(a_t \mid x, h_t, b_t, B_0)\,. \tag{19}$$

Since $\delta \sum_b \lambda_b$ is constant in π, the score-function gradient of the Lagrangian is

$$\nabla_\theta \mathcal{L}\big(\pi_\theta, \{\lambda_b\}\big) = \mathbb{E}_{x \sim p_{\mathcal{X}},\, B_0 \sim p_B,\, \zeta \sim \pi_\theta(\cdot \mid x, B_0)} \Big[\tilde{R}(\zeta, B_0)\, s_\theta(\zeta, x, B_0) \Big]\,. \tag{20}$$

Each action a_t is itself a token sequence, so each factor of Equation (19) expands into per-token log-probabilities and s_θ sums over all generated tokens. The return is terminal and undiscounted, so Equation (20) weights every token of a trajectory by the same scalar $\tilde{R}$, with no length normaliser.

As usual, we reduce variance by centring returns within each group. Write $\tilde{R}^{(g,i)} := \tilde{R}\big(\zeta^{(g,i)}, B_0^{(g)}\big)$, $s_\theta^{(g,i)} := s_\theta\big(\zeta^{(g,i)}, x^{(g)}, B_0^{(g)}\big)$, and $\bar{\tilde{R}}^{(g)} := G^{-1} \sum_{j=1}^{G} \tilde{R}^{(g,j)}$. The group's trajectories are i.i.d. *given* its task and budget, which is why the group mean is a valid baseline only at fixed $(x^{(g)}, B_0^{(g)})$. Conditionally on that pair, $\mathbb{E}_{\zeta \sim \pi_\theta(\cdot \mid x, B_0)}[s_\theta(\zeta, x, B_0)] = 0$, so the cross terms vanish and only the trajectory's own contribution to $\bar{\tilde{R}}^{(g)}$ survives. Averaging the conditional identity over the draw of the pair itself then scales the full gradient by $1 - 1/G$,

$$\mathbb{E}_{\substack{(x^{(g)}, B_0^{(g)}) \sim p_{\mathcal{X}} \otimes p_B \\ \zeta^{(g,1)}, \ldots, \zeta^{(g,G)} \sim \pi_\theta(\cdot \mid x^{(g)}, B_0^{(g)})}} \Big[\Big(\tilde{R}^{(g,i)} - \bar{\tilde{R}}^{(g)} \Big)\, s_\theta^{(g,i)} \Big] = \left(1 - \frac{1}{G} \right) \nabla_\theta \mathcal{L}\big(\pi_\theta, \{\lambda_b\}\big), \tag{21}$$

with the constant absorbed into the learning rate. We omit GRPO's standard-deviation and length normalisers of Equation (25), which do not preserve this gradient (Appendix E.1).

In practice we replace Equation (20) by the clipped Dr. GRPO surrogate Equation (28), recovering the advantage Equation (27). It agrees with the policy gradient to first order at $\theta = \theta_{\text{old}}$, where every ratio $r_t^{(g,i)}(\theta)$ equals one, while clipping controls the off-policy change over repeated optimisation steps on one batch of rollouts (Appendix E.1).

Multiplier gradient. Now fix π and write $p_b := \mathbb{P}_{\zeta \sim \pi(\cdot \mid b)}(C(\zeta, b) > 1) = g_b(\pi) + \delta$ for the overspend rate at budget b, with g_b the constraint functional of Equation (16). Differentiating Equation (11) in λ_b, with the penalty written per budget as in Equation (18), gives

$$\frac{\partial \mathcal{L}}{\partial \lambda_b} = -g_b(\pi) = -(p_b - \delta). \tag{22}$$

Since the multipliers minimise $\mathcal{L}$ subject to $\lambda_b \ge 0$, projected gradient descent in λ_b gives the projected dual update of Algorithm 1,

$$\lambda_b \leftarrow \left[\lambda_b - \beta \frac{\partial \mathcal{L}}{\partial \lambda_b} \right]_+ = \big[\, \lambda_b + \beta(p_b - \delta) \,\big]_+. \tag{23}$$

Replacing p_b by the batch estimate $\hat{p}_b$ gives Equation (29). Each multiplier therefore increases when $p_b > \delta$ and decreases when $p_b < \delta$, subject to projection at zero. The batch estimate $\hat{p}_b$ is unbiased for p_b under the current policy. We use it without smoothing across updates to keep the estimate on-policy. By Equation (22) the gradient at budget b does not involve $\lambda_{b'}$ for $b' \neq b$, so all multipliers update in parallel from one batch, as in Algorithm 1.

At a fixed point in the interior of the range Equation (30) permits, $\lambda_b > 0$ implies $p_b = \delta$, while $\lambda_b = 0$ implies $p_b \le \delta$. Each price therefore settles where its budget overspends at rate δ whenever that budget binds.

D.5 Complementary Roles of the Two Penalties

Both penalties in the `Dual` return Equation (11) activate on the same event $\{C(\zeta, B_0) > 1\}$ but act on different quantities. As a function of the normalised cost $c = C(\zeta, b)$ at a fixed task outcome, the

rate term $\Lambda_b \mathbf{1}[c > 1]$ is constant on $\{c > 1\}$, whereas P_{DAPO} of Equation (8) has slope α on $(1, 2)$. The rate term alone is therefore indifferent between a marginal and a gross overspend. Conversely, $\alpha = \frac{1}{2}$ is fixed and shared across budgets, so the overspend rate it induces at any budget is not controlled, whereas an interior fixed point of the dual update gives $p_b = \delta$ (Appendix D.4).

What reaches the policy is the group-centred advantage Equation (27). Fix a group g at budget b and drop the index g, writing $R^{(i)}$ for the task reward of rollout i, $u^{(i)} := \mathbf{1}[C(\zeta^{(i)}, b) > 1]$ for its overspend indicator, $P^{(i)} := P_{\mathrm{DAPO}}(\zeta^{(i)}, b)$ for its magnitude penalty and $\hat{p} := G^{-1} \sum_j u^{(j)}$ for the overspend fraction of the group. Centring each term of the return separately gives

$$A^{(i)} = \underbrace{\big(R^{(i)} - \bar{R}\big)}_{\text{task outcome}} - \underbrace{\big(P^{(i)} - \bar{P}\big)}_{\text{magnitude}} - \underbrace{\Lambda_b \big(u^{(i)} - \hat{p}\big)}_{\text{rate}}, \tag{24}$$

with bars denoting group means. Which terms survive depends on how the group splits on the budget. If no rollout overspends, both penalty terms vanish and the group reduces to ordinary task learning whatever Λ_b has reached. If every rollout overspends, the rate term cancels however large Λ_b is, and only the magnitude term separates the rollouts. Without it, a group that also shares its task outcome has zero advantage. This is the common case at a tight budget early in training, when every rollout overspends and none succeeds, so rate-only shaping gives no signal where budget pressure is greatest. If compliance is mixed, the rate term separates compliant from overspending rollouts by exactly Λ_b however far the overspenders ran over, while the magnitude term orders the overspending rollouts by at most α. The dual therefore sets the price of overspending at all, calibrated per budget to δ, and P_{DAPO} orders overspending trajectories by how far they ran over. Neither is recoverable from the other by rescaling, so `Dual` keeps both.

E Optimising the Dual Objective

We give the policy and dual updates used to optimise the `Dual` objective. We first state the Dr. GRPO objective used for the policy update (Appendix E.1), then give the policy and dual updates for the `Dual` objective (Appendix E.2), and present them together in a full training algorithm (Appendix E.3). Appendix D.4 derives both updates from the Lagrangian.

E.1 Group-Relative Policy Optimisation

GRPO (Shao et al., 2024) is a critic-free variant of PPO (Schulman et al., 2017). For each prompt, it samples a group of $G \in \mathbb{Z}_{>0}$ responses with rewards $\{r^{(i)}\}_{i=1}^{G} \subset \mathbb{R}$ and uses the group mean as the baseline, giving the advantage

$$A^{(i)} = \frac{1}{\sigma} \left(r^{(i)} - \frac{1}{G} \sum_{j=1}^{G} r^{(j)} \right), \tag{25}$$

where σ is the standard deviation of the group rewards. GRPO applies this advantage to each token through a clipped PPO surrogate, with the token updates normalised by response length.

Liu et al. (2025c) identify biases from both the reward standard-deviation and response-length normalisers. Normalising by σ gives greater weight to groups with lower reward variance. Normalising by response length gives larger per-token updates to shorter positive-advantage responses but smaller per-token updates to longer negative-advantage responses, encouraging failed responses to grow longer. Dr. GRPO removes both normalisers, using

$$A^{(i)} = r^{(i)} - \frac{1}{G} \sum_{j=1}^{G} r^{(j)} \tag{26}$$

and summing the surrogate over tokens without dividing by response length.

We use Dr. GRPO throughout, grouping trajectories by task and budget so that the within-group baseline compares rollouts subject to the same budget. Appendix E.2 gives the resulting update for our shaped return.

E.2 Policy and Dual Updates

Policy update. At each iteration, we draw $K \in \mathbb{Z}_{>0}$ task and budget pairs $(x^{(g)}, B_0^{(g)}) \sim p_{\mathcal{X}} \otimes p_B$, $g = 1, \ldots, K$, and sample $G \in \mathbb{Z}_{>0}$ trajectories $\zeta^{(g,i)} \sim \pi_\theta(\cdot \mid x^{(g)}, B_0^{(g)})$, $i = 1, \ldots, G$, under each. Writing $\tilde{R}^{(g,i)}$ for the shaped return of `Dual` shown in Figure 1, we centre the return within each group to obtain

$$A^{(g,i)} = \tilde{R}^{(g,i)} - \frac{1}{G} \sum_{j=1}^{G} \tilde{R}^{(g,j)}. \tag{27}$$

A group is thus G rollouts of one task at one budget, sharing an overspend threshold and multiplier $\lambda_{B_0^{(g)}}$. Centring within groups therefore contrasts trajectories that faced identical budgets. This is Equation (26) with $r^{(i)}$ read as $\tilde{R}^{(g,i)}$, and Equation (28) below is the corresponding surrogate.

Collecting everything the policy conditions on at a step into $z_t^{(g,i)} := \big(x^{(g)}, h_t^{(g,i)}, b_t^{(g,i)}, B_0^{(g)}\big)$ and writing the action as its token sequence $a_t^{(g,i)} = \big(a_{t,1}^{(g,i)}, \ldots, a_{t,L_t^{(g,i)}}^{(g,i)}\big)$, so that the importance ratio is taken per token as $r_{t,l}^{(g,i)}(\theta) = \pi_\theta\big(a_{t,l}^{(g,i)} \mid z_t^{(g,i)}, a_{t,<l}^{(g,i)}\big) / \pi_{\theta_{\text{old}}}\big(a_{t,l}^{(g,i)} \mid z_t^{(g,i)}, a_{t,<l}^{(g,i)}\big)$, and with clip range $\epsilon \in (0, 1)$, we ascend the Dr. GRPO surrogate

$$\mathcal{J}(\theta) = \frac{1}{KG} \sum_{g=1}^{K} \sum_{i=1}^{G} \sum_{t \le \tau^{(g,i)}} \sum_{l \le L_t^{(g,i)}} \min\Big(r_{t,l}^{(g,i)}(\theta) A^{(g,i)}, \operatorname{clip}\Big(r_{t,l}^{(g,i)}(\theta), 1-\epsilon, 1+\epsilon \Big) A^{(g,i)} \Big). \tag{28}$$

As in Dr. GRPO, the trajectory-level advantage is applied to every token without a length normaliser, which Appendix D.4 derives from the policy gradient of Equation (11).

Dual update. Each λ_b rises while the policy overspends at budget b more often than δ and falls otherwise, so a frequently violated budget grows more expensive, while a comfortably met budget becomes cheaper. We estimate the overspend rate from the batch rollouts drawn at budget b, pooled over their tasks, which is the marginal rate Equation (9) bounds. Writing $C^{(g,i)} := C(\zeta^{(g,i)}, B_0^{(g)})$, $\mathcal{I}_b := \{(g,i) : B_0^{(g)} = b\}$ and $\beta \in \mathbb{R}_{>0}$ for the dual step size,

$$\hat{p}_b = \frac{1}{|\mathcal{I}_b|} \sum_{(g,i) \in \mathcal{I}_b} \mathbf{1}\Big[C^{(g,i)} > 1 \Big], \qquad \lambda_b \leftarrow \big[\lambda_b + \beta(\hat{p}_b - \delta) \big]_+ . \tag{29}$$

This is projected gradient descent on Equation (11) in λ_b, whose gradient there is $\delta - p_b$ at the true rate $p_b \in [0, 1]$ (derived in Appendix D.4). The raw batch estimate, unsmoothed across updates, keeps the price on-policy.

Nothing in Equation (29) bounds λ_b on its own. If the target overspend rate δ is unattainable at a budget, λ_b can therefore continue to grow. We cap the effective price at $\frac{1}{2}$, projecting each multiplier onto $\big[0,\ \frac{1}{2} p_B(b)\big]$,

$$\lambda_b \leftarrow \min\Big\{ \big[\lambda_b + \beta(\hat{p}_b - \delta) \big]_+,\ \tfrac{1}{2} p_B(b) \Big\}. \tag{30}$$

Equivalently, the effective price satisfies $\Lambda_b \le \frac{1}{2}$. Since the DAPO penalty is also bounded by $\frac{1}{2}$, this is the largest dual price for which an overspending success is never valued below a compliant failure. We cap λ_b itself rather than only clipping the price in the shaped return, so that the price can fall immediately once the overspend rate drops below δ.

E.3 Training Algorithm

Algorithm 1 gives the full training procedure for `Dual`. Rollouts are budget-conditioned under a budget shared within each group, scored by the shaped return of Figure 1 with its soft-overlong penalty Equation (8), and used for the Dr. GRPO update Equation (28) and for the projected dual update Equation (29) on the raw within-batch overspend rate. Note that this is simply standard GRPO with a shaped reward, with the small additions of scalar updates of λ_b given by a deterministic rule that does not require extra rollouts or additional gradient calculations.

Every `Dual` variant changes only the observations and the reward an environment returns, plus, for the Lagrangian rate variants, one arithmetic update to the multipliers. Nothing else in a standard RL pipeline changes. Each environment draws its budget $B_0 \sim p_B$ at the start of a rollout collection phase, which under a uniform prior needs no coordination beyond a shared seed and gives every environment in a task group the same budget. It writes B_0 and the remaining budget b_t into the observations the agent sees (Appendix F.4), and at the end of generation returns the cost-conditioned reward $\tilde{R}(\zeta, B_0)$ in place of a binary reward. This is a single scalar with no gradients attached, so it enters the pipeline as any reward would.

For the Lagrangian rate variants, after each phase we compare the realised overspend rate at each budget, pooled over the tasks and rollouts drawn there, against the permitted rate δ and move λ_b accordingly. That update is closed-form on $|\mathcal{B}|$ scalars and needs no optimiser. But it needs an object acting outside individual environments to have access to all rollouts, and needs to communicate back the updated Lagrangian penalties λ_b to the environments.

Algorithm 1 Budget-Aware Policy Optimisation with the `Dual` objective

Require: Policy π_θ; task distribution $p_{\mathcal{X}}$; budget set $\mathcal{B}$ and prior p_B; cost map κ; tolerance δ; dual step β; soft-overlong coefficient α; groups K; group size G; clip range ϵ; horizon $T_{\max}$

Initialise $\lambda_b \geq 0$ for each $b \in \mathcal{B}$
for iteration $k = 0, 1, 2, \ldots$ **do**
  Draw K task and budget pairs $\big(x^{(g)}, B_0^{(g)}\big) \sim p_{\mathcal{X}} \otimes p_B$ for $g = 1, \ldots, K$
  for each group g **do**
    for each rollout $i = 1, \ldots, G$ in group g **do**
      $b_1^{(g,i)} \leftarrow B_0^{(g)}, \;\; \tau^{(g,i)} \leftarrow T_{\max}$ ▷ default if $a_\perp$ is never chosen
      for $t = 1, \ldots, T_{\max}$ **do**
        Sample $a_t^{(g,i)} \sim \pi_\theta\big(\cdot \mid x^{(g)}, h_t^{(g,i)}, b_t^{(g,i)}, B_0^{(g)}\big)$
        $b_{t+1}^{(g,i)} \leftarrow b_t^{(g,i)} - \kappa\big(a_t^{(g,i)}\big)$ ▷ may go negative, no hard wall
        if $a_t^{(g,i)} = a_\perp$ **then**
          $\tau^{(g,i)} \leftarrow t$; **break**
        end if
      end for
      $\zeta^{(g,i)} \leftarrow \big(o_1^{(g,i)}, a_1^{(g,i)}, \ldots, o_{\tau^{(g,i)}}^{(g,i)}, a_{\tau^{(g,i)}}^{(g,i)}, o_{\tau^{(g,i)}+1}^{(g,i)}\big)$ ▷ completed trajectory
      $R^{(g,i)} \leftarrow R\big(\zeta^{(g,i)}\big) \in \{0, 1\}$ ▷ binary terminal reward, truncated at $T_{\max}$ if $a_\perp$ not chosen
      $C^{(g,i)} \leftarrow \dfrac{1}{B_0^{(g)}} \sum_{t \leq \tau^{(g,i)}} \kappa\big(a_t^{(g,i)}\big)$ ▷ normalised cost Equation (5)
      $P_{\mathrm{DAPO}}^{(g,i)} \leftarrow \alpha \,\mathrm{clip}\,\big(C^{(g,i)} - 1,\, 0,\, 1\big)$ ▷ soft-overlong penalty Equation (8)
      $\tilde{R}^{(g,i)} \leftarrow R^{(g,i)} - P_{\mathrm{DAPO}}^{(g,i)} - \dfrac{\lambda_{B_0^{(g)}}}{p_B\big(B_0^{(g)}\big)} \mathbf{1}\big[C^{(g,i)} > 1\big]$ ▷ shaped return, Figure 1
    end for
    $A^{(g,i)} \leftarrow \tilde{R}^{(g,i)} - \frac{1}{G} \sum_{j=1}^{G} \tilde{R}^{(g,j)}$ for each i ▷ Dr. GRPO advantage Equation (27)
  end for
  Update θ by ascending the clipped surrogate $\mathcal{J}(\theta)$ of Equation (28)
  for each $b \in \mathcal{B}$ **do**
    $\mathcal{I}_b \leftarrow \big\{(g, i) : B_0^{(g)} = b\big\}$
    if $\mathcal{I}_b \neq \emptyset$ **then** ▷ update only budgets sampled this batch
      $\hat{p}_b \leftarrow \dfrac{1}{|\mathcal{I}_b|} \sum_{(g,i) \in \mathcal{I}_b} \mathbf{1}\big[C^{(g,i)} > 1\big]$ ▷ raw within-batch rate Equation (29)
      $\lambda_b \leftarrow \min\big\{\big[\lambda_b + \beta(\hat{p}_b - \delta)\big]_+,\; (1 - \alpha)\, p_B(b)\big\}$ ▷ projected dual update Equation (30)
    end if
  end for
end for

F EXPERIMENTAL SETUP

We give the environments, data splits, cost maps, and budget conditioning used in our experiments. Each environment uses the budget-augmented POMDP of Section 3.2, with the total budget B_0 given at the start of the episode and the remaining budget b_t reported after each turn.

F.1 APPWORLD

AppWorld (Trivedi et al., 2024) is an interactive coding-agent benchmark in which the agent completes digital tasks by writing Python code that calls simulated app APIs. Each executed code block is one model turn, including the terminal `apis.supervisor.complete_task()` call.

We train on the AppWorld training split and evaluate on the 168 tasks in `test_normal`, with 8 rollouts per task and budget, giving 1344 rollouts per budget. We also evaluate under distribution shift on the 417 tasks of `test_challenge`, with 4 rollouts per task under step budgets $\{4, 12, 20, 29, 36, 44, 52, 60\}$ at both scales. Standard errors are clustered by task. Budget grids are given in Appendix F.3.

Episodes have horizon $T_{\max} = 40$ turns for Qwen3.5-9B and 50 for Qwen3.5-4B. This exceeds the largest training budget, so overspending remains observable rather than being truncated at the budget.

F.2 ASTRA

ASTRA is a personal-assistant benchmark built on ToolSandbox (Lu et al., 2025), with tasks over email, calendars, messages, and contacts. Its native interface uses structured tool calls rather than AppWorld's Python code. We expose all tools through a single `all` namespace and retain AppWorld's Python-REPL interface, with calls of the form `apis.all.api_name(kwarg=val)`. This keeps the action interface aligned with AppWorld for the transfer experiments of Section 4.3. ASTRA grades success by similarity to a set of reference milestones; we binarise this score at its success threshold of 0.8.

We use ASTRA only to evaluate the AppWorld policies without retraining (Section 4.3). The evaluation covers all 3116 ASTRA scenarios under step budgets $\{2, 4, 6, 8, 10, 12, 14, 16, 24, 32\}$, which span the policies' own trained range, with a 40-turn horizon.

The transfer evaluation uses the Qwen3.5-9B AppWorld step block (Prompt F.3) at both scales. The task prompts also retain the same REPL framing and task instructions where compatible with ASTRA, differing where required by its tool interface. Prompts F.1 and F.2 give both prompts in full.

F.3 COST MAPS AND BUDGETED QUANTITIES

We vary the budgeted quantity through the cost map κ and leave the rest of the method unchanged. Table 3 gives the three cost maps used in our experiments.

Table 3: **Budgeted quantities.** Quantities defined by the cost map κ. Domains are AW AppWorld and AS ASTRA.

Unit	$\kappa(a_t)$	Interpretation of B_0	Domains
Steps	1	Model turns available to the agent	AW, AS
Tool calls	# API calls in a_t	API calls available to the agent	AW
Tokens	$\lvert a_t \rvert$ in tokens	Generated tokens (thinking + action)	AW

Under a step budget, each model turn costs one unit. Under a tool-call budget, each API call is charged independently. Under a token budget, all generated tokens, including reasoning and actions, are charged; task and environment tokens are not.

Budget grids. AppWorld uses six training budgets per grid, per model and cost unit. Evaluation extends each grid below and above its training range and, for Qwen3.5, includes unseen budgets between training points. Table 4 gives every grid.

Table 4: **AppWorld budget grids.** Training and evaluation budget grids, by cost unit and model. Both token evaluation grids span 704 to 11264. The Qwen3.5-4B grid adds its six trained rungs to the 39 anchors of the Qwen3.5-9B grid.

Unit	Model	Training grid $\mathcal{B}$	Evaluation grid $\mathcal{B}_{\text{eval}}$
Turns	Qwen3.5-9B	{10, 15, 18, 22, 25, 29}	{4, 8, 10, 12, 15, 18, 20, 22, 25, 27, 29, 32, 36}
	Qwen3.5-4B	{10, 15, 17, 20, 24, 26}	{4, 8, 9, 10, 11, 12, 15, 17, 18, 20, 21, 22, 24, 26, 29, 31, 32, 34, 36}
Tool calls	Qwen3.5-9B	{10, 15, 18, 22, 26, 29}	{6, 10, 12, 15, 18, 20, 22, 24, 26, 29, 32, 36}
	Qwen3.5-4B	{13, 15, 17, 19, 22, 25}	{6, 10, 11, 12, 13, 14, 15, 17, 19, 20, 21, 22, 24, 25, 26, 27, 30, 31, 35, 36}
Tokens	Qwen3.5-9B	{896, 1408, 1856, 2432, 3136, 3904}	39 anchors, 704–11264
	Qwen3.5-4B	{1452, 2073, 2489, 3066, 4264, 5186}	45 anchors, 704–11264

On ASTRA only the step unit is evaluated, on the grid of Appendix F.2. Tool-call and token evaluation on ASTRA is out of scope for this submission.

F.4 Budget Conditioning and Prompt Templates

The policy sees a single opening user turn holding the domain's task instructions with a `<budget>` block appended directly after the task statement, which states the total budget B_0. At every later turn the environment appends the remaining budget b_t to the observation, so the budget-conditioned policy Equation (3) reads both quantities directly throughout the episode. The header of the block is shared across domains and cost units. The guidance that follows it depends on the unit and, on AppWorld, on the model scale, and every method we compare at a given model and unit uses the same budget text in training and at evaluation alike. Prompt F.1 gives the AppWorld opening turn and Prompt F.2 the ASTRA opening turn. AppWorld is budgeted in three units (Appendix F.3) and so carries one budget block per unit. Prompts F.3 to F.5 give the blocks used for Qwen3.5-9B and Prompt F.6 those used for Qwen3.5-4B. ASTRA uses the step block of Prompt F.3 alone, for both scales. Only the user details, the date, the task string, and the budget B_0 vary between episodes.

Prompt F.1 AppWorld task prompt (opening user turn)

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously.

To do this, you will need to interact with app/s (e.g., spotify, venmo etc) using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with app/s using their associated APIs on my behalf.

Here are three key APIs that you need to know to get more information

To get a list of apps that are available to you.
```
print(apis.api_docs.show_app_descriptions())
```
To get the list of apis under any app listed above, e.g. spotify
```
print(apis.api_docs.show_api_descriptions(app_name='spotify'))
```
To get the specification of a particular api, e.g. spotify app's login api
```
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
```
Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

Key instructions:
(1) Make sure to end code blocks with ``` followed by a newline(\n).
(2) Remember you can use the variables in your code in subsequent code blocks.
(3) You can use the "supervisor" app to get information about my accounts and use the "phone" app to get information about friends and family.

(4) Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
(5) Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
(6) Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
(7) Once you have completed the task, make sure to call `apis.supervisor.complete_task()`. If the task asked for some information, return it as the answer argument, i.e. call `apis.supervisor.complete_task(answer=<answer>)`. Many tasks do not require an answer, so in those cases, just call `apis.supervisor.complete_task()` i.e. do not pass any argument.
(8) Do exactly what the task asks for, nothing more. Do not create, modify, or delete any data that the task does not explicitly require. If the task only asks you to read information, do not write or modify anything. If the task asks you to perform a specific action, perform only that action using the most direct approach available.
(9) If the task mentions a message, email, or request from someone, always read the actual content of that message/email first — it may contain important details, constraints, or filters not stated in the task description.
(10) When a task depends on the current date, time, or day of the week, always determine it programmatically (e.g., `from datetime import datetime; datetime.now()`) — never guess or assume.
(11) When filtering entities by category or relationship (e.g., "friends", "coworkers"), verify membership using the primary/authoritative data source for that relationship — do not infer it from other app-specific lists.
(12) When a task references a time range like "last N days including today", today counts as day 1 — so the start date is N-1 days ago, not N days ago.
(13) When writing structured output files (CSV, etc.), use minimal formatting — do not add unnecessary quoting or encoding around values.
(14) You should follow three phases: First, explore the APIs and all relevant data. You can explore a bit more alternative data just to go sure that you did not miss anything. Second, summarize your findings and compare them to the task again. Think about whether you might have missed something. Look it up and verify. Third, once you have all information, start conducting the task and complete it. Before you finish, verify that every aspect of the task has been accounted for.

Using these APIs, now generate code to solve the actual task:

My name is: {{`main_user.first_name`}} {{`main_user.last_name`}}. My personal email is {{`main_user.email`}} and phone number is {{`main_user.phone_number`}}. The current date is {{`main_user.session_start_time`}}.
Task: {{`input_str`}}

⟨ *budget-conditioning block, appended here* ⟩

Prompt F.2 ASTRA task prompt (opening user turn), aligned to Prompt F.1

I am your supervisor and you are a super intelligent AI Assistant whose job is to achieve my day-to-day tasks completely autonomously.

To do this, you will need to interact with apps using their associated APIs on my behalf. For this you will undertake a *multi-step conversation* using a python REPL environment. That is, you will write the python code and the environment will execute it and show you the result, based on which, you will write python code for the next step and so on, until you've achieved the goal. This environment will let you interact with these APIs on my behalf.

Here are three key APIs that you need to know to get more information

To get a list of apps that are available to you (this only ever lists the single 'all' app).
`print(apis.api_docs.show_app_descriptions())`
To get the list of apis under the 'all' app.
`print(apis.api_docs.show_api_descriptions(app_name='all'))`
To get the specification of a particular api, e.g. the 'all' app's send_email api
`print(apis.api_docs.show_api_doc(app_name='all', api_name='send_email'))`

Each code execution will produce an output that you can use in subsequent calls. Using these APIs, you can now generate code, that I will execute, to solve the task.

Key instructions:
(1) Make sure to end code blocks with ``` followed by a newline(\n).

(2) Remember you can use the variables in your code in subsequent code blocks.
(3) There is exactly one app, named 'all', which holds every task API. Use the `apis.api_docs.*` calls only to look up documentation, and call every task API as `apis.all.api_name(kwarg1=val1, ...)`. Do not invent other app names, and do not paste documentation output back as code.
(4) Always look at API specifications (using `apis.api_docs.show_api_doc`) before calling an API.
(5) Call APIs using only keyword arguments like `apis.all.api_name(kwarg1=val1, ...)`
(6) Write small chunks of code and only one chunk of code in every step. Make sure everything is working correctly before making any irreversible change.
(7) Many APIs return items in "pages". Make sure to run through all the pages by looping over `page_index`.
(8) Once you have completed the task, make sure to call `apis.supervisor.complete_task()`. If the task asked for some information, return it as the answer argument, i.e. call `apis.supervisor.complete_task(answer=<answer>)`. Many tasks do not require an answer, so in those cases, just call `apis.supervisor.complete_task()` i.e. do not pass any argument.
(9) Do exactly what the task asks for, nothing more. Do not create, modify, or delete any data that the task does not explicitly require. If the task only asks you to read information, do not write or modify anything. If the task asks you to perform a specific action, perform only that action using the most direct approach available.
(10) If the task mentions a message, email, or request from someone, always read the actual content of that message/email first — it may contain important details, constraints, or filters not stated in the task description.
(11) When a task depends on the current date, time, or day of the week, always determine it programmatically — never guess or assume.
(12) When filtering entities by category or relationship (e.g., "friends", "coworkers"), verify membership using the primary/authoritative data source for that relationship — do not infer it from other app-specific lists.
(13) When a task references a time range like "last N days including today", today counts as day 1 — so the start date is N-1 days ago, not N days ago.
(14) When writing structured output files (CSV, etc.), use minimal formatting — do not add unnecessary quoting or encoding around values.
(15) You should follow three phases: First, explore the APIs and all relevant data. You can explore a bit more alternative data just to go sure that you did not miss anything. Second, summarize your findings and compare them to the task again. Think about whether you might have missed something. Look it up and verify. Third, once you have all information, start conducting the task and complete it. Before you finish, verify that every aspect of the task has been accounted for.

Using these APIs, now generate code to solve the actual task:

The current date is {{ session_start_time }}.
Task: {{ input_str }}

⟨ *budget-conditioning block, appended here* ⟩

Prompt F.3 Budget-conditioning block, step budget, Qwen3.5-9B (AppWorld and ASTRA)

<budget>You have a budget of B_0 steps.

Only the code blocks you send count toward your step budget. After each step you are told how many steps remain.

Before sending a code block, think carefully about how many steps you have left and plan how best to use them to complete the task within budget. Because a whole block costs just one step, batch several API calls into a single block whenever they don't depend on each other's results — making one call per step wastes budget. Only split work across steps when a later call genuinely needs an earlier call's output.

Solve the task successfully while staying strictly within your allocated step budget.</budget>

Prompt F.4 Budget-conditioning block, API-call budget, Qwen3.5-9B (AppWorld)

<budget>You have a budget of B_0 API calls.

Every `apis.<app>.<endpoint>()` call you make costs 1 from your budget — this includes `apis.api_docs` lookups, not just calls to the task apps themselves. Thinking and plain code are free, and it doesn't matter how many turns your calls are spread across: batching several into one turn costs the same as spreading them out — only making fewer calls total reduces cost. After each step you are told how many API calls remain.

Before making a call, check whether you already have what you need from an earlier call, an earlier doc lookup, or the task description — don't re-look up something you've already seen this episode, and don't make a call just in case. Plan which calls are actually necessary to solve the task.

Solve the task successfully while staying strictly within your allocated API-call budget.</budget>

Prompt F.5 Budget-conditioning block, token budget, Qwen3.5-9B (AppWorld)

<budget>You have a budget of B_0 tokens.

Every token you generate counts toward your token budget: your reasoning between <think> and </think>, plus the code you write after </think>. The task description and the environment's replies are free — only your own output is charged.

After each step you are told how many tokens you have left and how many your previous step used, so you can calibrate as you go. Before each step, decide the total number of tokens that step will use across your reasoning and your code, state that number in your first line, and keep to it — think enough to act well but no more, since verbose reasoning spends budget just like long code. Then re-plan at the next step using the updated count.

Batch independent API calls into one block whenever they don't depend on each other's results; only split work across steps when a later call needs an earlier call's output.

Solve the task successfully while staying strictly within your allocated token budget.</budget>

Prompt F.6 Budget-conditioning blocks, Qwen3.5-4B (AppWorld)

Step budget. <budget>You have a budget of B_0 steps.

Each code block you send counts as one step against your budget. The reasoning you do before sending it is free. After each step you are told how many steps remain.

You finish by calling `apis.supervisor.complete_task()`, and the block containing it counts as a step like any other, so it has to be sent while budget remains.

Solve the task successfully while staying strictly within your allocated step budget.</budget>

API-call budget. <budget>You have a budget of B_0 API calls.

Every `apis.<app>.<endpoint>()` call you make counts as one API call against your budget, including `apis.api_docs` lookups and `apis.supervisor.complete_task()`. How you group calls into code blocks does not change the count: five calls in one block cost five, exactly as five calls spread over five blocks do. Reasoning, and code that makes no API call, are free. After each step you are told how many API calls remain.

You finish by calling `apis.supervisor.complete_task()`, which costs one call, so it has to be made while budget remains.

Solve the task successfully while staying strictly within your allocated API-call budget.</budget>

Token budget. <budget>You have a budget of B_0 tokens.

Every token you generate counts against your token budget: your reasoning between <think> and </think>, and the code you write after </think>. The task description and the environment's replies are free – only your own output is charged. After each step you are told how many tokens remain and how many your last step used.

You finish by calling `apis.supervisor.complete_task()`, and the tokens of the block containing it are charged like any other, so it has to be sent while budget remains.

Solve the task successfully while staying strictly within your allocated token budget.</budget>

At every subsequent turn the environment appends the remaining budget as <budget>Budget remaining: b_t of B_0 {{ unit }}.</budget>, with the unit word matching the cost map, so the policy observes b_t directly throughout the episode. Once the budget is exceeded the line reports 0 remaining and adds `You are` k {{ unit }} `over budget.` Under the token cost map the line also reports the tokens used by the previous step and repeats the request to state a token estimate before generating. The guidance in each block changes with the unit because the lever available to the agent changes with it. Batching several API calls into one block saves budget under a step budget and saves nothing under an API-call budget, while the token budget charges the agent's reasoning alongside its code. The Qwen3.5-9B blocks also suggest a strategy for each unit, whereas the Qwen3.5-4B blocks state only the cost function and the constraint. In every case the header states only the total budget, and the per-turn cost and the update mechanics are carried

by the guidance that follows it. For Qwen3.5-9B, AppWorld and ASTRA use the same step block verbatim, which is what makes the transfer evaluation of Section 4.3 a test of the policy rather than of the wording. The Qwen3.5-4B policies are trained with the step block of Prompt F.6 but evaluated on ASTRA with that of Prompt F.3, so their ASTRA transfer also changes the budget wording.

F.5 Models and Dr. GRPO Settings

We train Qwen3.5 at 4B and 9B parameters and report both scales throughout. Every method continues training from each model's released post-trained checkpoint, with no supervised warm start of our own, and all parameters are updated, with no LoRA, adapters, or frozen layers. The `Rate` policies at Qwen3.5-4B come from an earlier round of training than the other Qwen3.5-4B policies. They were trained for 120 updates at a learning rate of 5×10^{-6} under step and API-call budgets and 3×10^{-6} under token budgets, on the API-call grid $\{10, 14, 17, 20, 24, 27\}$ and the token grid $\{896, 1216, 1536, 2048, 2624, 3456\}$.

Optimisation uses the unbiased Dr. GRPO advantage Equation (27), with neither length nor standard-deviation normalisation, at $\gamma = 1$, for 120 updates on Qwen3.5-9B and 80 on Qwen3.5-4B. The learning rate is flat at 3×10^{-6} with no schedule. Thinking is enabled, with a 768-token allowance on Qwen3.5-9B and 512 on Qwen3.5-4B. Under the step and tool-call cost maps this spends tokens and not budgeted units, so it is orthogonal to the budget, whereas under the token cost map of Appendix F.3 it is priced.

Each iteration draws K groups (Equation (27)), with $K = 8$ on Qwen3.5-9B and $K = 32$ on Qwen3.5-4B, each comprising $G = 8$ completions for a shared task and a shared budget, sampled at temperature 1.0 with nucleus and top-k truncation disabled. The policy update is the clipped surrogate Equation (28) with symmetric clip range $\epsilon = 0.2$ and no KL penalty or entropy bonus, so the return is shaped only by each method's own objective (Section 3.3). Qwen3.5-9B takes 2 epochs per update with gradient accumulation over 8 microbatches of at most 4 segments, and Qwen3.5-4B takes 1 epoch over 2 microbatches of at most 8. Before the update, groups whose advantages are all zero are dropped, and negative-advantage segments are subsampled so that they make up at most a quarter of the batch.

F.6 Budget and Constraint Settings

Budgets are drawn uniformly from $\mathcal{B}$, so $1/p_B(b) = |\mathcal{B}|$ and the effective price of `Dual` and `Rate` (Figure 1) is $\Lambda_b = |\mathcal{B}|\lambda_b$. Episodes run to a horizon $T_{\max}$ set per model, at 40 turns for Qwen3.5-9B and 50 for Qwen3.5-4B on AppWorld, and at 40 for the ASTRA evaluation. Every setting exceeds the largest training budget on its grid, so overspend stays visible to both the dual and the soft-overlong penalty. The ramp of Equation (8) saturates at twice the budget, so under the step cost map it reaches its maximum of α only at budgets with $2b \leq T_{\max}$. On AppWorld this leaves the top three budgets of the Qwen3.5-9B step grid and the top budget of the Qwen3.5-4B step grid short of that maximum, so at those budgets the penalty grades overspend over the whole reachable range without flattening within it.

We set the overspend rate $\delta = 0.05$ and the dual step $\beta = 0.01$, raised to $\beta = 0.04$ on Qwen3.5-4B, whose updates draw four times as many trajectories, so that the step per trajectory is matched. The price integrates at effective rate $|\mathcal{B}|\beta$, which is 0.06 on the six-rung Qwen3.5-9B grids, and 0.24 on the Qwen3.5-4B grids, and we drive the dual with the raw within-batch rate Equation (29), with no smoothing across batches. The soft-overlong coefficient is $\alpha = 0.5$, the value fixed by the reward-scale argument of Appendix C.1.

Multipliers are initialised at $\lambda_b = 0$ for every budget and are projected after every update onto $\big[0, (1-\alpha)p_B(b)\big]$ (Equation (30)), so the effective price Λ_b of Figure 1 never exceeds $1 - \alpha = 0.5$, the largest value at which an overspending success is never rated below compliant failure (Appendix E.2).

F.7 Compute

Training runs on 8× B200 GPUs with tensor-parallel vLLM generation and an FSDP trainer. Evaluation runs on 4× H100 GPUs for Qwen3.5-9B and 8× H100 for Qwen3.5-4B.

Table 5: **Training hyperparameters.** Budget grids are per model and cost unit (Appendix F.3). Settings that differ between the two scales are given per model, and every other setting is common to every method, model, and domain.

Hyperparameter	Value
Model and optimisation	
Model families and scales	Qwen3.5 4B and 9B (released post-trained checkpoints)
Fine-tuning	full-parameter, no LoRA or adapters
Optimiser	Dr. GRPO, $\gamma = 1$
Training updates	120 (Qwen3.5-9B), 80 (Qwen3.5-4B)
Learning rate	flat, no schedule, 3×10^{-6}
Budget family and constraint	
Budget grid $\mathcal{B}$	six rungs per model and cost unit (full grids in Appendix F.3)
Budget prior p_B	uniform on $\mathcal{B}$
Episode horizon $T_{\max}$	AppWorld 40 (Qwen3.5-9B) and 50 (Qwen3.5-4B), ASTRA 40 at evaluation
Overspend rate δ	0.05
Dual step β	0.01 (Qwen3.5-9B), 0.04 (Qwen3.5-4B)
Initial multipliers λ_b	0 for all b, effective price Λ_b capped at $1 - \alpha = 0.5$
Soft-overlong coefficient α	0.5
Rollout collection and policy update	
Groups per iteration K	8 (Qwen3.5-9B), 32 (Qwen3.5-4B)
Group size G	8
Sampling temperature	1.0 (top-p, top-k disabled)
Thinking tokens	768 (Qwen3.5-9B), 512 (Qwen3.5-4B)
PPO clip range ϵ	0.2 (symmetric)
PPO epochs per update	2 (Qwen3.5-9B), 1 (Qwen3.5-4B)
Gradient accumulation	8 microbatches of ≤ 4 segments (Qwen3.5-9B), 2 of ≤ 8 (Qwen3.5-4B)
Gradient-norm clip	1.0
Entropy and KL coefficients	0, no KL term

What the dual costs to train. `Dual` and `Linear-Mag` are the right pair to compare on training cost, since neither ever truncates a rollout and the two therefore differ only in the dual. That difference carries no optimisation overhead. The dual is $|\mathcal{B}|$ scalars updated once per iteration by Equation (29), from a violation estimate already computed for the shaped return of `Dual` (Figure 1), so it adds no parameters, no extra forward or backward pass, and no extra rollouts, so a `Dual` iteration and a `Linear-Mag` iteration do the same optimiser work on the same batch, at the same K, G, and accumulation settings on the same hardware. The two can diverge in wall-clock only through how much the policy chooses to generate, which is a learned effect and not an algorithmic one, and which does not run in the obvious direction, since at matched late updates the `Dual` run trained on *fewer* selected segments per update than its dual-off counterpart, so the price does not inflate the training batch. We therefore treat the two methods as trained at matched compute throughout Section 4.

G LIMITATIONS

Scope of evaluation. We train only on AppWorld and only with Qwen3.5 models at 4B and 9B, so we do not know whether the ranking of objectives holds for other model families, larger scales, or other training domains. Transfer to AppWorld `test-challenge` and ASTRA is evaluated under a step budget alone, so we do not test whether token and tool-call budgeting transfer out of domain. Each policy is also trained and evaluated in a single cost unit, and we do not test whether one policy can serve budgets given in several units.

Cost maps. Our cost map κ prices actions alone, which is exact for environment steps, tool calls, and emitted tokens, but state-dependent and stochastic costs such as realised latency or monetary

spend fall outside it. Widening to $\kappa : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}$ requires no change downstream, since only the realised cost enters Equation (5), but we do not evaluate it. Constraining several quantities jointly, which Liu et al. (2025b) identify as open, would require one multiplier per budget per quantity and remains untested.

Budget families. We train on a finite grid of six budgets per model and cost unit, drawn from a uniform prior. The policies generalise to untrained budgets inside and beyond that grid, but we do not characterise how far, nor which grid or prior best supports extrapolation. We also draw the task and the budget independently, whereas a user who knows a request is hard will often set a looser budget for it. None of the objectives we compare requires that independence, but our grids and results are reported under it.

Compliance and adaptation. No method reaches full compliance, particularly at budgets that are tight relative to the spend of successful trajectories. We argue in Section 4.1 that this partly reflects a trade-off with giving up early on tasks that could still be solved, but we do not separate that trade-off from poor budget awareness. Under API-call budgets the spend of the trained policies is almost flat across budgets, so under that unit they appear to learn a single fixed strategy rather than adapting to the budget they are given.

Inference-time scaffolds. We train the policy and hold the harness around it fixed, so every method we report is trained and evaluated under the same conditions. We therefore do not test budget-aware inference layers on top of our trained policies, such as a value tree search that spends the remaining allowance where step-level value is highest (Li et al., 2026). Nothing in the objectives of Section 3.3 rules such a layer out, since it changes how a policy's outputs are used rather than what the policy is trained to maximise, and a policy that already manages its own spend should be a good starting point for one. A branching search would also need an environment it can restore, which ours do not provide, so layers that do not re-explore are the easier fit. We leave this combination to future work.