Finding the Time to Think: Learning Planning Budgets in Real-Time RL

Aneesh Muppidi[*,1,2‡] **Firas Darwish**[*] **Dylan Cope**[1] **Joao F. Henriques**[†,2]

Jakob Nicolaus Foerster[†,1]

[1] *FLAIR, University of Oxford* [2] *VGG, University of Oxford*

[*]Equal contribution. [†]Equal advising.

Abstract

Deliberating takes time. In real-time settings, that time is not free. Standard reinforcement learning (RL) sidesteps this as the environment waits indefinitely for the agent's decision. Instead, we study real-time RL environments where the environment progresses while waiting for the agent's action. Building on prior real-time formalizations, we introduce *variable-delay real-time RL*, where the agent chooses how long to deliberate at each decision point since the environment progresses. For the planning agents we use, the right delay is state-dependent, and naively planning how long to plan can paralyze the agent. We instead approach this setting by training a lightweight gating policy on top of a planner to select state-dependent planning budgets. Across real-time Pac-Man, Tetris, Snake, Speed Hex, and Speed Go, our gating policy outperforms fixed-budget and heuristic baselines, and transfers to a real-time setup where the environment and agent run on two different GPUs. Project website, code, and checkpoints: aneeshers.github.io/realtime-rl/.

1 Introduction

Expert decision-makers do not think equally hard about every choice. They must recognize which decisions require more time or effort to think through and which decisions can be made quickly, as they are constrained in the cognitive resources they can devote to decision-making (Otto & Daw, 2019). For example, a skilled chess player managing the clock in speed chess will play most moves quickly and reserve their time for critical board positions when they need to think for longer. In standard RL, and Markov Decision Processes (MDPs), the environment waits as the agent may deliberate indefinitely before committing to an action, and computation does not carry any cost.

Real-time settings break this assumption: the world keeps progressing even if the agent takes longer to act. The cost of deliberation is not wall-clock latency, but steps taken by the environment while the agent is still deciding. A perfect action executed one moment too late may be far worse than a timely, good-enough action.

Ramstedt & Pal (2019) formalized this concern by fixing the action delay at exactly one timestep. We generalize Ramstedt & Pal (2019)'s framework and introduce *variable-delay real-time RL*, where the agent is a procedure that runs in time rather than a one-frame function, so the time any particular action-producing computation spans is paid as progress in the world.

[‡]Correspondence to: aneesh.muppidi@magd.ox.ac.uk. Version of June 17, 2026.

We explore this setting with agents that use anytime planning algorithms to improve the quality of their actions, but must confront that doing so takes time in which the environment continues to evolve. Specifically, we use AlphaZero-style models (Silver et al., 2018) that run Monte Carlo Tree Search (MCTS) at decision time to improve action quality (see Appendix B for a primer). A property of these agents is that more search produces better actions but takes longer to run, and we verify empirically (Figure 3) that planning quality and inference latency scale together.

Our setting introduces three challenges. Firstly, choosing the right delay is state-dependent: some states reward careful planning while others demand immediate reaction. Secondly, the agent must commit to *something* during the intermediate frames, since the environment will not wait for the planner. Lastly, choosing how long to deliberate is fundamentally a problem of *meta-reasoning* (Russell & Wefald, 1991; Horvitz & Klein, 2013), which risks an obvious *paradox of planning-about-planning*: the meta-decision occurs while the environment continues to evolve, so deciding how long to think could incur the same per-frame cost as thinking itself.

We address these challenges by training a lightweight *gating policy* on top of a *frozen* planner that decides how long to plan at each decision point. Our design escapes the paradox in practice because a single gate forward pass is orders of magnitude cheaper than an MCTS rollout, and therefore adds negligible real-time overhead. We employ our *gating policy* in *real-time* Pac-Man, Tetris, Snake, in which the world progresses while the planner runs in the background (Figure 1), and *clock environments* (Speed Hex, Speed Go), in which the board is static but each player has a clock that depletes with thinking time. Section 3 formalizes both.

Our contributions are the following:

1. **Variable-delay real-time RL.** A generalization of Ramstedt & Pal (2019)'s fixed-delay framework: under the real-time interaction protocol the agent is a procedure that runs in time rather than a function evaluated instantaneously, and the time spent producing any action is paid as progress in the world.
2. **Planning quality vs. inference time.** We empirically characterize how planning quality and real-time inference cost co-scale with MCTS simulation count, establishing the joint tradeoff that motivates adaptive allocation.
3. **Adaptive gating on a frozen planner.** A lightweight gating policy, trained with PPO on top of a frozen AlphaZero planner, that selects a state-dependent planning budget at each decision.

Across five environments, the gating policy which adaptively selects among a set of planning budgets outperforms baselines constrained to use a single fixed planning budget at every step. To validate that the environments and method we designed capture the challenges of real-time decision-making, we transfer the learned gating policy to a real-time hardware setting where the environment runs on one GPU and the agent runs on a second GPU without any retraining or architectural changes to demonstrate that it performs effectively.

2 Related Work

Real-time RL and delayed MDPs. The standard MDP assumes the environment waits during action selection. Travnik et al. (2018) flagged this as a fundamental mismatch with real-time interaction, and Ramstedt & Pal (2019) formalized an alternative in which the agent has exactly one timestep to compute its next action, mathematically equivalent to a 1-step constant-delay MDP (Walsh et al., 2009). Subsequent work extends this to fixed action and observation delays of larger length

(Derman et al., 2021; Bouteiller et al., 2020; Katsikopoulos & Engelbrecht, 2003), to concurrent control (Xiao et al., 2020), and to asynchronous and pipelined architectures that mitigate inference cost (Riemer et al., 2024; Anokhin et al., 2025). Across this literature, delay is imposed by the system. We instead let the agent choose its delay at each decision point, turning delay selection itself into the learning problem.

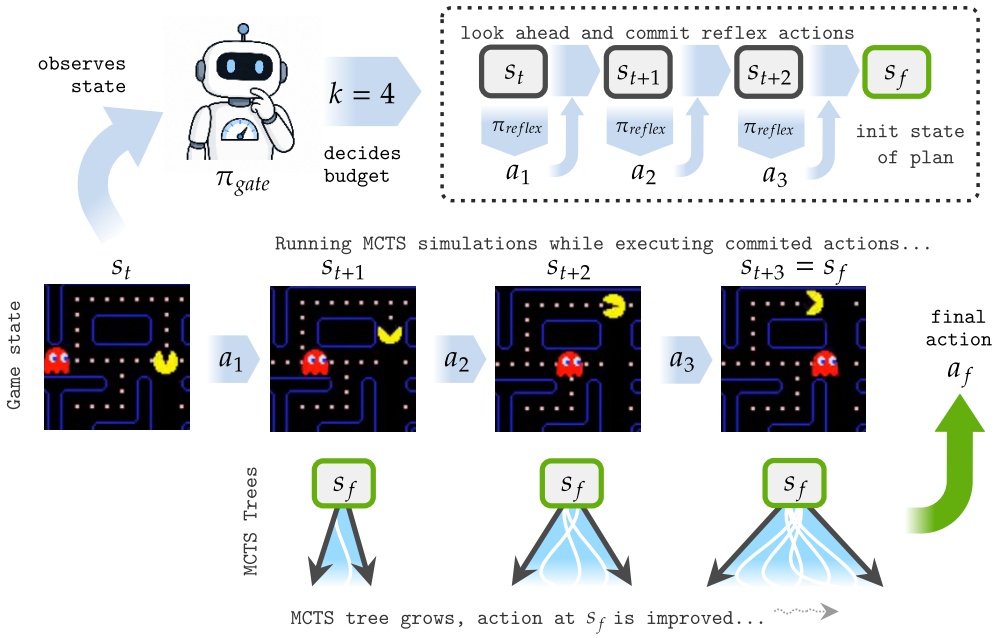

Figure 1: Given the current state, the gating policy chooses whether to react immediately or spend time planning, selecting the number of timesteps k over which to plan. The agent then takes $k-1$ committed actions using π_{reflex} (π_0) while MCTS plans, and finally executes the planned action.

Value of computation and meta-reasoning. The view that rational agents must reason not only about what to do but about how much to think traces to bounded rationality (Simon et al., 1972) and Good's *Type II rationality* (Good, 1952). Russell & Wefald (1991) made this operational by defining the *value of computation* (VOC) as the expected improvement in decision quality minus its cost, and bounded optimality (Russell, 1991) reframed rational agency as the optimal use of fixed computational resources. Anytime algorithms (Dean et al., 1988; Zilberstein, 1996; Likhachev et al., 2003; Korf, 1990) produce monotonically improving answers given more time, and Bayesian flexible-computation methods (Horvitz et al., 1989) formalize when to stop. Applied to MCTS, value-of-computation ideas yield Bayesian formulations of which simulation to run next (Hay et al., 2014; Tolpin & Shimony, 2012; Sezener & Dayan, 2020; Lin et al., 2015), while classical chess- and Go-engine time management allocates clock per move via hand-designed heuristics (Baier & Winands, 2015; Huang et al., 2010; Baudiš & Gailly, 2011). The cognitive-science literature models bounded deliberation itself as a meta-MDP whose actions are computations (Lieder & Griffiths, 2017; Griffiths et al., 2019; Callaway et al., 2017; Cope et al., 2023) and shows empirically that humans allocate planning effort to high-stakes decisions in ways predicted by this model. Our gating policy fits this lineage: it is a learned estimator that chooses total budget per decision rather than which simulation to run next, formalized as a meta-MDP over a frozen planner.

Adaptive compute in model-based RL. The closest neighbors to our work learn to allocate planning compute on top of a model-based RL agent. Metacontrol (Hamrick et al., 2017) chooses how many imagination steps to run; Thinker and Dynamic Thinker (Chung et al., 2023; Wang et al., 2025) let the agent decide when to imagine alternative trajectories on top of a learned world model; Hamrick et al. (2020) provides the key empirical motivation: shallow MuZero trees often suffice, and the marginal value of additional simulations varies sharply by state. AlphaZero-family planners (Silver et al., 2018; Schrittwieser et al., 2020; Danihelka et al., 2022) treat MCTS budget as a fixed deployment hyperparameter. Other axes of "learning to search" (Guez et al., 2018; Farquhar et al., 2017; Hamrick et al., 2019; Racanière et al., 2017) modify the planner's internal behavior rather than its budget; we hold the planner fixed and control only how long it runs.

Adaptive compute in language models. The same thesis has been independently developed for sequence models. PonderNet and adaptive computation time (Graves, 2016; Banino et al., 2021; Schuster et al., 2022) learn per-input halting; test-time compute scaling (Snell et al., 2024; Guo et al., 2025; Muennighoff et al., 2025) and RL-trained reasoning-length controllers (Aggarwal & Welleck, 2025; Muppidi et al., 2025; Fang et al., 2025; Shen et al., 2025) learn per-query thinking budgets; cascaded routing (Kim et al., 2023; Chen et al., 2023; Ong et al., 2024) defers easy queries to cheap models. The shared thesis is that a small learned policy can profitably decide how much computation to spend per decision. Our setting differs structurally: the cost of thinking is endogenous to the environment dynamics, so thinking longer changes the state the agent eventually acts in rather than incurring an external penalty.

Our setting. We share the thesis of the works above but address a setting none of them does. We generalize Ramstedt & Pal (2019)'s fixed-delay framework to variable delay, where the cost of deliberation is paid in *environmental progression* rather than artificial reward shaping, so the gating policy learns the value of simulation budgets from outcomes alone. We gate a *frozen* AlphaZero-style planner with no joint training, formalized as a meta-MDP over options (Sutton et al., 1999; Bacon et al., 2017) whose holding times are the chosen budgets. We also introduce a protocol that makes this real-time cost legible during training and that describes the dynamics of true asynchronous execution, so the trained policy transfers to a two-GPU deployment with no architectural change.

3 Variable-Delay Real-Time RL

3.1 Problem: real-time MDPs

Consider a standard MDP $E = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$. The *real-time interaction protocol* changes only how the agent and environment exchange actions: the environment advances by one frame at fixed intervals $t = 0, 1, 2, \ldots$ regardless of the agent, applying at each frame whatever action the agent has submitted by then and defaulting to an environment-defined fallback (typically a no-op) if none. The MDP and the return $\mathbb{E}[\sum_t \gamma^t r_t]$ are unchanged.

The agent is therefore no longer a function $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ but a *procedure* that runs in time and emits an action at each frame. This is the only change—but it has a sharp consequence: thinking costs progress in the world, since the environment advances while the agent computes.

Relation to existing real-time RL. The Real-Time MDP of Ramstedt & Pal (2019) is the special case in which the agent's procedure is constrained to be a function $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ that takes exactly one frame to evaluate—equivalent to a 1-step constant-delay MDP (Walsh et al., 2009). Fixed-delay MDPs of length K (Walsh et al., 2009; Derman et al., 2021) fix the procedure's latency to K frames. Our setting subsumes both: any procedure satisfying the one-action-per-frame contract is valid, and the agent—not the environment—decides how many frames any particular computation spans.

3.2 Solution: an SMDP over budgeted options

The real-time protocol leaves the agent's procedure unspecified. Our framework constructs one from three ingredients: a fast *reflex policy* that supplies an action every frame, a finite set of slow-but-better *anytime action-refinement computations* that the agent invokes when it can afford to wait, and a learned *gating policy* that decides at each meta-decision which computation (if any) to run. The first two are combined into temporally extended *budgeted options*; the gating policy operates as a meta-policy over those options in a semi-Markov decision process (SMDP).

Reflex policy. We commit to a single fast policy $\pi_{\text{reflex}}(a \mid s)$ that runs in well under one frame. It supplies the agent's frame-by-frame output under the real-time protocol regardless of what else is being computed in the background. Sections 3.3 and 3.4 instantiate π_{reflex} per environment.

Anytime action-refinement computations. We additionally equip the agent with a finite family of *anytime action-refinement computations* $\{c_k\}_{k\in\mathcal{K}}$ (Russell & Wefald, 1991; Zilberstein, 1996), indexed by discrete duration $k \in \mathcal{K}$: each c_k is an anytime algorithm whose job is to refine the action choice given more compute. Computation c_k runs for exactly k frames once initiated and, at the start of its k-th frame, produces a refined action distribution $\pi_k(\cdot \mid s_{t_n})$ at the state s_{t_n} where it was initiated; longer-running c_k produce expectedly better actions, a property we verify for our instantiation in Figure 3. Throughout this paper we instantiate c_k as MCTS run for k frames from s_{t_n}, but the framework applies to any anytime action-refinement algorithm with known per-budget duration.

Budgeted options. For each $k \in \mathcal{K}$ we define a *budgeted option* o_k[1] that wraps c_k into a single temporally extended action: o_k initiates c_k, emits $k-1$ *committed actions* drawn from π_{reflex} during the wait window, and on the terminal frame applies c_k's output. Concretely, if o_k is initiated in state s_t, then for the intermediate frames $j = 0, \ldots, k-2$,

$$a_{t+j} \sim \pi_{\text{reflex}}(\cdot \mid s_{t+j}), \qquad s_{t+j+1} \sim P(\cdot \mid s_{t+j}, a_{t+j}), \tag{1}$$

and on the terminal frame

$$a_{t+k-1} \sim \pi_k(\cdot \mid s_t), \qquad s_{t+k} \sim P(\cdot \mid s_{t+k-1}, a_{t+k-1}), \tag{2}$$

where π_k is the action distribution produced by c_k initiated at s_t. The option induces a transition kernel $P_k(s' \mid s) = \Pr(s_{t+k} = s' \mid s_t = s, o_k)$ and an option-level reward

$$R_k(s) \;=\; \mathbb{E}\left[\sum_{j=0}^{k-1} \gamma^j \, r(s_{t+j}, a_{t+j}) \;\middle|\; s_t = s, o_k\right]. \tag{3}$$

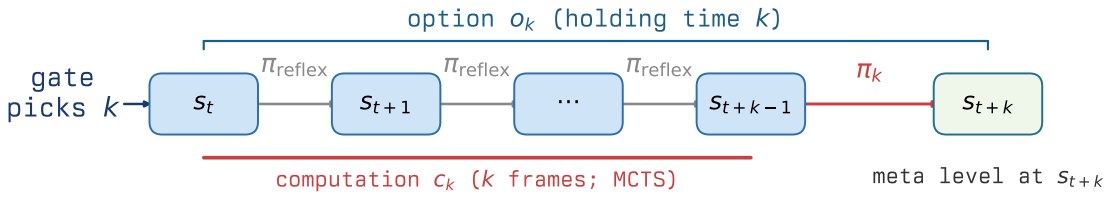

Figure 2: Timeline of a budgeted option o_k. The agent emits $k-1$ committed actions from π_{reflex} while computation c_k runs (instantiated as MCTS for k frames), then applies c_k's output π_k and returns to the meta level at s_{t+k}. In clock environments, the committed steps are no-ops that only consume clock.

Meta-level SMDP. The gating policy $\pi_{\text{gate}}(k \mid s_t)$ is a meta-policy that chooses among the $|\mathcal{K}|$ budgeted options. At each meta-decision state s_t the agent samples k, executes o_k, and returns to the meta level in state s_{t+k} after k primitive frames. The induced control problem is a semi-Markov decision process (Sutton et al., 1999; Bacon et al., 2017) with meta-state space $\mathcal{S}$, meta-action space $\mathcal{K}$, holding time $\tau(k) = k$, kernel P_k, and reward R_k. Its meta-Bellman equation is

$$V(s_t) \;=\; \mathbb{E}_{k\sim\pi_{\text{gate}}(\cdot\mid s_t)}\left[\underbrace{\sum_{j=0}^{k-1}\gamma^j r_{t+j}}_{\text{option reward over } k \text{ frames}} \;+\; \underbrace{\gamma^k V(s_{t+k})}_{\text{discounted bootstrap}}\right]. \tag{4}$$

[1]Our budgeted options are a special case of the semi-Markov options of Sutton et al. (1999): an option is a triple $\langle \mathcal{I}, \pi, \beta\rangle$ with input set $\mathcal{I}$, internal policy π, and termination condition β. Sutton et al. (1999) note that "[s]ometimes it is useful for options to 'timeout,' to terminate after some period of time has elapsed even if they have failed to reach any particular state," and that this requires the semi-Markov generalization in which β may depend on the history since the option was initiated. Each o_k uses exactly this construction: deterministic termination after k primitive frames. See Sutton, Precup, and Singh (1999), §4 for the full formalism.

The gating policy is therefore solving a discrete control problem: not which primitive action to take now, but which temporally extended computation-and-control routine to invoke. In simulation we throttle MCTS to a fixed number of simulations per frame (32 in the committed-action environments; see Section 3.3), so the holding time of o_k is exactly k frames by construction and this SMDP has deterministic holding times. In real-time deployment, MCTS runtime is slightly stochastic, so the hardware system is closer to a continuous-time SMDP. Appendix L discusses this distinction.

Why options rather than state augmentation. The RTMDP framework (Ramstedt & Pal, 2019) handles its fixed one-step delay by augmenting the state with the action currently in flight, $\mathbf{x}_t = (s_t, a_t)$, with transition kernel

$$P_{\mathrm{RT}}(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \hat{a}_t) = P(s_{t+1} \mid s_t, a_t)\, \delta(a_{t+1} - \hat{a}_t), \tag{5}$$

where δ is a Dirac delta. This is the natural representation when delay is fixed and the agent emits nothing during the wait window. In our setting both assumptions fail: k is chosen by the agent at every decision frame, and during the wait window the agent is actively controlling the environment through π_{reflex}. State augmentation would have to grow with $|\mathcal{K}|$ and encode the within-option phase; budgeted options collapse the within-option dynamics into a single SMDP transition, leaving the gating policy a standard meta-decision problem.

3.3 Committed-action environments: Pac-Man, real-time Tetris, Snake

In committed-action environments, acting and planning happen concurrently. The $k-1$ committed actions are drawn from π_{reflex}, which we instantiate as the planner's own policy network evaluated without MCTS, computed via a single forward pass in roughly two milliseconds, while the full MCTS runs in the background. We choose this instantiation rather than a separately trained reflex policy because it requires no additional training and keeps the π_{reflex} aligned with the MCTS tree's internal rollouts. On the k-th frame the planner's chosen action is applied. MCTS rolls out the same $k-1$ committed steps that the agent will actually execute, so search is performed over the future state in which the planned action will land. The cost of deeper planning is therefore exactly $k-1$ additional frames of world progression before the carefully chosen action lands.

We instantiate this setting in three environments derived from `Jumanji` (Bonnet et al., 2024): Pac-Man, Snake, and a real-time variant of Jumanji Tetris (Appendix D.1). The source of progression differs across environments (ghost motion in Pac-Man, gravity in real-time Tetris, body elongation in Snake), but in all cases a deeper plan arrives later, after the world has changed.

We use $\mathcal{K} = \{1, 2, 3, 4\}$ and calibrate 32 MCTS simulations to one environment frame, which corresponds to roughly 9 FPS on an H100 (Section 7). At budget k the planner runs $32k$ simulations over k frames, so each frame still receives 32 simulations on average regardless of k. Differences in performance reflect *when* compute is allocated, not the average compute spent per frame.

3.4 Clock environments: Speed Hex and Speed Go

Clock environments instantiate our framework with a degenerate reflex policy: π_{reflex} deterministically emits the no-op action, so committed steps leave the board unchanged and only c_k's terminal action affects play. The clock is therefore the only dynamic element of the environment; it decrements with thinking time over the k frames of an option, while the board state remains fixed until the planner's chosen action is applied. Intuitively, this is the same pressure as in speed chess: the board waits for your move, but your limited clock keeps running while you think.

We instantiate this setting in Speed Hex (11×11) and Speed Go (9×9), both board-game environments built on `pgx` (Koyamada et al., 2023). We use a one-to-one mapping between MCTS simulations and clock ticks: each simulation increments the player's clock by one unit.

For clock environments the equal-compute-per-frame constraint does not apply, so we calibrate the simulation options separately to ensure each option represents a meaningfully distinct tradeoff

between inference latency and planning quality. Calibration details and the resulting option sets $\mathcal{K} = \{2, 8, 32, 128\}$ for Speed Hex and $\mathcal{K} = \{16, 32, 64, 96\}$ for Speed Go are in Appendix E.

4 Adaptive Gating Policy

4.1 The planning tradeoff

The core observation motivating our approach is that planning quality and inference cost scale together. As an agent runs more MCTS simulations, its action quality improves; but more simulations also take longer to run, and in real-time settings that longer inference is exactly what advances the environment, or depletes the clock, before the agent acts. We empirically verify this relationship in Figure 3. The full five-environment scaling results appear in Appendix E. This joint scaling means we do not need to learn *how* to plan as the planner handles that, only *when* the additional quality gain is worth the additional real-time cost.

4.2 The gating policy

The gating policy takes three inputs at each meta-step: (1) the raw game observation, (2) the planner's intermediate spatial features extracted from its frozen trunk and (3) the planner's scalar value estimate $V(s_t)$. In clock environments the observation also includes the remaining time budget. Together these give the gating policy both a direct view of the board and a compressed summary of the planner's own confidence, without requiring any access to the planner's internals beyond a single forward pass. A lightweight network processes these inputs and produces a distribution over k and a baseline value for training. All architecture details are in Appendix I.

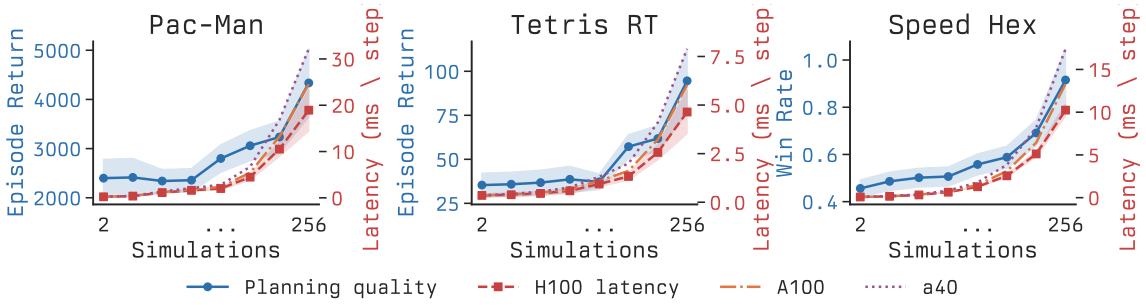

Figure 3: Across Pac-Man, Tetris, and 2-player Speed Hex, planning quality rises with simulation count while inference latency rises alongside it. Blue denotes performance; dashed curves denote per-step latency on H100, A100, and A40; shading shows ±SE.

4.3 Training

We train in two phases. First, we train AlphaZero-style base planners for each environment. We then freeze the selected base planner and train the gating policy with PPO (Schulman et al., 2017) on top of it. For the clock environments, these base planners are trained by self-play (Appendix I). Because each meta-step has variable duration k, we adapt Generalized Advantage Estimation (Schulman et al., 2015) to carry the per-meta-step discount γ^k through the advantage computation (see Appendix C). Making this meta-MCTS AlphaZero stack computationally feasible requires careful attention to several JAX implementation details, which we summarize in Appendix I.

5 Experiments

Baselines. For real-time Pacman, Tetris, and Snake we compare against: *always-k* policies for each $k \in \mathcal{K}$; and a *random* policy selecting k uniformly at each meta-step. For speed environments we

additionally compare against *greedy* (always use maximum simulations) and *midpeak* (a bell-curve allocation that concentrates compute at the critical mid-game, a common heuristic in game engines Baier & Winands (2015); Huang et al. (2010)). For these clock heuristics we tune the bell-curve shape separately at each evaluation budget and report the best resulting strategy. All results are averaged over 100 independent episode seeds; we report mean ± standard error in Fig. 4.

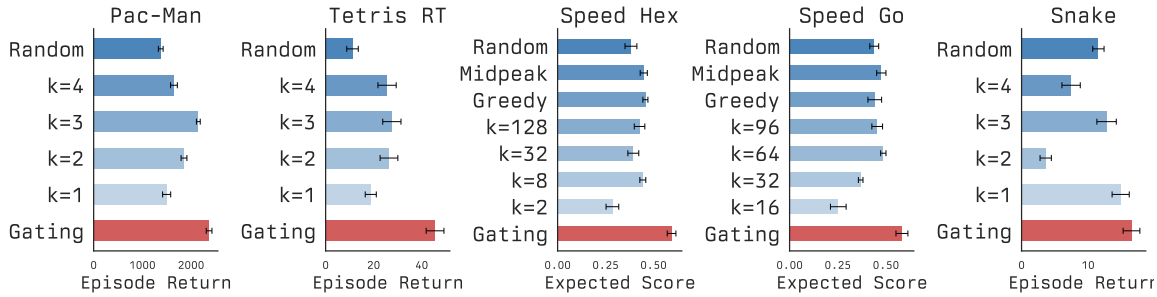

Figure 4: Across all five environments, the gating policy outperforms fixed-budget and heuristic baselines, showing that adaptive allocation matters more than committing to a single search budget. Bars show mean ± SE over 100 episodes; for Speed Hex and Speed Go, expected score is averaged over the shared sampled clock budgets.

For both Speed Hex and Speed Go, we evaluate across the same five sampled clock budgets, $T \in \{300, 1200, 2300, 3500, 4100\}$, and average the resulting head-to-head expected scores. The fixed-budget baselines use 2/8/32/128 simulations for Speed Hex and 16/32/64/96 simulations for Speed Go. In these main clock benchmarks, exhausting the clock does not immediately end the game. We found that simply exhausting the clock was too easy a setting (Appendix H), so we study the harder π_{reflex}-fallback setting, where control passes to π_{reflex} after the clock runs out and the task becomes one of resource allocation rather than simply avoiding timeout.

The gating policy outperforms every fixed-budget baseline across all environments. In Pac-Man, the best fixed policy (k=3, 96 sims/step) scores 2149; the gating policy reaches 2370 (+10.3%). In real-time Tetris, the best fixed policy (k=3) scores 27.6; the gating policy reaches 45.6 (+65%). In Snake, the best fixed policy (k=1) scores 14.91; the gating policy reaches 16.54 (+10.9%). In Speed Hex, averaged over the shared sampled clock budgets, the gating policy reaches an expected score of 0.58, compared with 0.43 for the best fixed-budget baseline (k=128) and 0.46 for the best heuristic baseline (greedy). In Speed Go, averaged over the same budgets, the gating policy reaches an expected score of 0.59, compared with 0.51 for the best fixed-budget baseline (k=64) and 0.50 for the best heuristic baseline (midpeak).

Across environments, the gate learns dynamic, state and budget-dependent allocation strategies that outperform both fixed-budget policies and hand-designed heuristics (Figure 6, Appendix Figure 11). The random baseline falls well below every fixed-k policy in both environments, confirming that *when* to allocate compute matters as much as how much.

6 Analysis

6.1 What triggers deeper planning?

Across the environments, our gating policy allocates compute in states where the consequence of a suboptimal action is large and additional search budget is likely to change the decision (Figure 5).

Pac-Man. In the current evaluation set, larger chosen budgets are associated with greater nearest-ghost distance in Pac-Man: when the nearest ghost is close, the policy defaults to the more reactive k=1 option; at intermediate distances it shifts toward k=2; and when the threat is farthest away it

can afford the deeper k=4 plan. We also see in Appendix G that the gating policy becomes more reactive late in the episode.

Real-time Tetris. Board density is the dominant feature in real-time Tetris. k=1 is selected almost exclusively on near-empty boards (fill ≈ 0.05, stack ≈ 3.5 rows) where any reasonable placement is acceptable. k=2 and k=4 are selected on dense boards (fill ≈ 0.25–0.32, stacks ≈ 12 rows) where placement precision is consequential. k=3 is never used (0%), producing the bimodal "react or plan deeply" strategy visible in Figure 5. Piece type also modulates budget: the Z-piece ($\bar{k}$=2.98$\pm$0.04) and O- and T-pieces ($\bar{k}$=2.88$\pm$0.04) receive the most deliberation, while the J-piece ($\bar{k}$=2.66$\pm$0.04) receives the least, reflecting differences in placement geometry and expected outcome variance.

Snake. Spatial constraint, rather than simple goal distance, is the main trigger. k=1 dominates on open boards, while k=3 appears when reachability drops, local body density rises, and the snake has fewer safe continuations. k=2 occupies a milder regime associated with navigating toward more distant fruit on otherwise open boards. Immediately after eating, when the body has just grown, k=3 usage jumps from 3.2% to 13.5%, indicating that the policy treats body-growth events as a prospective spatial-risk signal.

Clock environments. In the clocked PGX games, a key trigger is the interaction between board state and remaining time. Figure 6 shows the resulting shift in allocation for both Speed Hex and Speed Go. Under the small clock budget (T=300), the policy heavily favors the cheapest option, reflecting the high opportunity cost of thinking. Under the larger clock budget (T=4100), the same learned policy spreads probability mass much more evenly across the budgets, allocating deeper search substantially more often. Because the policy is trained across many clock settings rather than tuned separately at each one, this shift is evidence of budget-conditioned generalization.

7 Real-Time Deployment

We validate the committed-action framework in a two-GPU asynchronous setup across three committed-action environments: real-time Tetris, Pac-Man, and Snake. One GPU runs the environment and executes committed actions continuously; a second GPU runs the MCTS planner. At each meta-decision the current game state is transferred to the planning GPU, which begins search while the environment GPU keeps the game moving with committed actions. When planning finishes, after $k \times 32$ simulations, the result is applied at the next environment step (Figure 7).

Figure 8 summarizes 45 measured deployments: 3 environments $\times$ 3 GPU classes $\times$ 5 frame rates (8–12 FPS). The asynchronous execution pattern is unchanged from training (see Section 3.3): a k=4 meta-step at 9 FPS spans four 111 ms frames, while committed actions from the reflex policy keep the environment moving and MCTS finishes asynchronously on the second GPU.

The main result, as seen in Fig. 8, is that simulation-trained policies transfer cleanly to asynchronous hardware deployment over a broad operating range. Latency remains well within budget on H100 in all three environments, and remains reliable on A100 except near the tightest deadline regimes. Return transfer is similarly strong. At 9 FPS, deployed returns remain close to their simulation counterparts across all three environments, and the learned budget distribution also transfers cleanly. In real-time Tetris on H100, the policy preserves the same strongly bimodal "react or plan deeply" strategy seen in simulation. Appendix J gives the full per-environment, per-FPS, per-GPU analysis.

This result tests a key hypothesis behind the committed-action training protocol. We treat the $k-1$ committed steps executed in simulation as a model of real asynchronous deployment: if this abstraction is correct, then a policy trained under it should transfer to a hardware-separated environment-and-planner system without modification. Because training simulates the planning delay by actually executing committed actions in the environment, the transfer result supports this hypothesis and suggests that the separation between environment and planner is the right abstraction for real-time deployment.

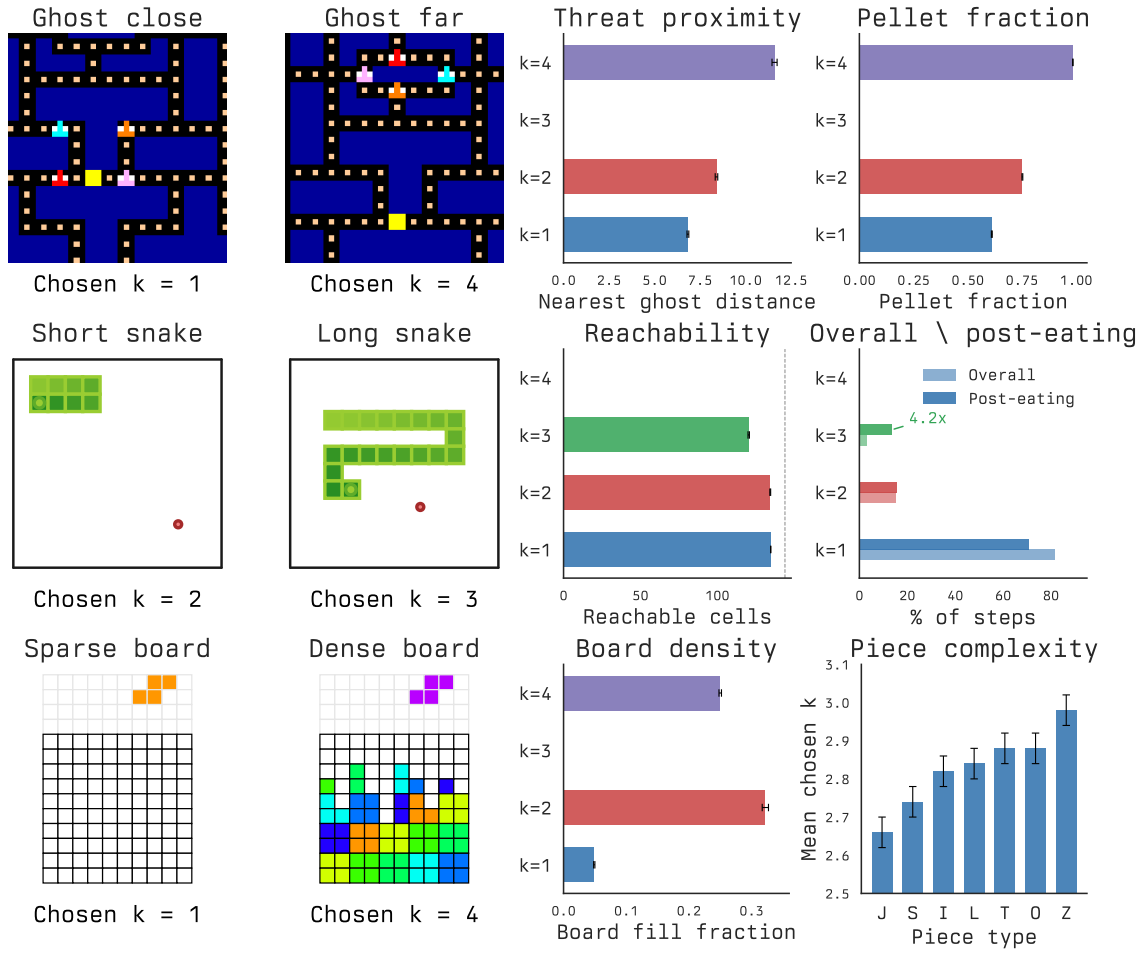

Figure 5: The policy plans deeply precisely when the state is dangerous or constrained. Across Pac-Man, real-time Tetris, and Snake, larger chosen budgets are associated with higher threat, denser boards, or fewer safe continuations, indicating that the gate is responding to meaningful decision difficulty. Plots show state features conditioned on chosen budget k (mean ± 1 SE, 100 episodes).

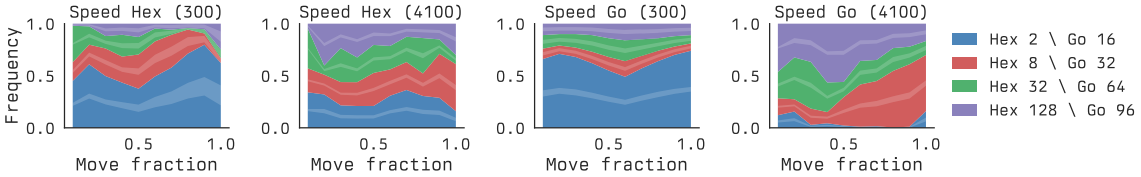

Figure 6: In both Speed Hex and Speed Go, the learned policy is much more reactive under the small budget (T=300) and distributes mass toward deeper options once more clock is available (T=4100).

8 Conclusion

We generalized Ramstedt & Pal (2019)'s fixed-delay real-time RL framework by treating the agent as a procedure that runs in time rather than a function evaluated instantaneously: under the real-time interaction protocol the environment advances every frame regardless, and the time spent producing any action is paid as progress in the world. We then instantiated this with an SMDP over budgeted options, in which a lightweight gating policy trained with PPO on top of a frozen AlphaZero planner learns to invest more search where it matters and react quickly elsewhere. Across

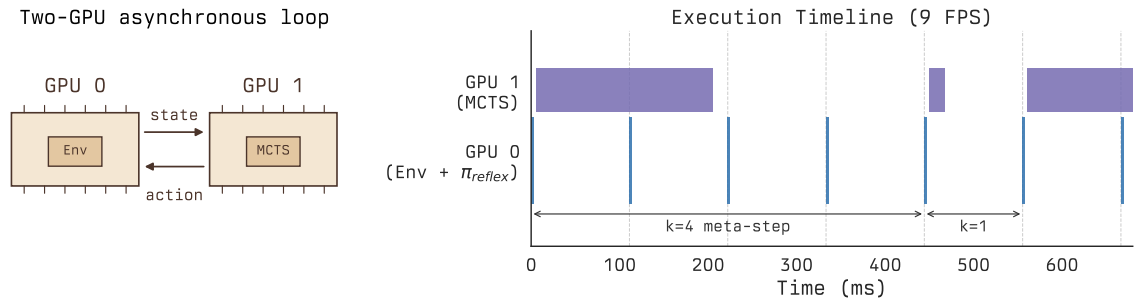

Figure 7: Two-GPU asynchronous deployment pipeline. GPU 0 runs the environment continuously via committed (reflex) actions; GPU 1 runs MCTS in parallel, so a larger k buys more planning time without pausing the game. *Left:* schematic of the two-GPU loop. *Right:* execution trace at 9 FPS, a k=4 meta-step spans four 111 ms frames, followed by a k=1 recovery step.

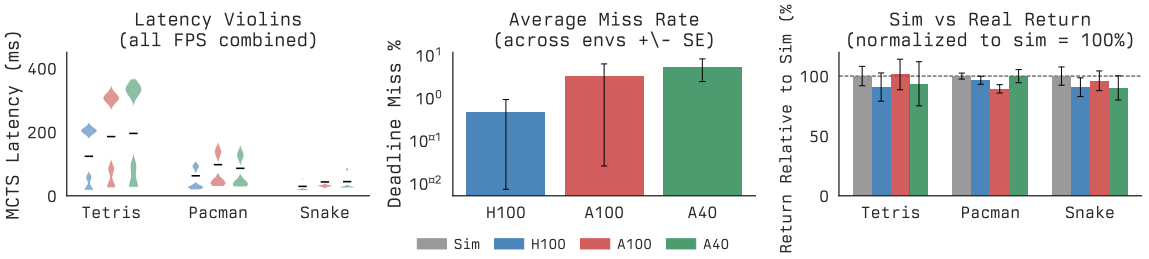

Figure 8: Simulation-trained policies transfer cleanly to hardware deployment. *Left:* MCTS latency variance tightens with GPU class—H100 is most consistent, A100 intermediate, A40 widest. *Centre:* H100 and A100 miss deadlines rarely; A40 breaks down only at the tightest frame rates. *Right:* Deployed returns at 9 FPS match simulation closely across all environments and GPU classes.

five environments spanning committed-action and clock-based real-time mechanisms, the gating policy outperforms every fixed-budget baseline, and the simulation-trained policy transfers to a true two-GPU asynchronous deployment with no architectural change. While we instantiate the framework with MCTS, it applies more broadly to any anytime action-refinement algorithm with known per-budget duration.

Limitations. Our committed-action protocol relies on the MCTS tree faithfully simulating the $k-1$ committed steps, which assumes a perfect environment simulator; extending to learned-dynamics planners (e.g., MuZero) is natural future work. The gating policy is trained on top of a *frozen* base planner, and Appendix F shows that the choice of base checkpoint substantially affects downstream performance; joint optimization of planner and gate remains open. We hand-calibrate a small discrete budget set per environment; continuous or richer compute vocabularies are natural extensions but require recalibrating the cost-quality tradeoff.

Acknowledgments

We thank Justin Svegliato for valuable feedback on our metareasoning definitions and framing, and Mattie Fellows and Uljad Berdica for helpful discussions and feedback on earlier drafts. A. Muppidi and F. Darwish are supported by the Rhodes Scholarship (Rhodes Trust). The authors declare no competing interests.

References

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning. *arXiv preprint arXiv:2503.04697*, 2025.

Ivan Anokhin, Rishav Rishav, Matthew Riemer, Stephen Chung, Irina Rish, and Samira Ebrahimi Kahou. Handling delay in real-time reinforcement learning. *arXiv preprint arXiv:2503.23478*, 2025.

Thomas William Anthony. *Expert iteration*. PhD thesis, UCL (University College London), 2021.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Hendrik Baier and Mark HM Winands. Time management for monte carlo tree search. *IEEE transactions on computational intelligence and AI in games*, 8(3):301–314, 2015.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Petr Baudiš and Jean-loup Gailly. Pachi: State of the art open source go program. *Advances in computer games*, pp. 24–38, 2011.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence I. Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries P. Smit, Nathan Grinsztajn, Raphael Boige, Cemlyn N. Waters, Mohamed A. Mimouni, Ulrich A. Mbou Sob, Ruan de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in jax, 2024. URL https://arxiv.org/abs/2306.09884.

Akhilan Boopathy, Aneesh Muppidi, Peggy Yang, Abhiram Iyer, William Yue, and Ila Fiete. Permutation invariant learning with high-dimensional particle filters. *arXiv preprint arXiv:2410.22695*, 2024.

Yann Bouteiller, Simon Ramstedt, Giovanni Beltrame, Christopher Pal, and Jonathan Binas. Reinforcement learning with random delays. In *International conference on learning representations*, 2020.

Frederick Callaway, Sayan Gul, Paul M Krueger, Thomas L Griffiths, and Falk Lieder. Learning to select computations. *arXiv preprint arXiv:1711.06892*, 2017.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Stephen Chung, Ivan Anokhin, and David Krueger. Thinker: Learning to plan and act. *Advances in Neural Information Processing Systems*, 36:22896–22933, 2023.

Dylan Cope, Justin Svegliato, and Stuart Russell. Learning to Plan with Tree Search via Deep RL. In *PRL Workshop Series – Bridging the Gap Between AI Planning and Reinforcement Learning*, 2023.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*, 2022.

Thomas L Dean, Mark S Boddy, et al. An analysis of time-dependent planning. In *AAAI*, volume 88, pp. 49–54, 1988.

DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens,

Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL http://github.com/deepmind.

Esther Derman, Gal Dalal, and Shie Mannor. Acting in delayed environments with non-stationary markov policies. *arXiv preprint arXiv:2101.11992*, 2021.

Gongfan Fang, Xinyin Ma, and Xinchao Wang. Thinkless: Llm learns when to think. *arXiv preprint arXiv:2505.13379*, 2025.

Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree-structured models for deep reinforcement learning. *arXiv preprint arXiv:1710.11417*, 2017.

Johannes Forkel and Jakob Foerster. Entropy is all you need for inter-seed cross-play in hanabi. *arXiv preprint arXiv:2511.22581*, 2025.

Irving John Good. Rational decisions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 14(1):107–114, 1952.

Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Thomas L Griffiths, Frederick Callaway, Michael B Chang, Erin Grant, Paul M Krueger, and Falk Lieder. Doing more with less: meta-reasoning and meta-learning in humans and machines. *Current Opinion in Behavioral Sciences*, 29:24–30, 2019.

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. In *International conference on machine learning*, pp. 1822–1831. PMLR, 2018.

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. *arXiv preprint arXiv:2501.12948*, 2025.

Jessica B Hamrick, Andrew J Ballard, Razvan Pascanu, Oriol Vinyals, Nicolas Heess, and Peter W Battaglia. Metacontrol for adaptive imagination-based optimization. *arXiv preprint arXiv:1705.02670*, 2017.

Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Tobias Pfaff, Theophane Weber, Lars Buesing, and Peter W Battaglia. Combining q-learning and search with amortized value estimates. *arXiv preprint arXiv:1912.02807*, 2019.

Jessica B Hamrick, Abram L Friesen, Feryal Behbahani, Arthur Guez, Fabio Viola, Sims Witherspoon, Thomas Anthony, Lars Buesing, Petar Veličković, and Théophane Weber. On the role of planning in model-based deep reinforcement learning. *arXiv preprint arXiv:2011.04021*, 2020.

Nicholas Hay, Stuart Russell, David Tolpin, and Solomon Eyal Shimony. Selecting computations: Theory and applications. *arXiv preprint arXiv:1408.2048*, 2014.

Eric J Horvitz and Adrian Klein. Reasoning, metareasoning, and mathematical truth: Studies of theorem proving under limited resources. *arXiv preprint arXiv:1302.4960*, 2013.

Eric J Horvitz, Gregory F Cooper, and David E Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *IJCAI*, volume 2, pp. 1121–1127, 1989.

Shih-Chieh Huang, Remi Coulom, and Shun-Shii Lin. Time management for monte-carlo tree search applied to the game of go. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pp. 462–466. IEEE, 2010.

Konstantinos V Katsikopoulos and Sascha E Engelbrecht. Markov decision processes with delays and asynchronous cost collection. *IEEE transactions on automatic control*, 48(4):568–574, 2003.

Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. *Advances in Neural Information Processing Systems*, 36:39236–39256, 2023.

Richard E Korf. Real-time heuristic search. *Artificial intelligence*, 42(2-3):189–211, 1990.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 45716–45743, 2023.

Falk Lieder and Thomas L Griffiths. Strategy selection as rational metareasoning. *Psychological review*, 124(6):762, 2017.

Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. *Advances in neural information processing systems*, 16, 2003.

Christopher H Lin, Andrey Kolobov, Ece Kamar, and Eric Horvitz. Metareasoning for planning under uncertainty. *arXiv preprint arXiv:1505.00399*, 2015.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.

Aneesh Muppidi, Zhiyu Zhang, and Heng Yang. Fast trac: A parameter-free optimizer for lifelong reinforcement learning. *Advances in Neural Information Processing Systems*, 37:51169–51195, 2024.

Aneesh Muppidi, Katrina Brown, and Rana Shahout. Predictive scheduling for efficient inference-time reasoning in large language models. In *ES-FoMo III: 3rd Workshop on Efficient Systems for Foundation Models*, 2025.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

A Ross Otto and Nathaniel D Daw. The opportunity cost of time modulates cognitive effort. *Neuropsychologia*, 123:92–105, 2019.

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Simon Ramstedt and Chris Pal. Real-time reinforcement learning. *Advances in neural information processing systems*, 32, 2019.

Matthew Riemer, Gopeshh Subbaraj, Glen Berseth, and Irina Rish. Enabling realtime reinforcement learning at scale with staggered asynchronous inference. *arXiv preprint arXiv:2412.14355*, 2024.

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial intelligence*, 49(1-3):361–395, 1991.

Stuart J Russell. An architecture for bounded rationality. *ACM SIGART Bulletin*, 2(4):146–150, 1991.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472, 2022.

Eren Sezener and Peter Dayan. Static and dynamic values of computation in mcts. In *Conference on Uncertainty in Artificial Intelligence*, pp. 31–40. PMLR, 2020.

Yi Shen, Jian Zhang, Jieyun Huang, Shuming Shi, Wenjing Zhang, Jiangze Yan, Ning Wang, Kai Wang, Zhaoxiang Liu, and Shiguo Lian. Dast: Difficulty-adaptive slow-thinking for large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 2322–2331, 2025.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

Herbert A Simon et al. Theories of bounded rationality. *Decision and organization*, 1(1):161–176, 1972.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

David Tolpin and Solomon Shimony. Mcts based on simple regret. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pp. 570–576, 2012.

Jaden B Travnik, Kory W Mathewson, Richard S Sutton, and Patrick M Pilarski. Reactive reinforcement learning in asynchronous environments. *Frontiers in Robotics and AI*, 5:79, 2018.

Thomas J Walsh, Ali Nouri, Lihong Li, and Michael L Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18(1):83–105, 2009.

Kevin A. Wang, Jerry Xia, Stephen Chung, and Amy Greenwald. Dynamic thinker: Optimizing decision-time planning with costly compute. In *The Seventeenth Workshop on Adaptive and Learning Agents*, 2025. URL https://openreview.net/forum?id=yGglBJ1pjZ.

Ted Xiao, Eric Jang, Dmitry Kalashnikov, Sergey Levine, Julian Ibarz, Karol Hausman, and Alexander Herzog. Thinking while moving: Deep reinforcement learning with concurrent control. *arXiv preprint arXiv:2004.06089*, 2020.

Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73–73, 1996.

A Reproducibility

The full JAX implementation of every environment, base planner, and gating policy in this paper, together with pretrained checkpoints for all five environments and the two-GPU deployment harness, is released at `https://aneeshers.github.io/realtime-rl/`. Each result reported in Sections 5 and 7 has a corresponding environment-variable-driven launcher script (e.g. `eval_pacman_gating.sh`, `deploy.sh`) that reproduces the reported number from a single command using the shipped checkpoints, and the release README lists expected returns per command so any discrepancy is immediately visible. Full training hyperparameters are in Appendix I, base-checkpoint selection is in Appendix F, and the two-GPU deployment grid is in Appendix J.

B AlphaZero and MCTS

This appendix explains the planning infrastructure used in this paper. It targets readers familiar with deep RL but not with tree-search methods.

The planner used throughout this paper is Monte Carlo Tree Search (MCTS) guided by a neural network, the same combination introduced in AlphaZero (Silver et al., 2018). The intuition is that the network alone is fast but imperfect: it gives a quick estimate of which actions look promising, but it never actually checks. MCTS turns a fixed compute budget into action-quality: it spends that budget on simulated rollouts, focusing on the actions the network considers most promising while still exploring less-likely alternatives, and uses what it learns to choose a better action than the network's raw guess. More simulations produce better actions but take longer, which is the tradeoff the gating policy in the main paper learns to manage.

What does "running a simulation" mean in MCTS?

A *simulation* (or *playout*) is one traversal of the search tree from the root (the current state) down to a leaf, followed by a backup of the resulting value estimate up the tree. Each step of the traversal picks a child to descend into, balancing two pressures: prefer children that have looked good so far (high empirical value), but also try children the search has barely visited (high prior, low visit count) in case those are even better. The PUCT rule formalizes this:

$$a^* = \arg\max_a \Big[Q(s,a) + c \cdot \pi_\theta(a|s) \cdot \frac{\sqrt{N(s)}}{1+N(s,a)} \Big],$$

where $Q(s,a)$ is the empirical mean value of subtree a, $\pi_\theta(a|s)$ is the network's prior over actions, $N(s)$ is the total visit count of node s, and $N(s,a)$ is the visit count of edge (s,a). Repeated simulations concentrate visits on high-value, high-prior subtrees while continuing to explore less-visited ones. After n simulations the recommended action is the *most-visited* child of the root.

How is the final action selected?

At evaluation time the agent picks the most-visited root child. During training, the visit-count distribution $\hat{\pi}(a) \propto N(s_0,a)^{1/\tau}$ (temperature $\tau > 0$) is used as a *search target*: the network is updated to predict $\hat{\pi}$ at state s_0. Lower τ concentrates probability on the most-visited action; higher τ spreads probability across alternatives, useful for exploration early in training. This setup (search produces a better-than-raw-policy distribution, and the network learns to imitate it) is called *expert iteration* (Anthony, 2021), and over many iterations it tightens the network's prior, making future searches better-targeted.

What is the difference between AlphaZero and MuZero?

AlphaZero Silver et al. (2018) requires a *perfect simulator*: given state s and action a, the simulator returns the exact next state s'. MCTS runs entirely inside this simulator; the network supplies priors and value estimates but never models transitions.

MuZero Schrittwieser et al. (2020) replaces the perfect simulator with a *learned latent-space dynamics model*. Transitions during search are predicted by a network rather than a ground-truth engine, so MuZero can plan in environments without a simulator (e.g. Atari frames).

All environments in this paper expose a JAX-native step function, so we use the AlphaZero paradigm. This choice is also what enables our committed-action protocol (Section 3.3): the search tree explicitly rolls forward the $k-1$ committed steps to reach the future state where the planned action will land, which is only possible when the simulator is perfect.

AlphaZero was designed for two-player zero-sum games. How does it transfer to single-agent settings?

The core loop is *expert iteration* Anthony (2021): at each iteration the agent uses MCTS to produce an expert distribution over actions (better than the network's raw policy because it incorporates lookahead), and the network is trained to imitate that expert. Over many iterations, the network's prior gets stronger, which makes future searches better-targeted. The value head uses a bootstrapped return instead of a zero-sum game outcome; otherwise the algorithm is identical to two-player AlphaZero. Algorithm 1 formalizes this.

Algorithm 1 Single-agent AlphaZero via Expert Iteration

Require: Network $f_\theta = (\pi_\theta, v_\theta)$, environment $\mathcal{E}$, simulations per step n, replay buffer $\mathcal{B}$

repeat
  $\mathcal{D} \leftarrow \{\}$ ▷ episode data buffer
  $s_0 \leftarrow \mathcal{E}.\text{reset}()$
  for $t = 0, 1, \ldots, T-1$ **do** ▷ **Act**
    Run n MCTS simulations from s_t using f_θ
    $\hat{\pi}_t(a) \propto N(s_t, a)^{1/\tau}$ ▷ visit-count policy
    $a_t \sim \hat{\pi}_t; \quad s_{t+1} \leftarrow \mathcal{E}.\text{step}(a_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t,\, \hat{\pi}_t)\}$
  end for
  Compute returns $z_t = \sum_{k \geq 0} \gamma^k r_{t+k}$ for each t ▷ **Store**
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{(s_t, \hat{\pi}_t, z_t)\}_{t=0}^{T-1}$
  Sample mini-batch from $\mathcal{B}$ ▷ **Learn**
  $\mathcal{L}(\theta) = \dfrac{1}{|\mathcal{B}|} \displaystyle\sum_{(s,\hat{\pi},z) \in \mathcal{B}} \Big[\big(v_\theta(s) - z\big)^2 - \hat{\pi}^\top \log \pi_\theta(s) \Big]$
  $\theta \leftarrow \theta - \alpha \, \nabla_\theta \, \mathcal{L}(\theta)$
until convergence

How does MCTS scale on modern accelerators?

A common worry is that MCTS is inherently sequential, each simulation updates the Q and N tables that the next simulation reads. In practice, two layers of parallelism make this much less of a bottleneck than it appears.

Environment-level parallelism. We maintain a batch of E independent environment instances (32 in our experiments). Each instance has its own search tree; all E leaf nodes are evaluated in a *single batched forward pass* of the neural network, which is the GPU/TPU bottleneck.

Intra-tree simulation loop via `jax.lax.scan`. Within each tree the n simulations are unrolled as a `scan`. Each iteration: (1) traverse all E trees to their current leaf using PUCT (parallelised over E), (2) batch-evaluate the E leaves in one network call, (3) back-propagate values. Because JAX traces the full scan at compile time, the tree-update logic is fused into a single XLA kernel with zero Python-level iteration overhead at runtime.

The result is that all n simulations across all E environments are compiled into a single `jit`-ted function that runs entirely on the accelerator, with one network forward pass per simulation step and no host-device transfers during rollout.

C Variable-duration GAE

We re-introduce the meta-step subscript k_t in this appendix because the derivation reasons about a sequence of meta-decisions rather than a single budget choice.

Standard GAE (Schulman et al., 2015) estimates the advantage at timestep t as a λ-weighted sum of one-step TD residuals,

$$\hat{A}_t \;=\; \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}, \qquad \delta_t \;=\; r_t + \gamma\, V(s_{t+1}) - V(s_t), \tag{6}$$

which assumes a unit time gap between consecutive states. In our SMDP, consecutive meta-states s_t and s_{t+k_t} are separated by a variable number of environment frames k_t, so the per-step discount in the TD residual becomes γ^{k_t}:

$$\delta_t \;=\; R_t + \gamma^{k_t}\, V(s_{t+k_t}) - V(s_t), \qquad R_t \;=\; \sum_{j=0}^{k_t-1} \gamma^j \, r_{t+j}. \tag{7}$$

The discount accumulated between meta-steps t and $t+l$ is $\gamma^{\sum_{j=0}^{l-1} k_{t+j}}$ rather than γ^l, giving the meta-step advantage

$$\hat{A}_t \;=\; \sum_{l=0}^{\infty} \lambda^l \left(\prod_{j=0}^{l-1} \gamma^{k_{t+j}} \right) \delta_{t+l}, \tag{8}$$

or, recursively,

$$\hat{A}_t \;=\; \delta_t + \gamma^{k_t} \lambda \, \hat{A}_{t+1}. \tag{9}$$

The structure is identical to standard GAE; the only change is that the per-step discount is γ^{k_t} rather than γ. We compute these advantages in a single backward pass over each rollout using equation (9).

Without this correction, treating each meta-step as a unit time gap would systematically understate the discount on long-budget meta-steps, biasing the value function in favor of many short budgets over fewer long ones. The γ^{k_t} correction makes the value function comparable across budget choices.

D Environment Details

D.1 Real-time Tetris

Our real-time Tetris environment is a modification of Jumanji Tetris in which each environment step is one gravity tick rather than one full placement decision. The board is 20×10, episodes last at most 2000 gravity ticks, and the observation contains the locked board, the current falling tetromino, its (x, y) position, and the gravity-tick count.

Every call to `env.step(action)` first applies the chosen control and then applies one gravity update. The action set has six elements:

- **Left / Right**: translate the falling piece one column in the respective direction; invalid lateral moves are ignored.
- **Rotate-CW / Rotate-CCW**: rotate the piece 90ř clockwise or counter-clockwise; invalid rotations are ignored.
- **Hard-drop**: instantly translate the piece to the lowest valid row and lock it; a new piece spawns immediately.
- **Noop**: no horizontal or rotational change; the piece descends one row by gravity.

A piece also locks whenever gravity would move it into an occupied cell; completed rows are then cleared and scored with the standard convex Tetris reward schedule. The episode ends on board

top-out, i.e. when a newly spawned piece cannot be placed at the spawn position, or when the 2000-tick horizon is reached.

For the committed-action experiments in the main paper we use the `TetrisRTKT` variant. The underlying environment dynamics are identical, but during the $k-1$ delay steps inside MCTS the tree rolls out policy-guided committed actions rather than assuming a pure gravity-only noop sequence. This makes the search-time delay model match the deployment-time reflex behavior more closely.

D.2 Committed-action execution

During the $k-1$ delay steps while the MCTS planner is running, the agent executes committed actions in the environment. In all variants these are drawn from the argmax of the base model's raw policy logits—a single inexpensive forward pass, with no additional MCTS compute. The two variants differ only in what the *MCTS tree simulates* during its internal rollout of the delay window, not in what is executed in the environment. We refer to these internal simulator variants as RT_KStep (noop) and RT_KT (logit).

RT_KStep (noop) simulates the delay steps inside the tree using a fixed action (gravity/noop), which is fast but mismatches what the agent will actually do.

RT_KT (logit) simulates the delay steps inside the tree using $\text{argmax}(\pi_{\text{logits}})$, matching the committed actions the agent executes at deployment and making the search model more accurate at no extra environment cost. All real-time Tetris results reported in the main paper use the RT_KT variant.

Crucially, neither variant uses MCTS for the committed actions executed in the environment, so there is no compute-parity violation: every option $k \in \mathcal{K}$ uses the same number of policy forward passes for its delay steps.

D.3 Speed Hex

Speed Hex is built on the pgx Hex library (11×11 board). Each player begins with $T = 300$ clock ticks; time spent planning is deducted after each move. MCTS search uses board-only dynamics (no clock deduction during simulation), but the observation (and therefore the network's value estimates) includes the remaining clock for both players. The gating policy is a GRU network that maintains hidden state across moves within an episode, allowing it to track game progression and clock pressure jointly.

Training notes. Early experiments fine-tuned the gating policy against fixed-budget opponents rather than via self-play. This regime exhibited plasticity loss, the network's ability to adapt to new opponents degraded over successive fine-tuning rounds, consistent with findings in continual RL settings (Muppidi et al., 2024; Boopathy et al., 2024). We therefore switched to self-play, which eliminated plasticity loss entirely. In self-play we found that increasing entropy regularization was the single most impactful training intervention, consistent with recent results in other self-play board-game domains (Forkel & Foerster, 2025).

D.4 Speed Go

Speed Go is built on the pgx Go library (9×9 board). As in Speed Hex, each player has an explicit clock, planning consumes clock only after the move is played, and MCTS search itself uses board-only dynamics while the network observes the remaining clock for both players. We use the same recurrent gating architecture and evaluate the policy over the same shared five sampled clock budgets used in the main text. The same training observations apply: self-play with elevated entropy regularization was decisive, and no plasticity loss was observed once fine-tuning against fixed opponents was abandoned.

E Simulation Option Calibration for Clock Environments

For committed-action environments, the equal-budget constraint (32 sims/frame) naturally produces meaningful option spacing. For clock environments this constraint does not apply, so we select simulation options by examining two curves simultaneously: planning quality (win rate or return) vs. simulation count, and inference latency vs. simulation count.

An option is *useful* if it is distinguishable from its neighbors on both dimensions: adding more simulations should noticeably improve play *and* noticeably increase latency. Options that are close in both cost and quality give the gating policy nothing to learn; options that differ in cost but not quality waste planning budget.

For Speed Hex we found that options $\{2, 8, 32, 128\}$ satisfy this criterion across the relevant range of clock settings, and for Speed Go we found that $\{16, 32, 64, 96\}$ yield similarly distinct quality-latency tradeoffs. Figure 9 shows the full co-scaling results across all five evaluation environments.

Go and Hex calibration. To calibrate the clock-environment options more directly, we examine how average expected score changes as inference budget increases. The resulting curves are shown in Figure 10. For Speed Go, the selected set $\{16, 32, 64, 96\}$ spans the steep part of the quality curve: average expected score versus all other budgets rises from 0.314 at 16 simulations to 0.506 at 32, 0.704 at 64, and 0.782 at 96. The largest gains occur at 16→32 (+0.193) and 32→64 (+0.198), while 64→96 still provides a clear additional improvement (+0.078). Pairwise tournament results among the selected Go options also remain well separated: 64 simulations scores 0.713 against 32 and 0.868 against 16, while 96 scores 0.595 against 64 and 0.773 against 32.

For Speed Hex, the selected set $\{2, 8, 32, 128\}$ follows the same design principle and is now supported by the completed inference-budget tournament. Average expected score versus all other budgets rises from 0.340 at 2 simulations to 0.422 at 8, 0.495 at 32, and 0.710 at 128, with successive gains of +0.082, +0.073, and +0.214. The pairwise Hex tournament shows the same separation: 8 simulations scores 0.550 against 2, 32 scores 0.564 against 8 and 0.643 against 2, and 128 scores 0.693 against 32 and 0.727 against 8. Thus, in both clocked domains, the selected options provide the gating policy with meaningfully separated quality-latency tradeoffs.

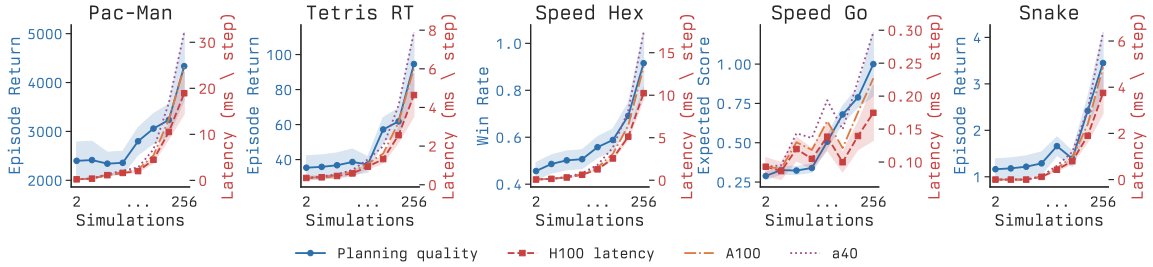

Figure 9: Full co-scaling results across Pac-Man, real-time Tetris, Speed Hex, Speed Go, and Snake. Blue denotes planning quality; dashed curves denote per-step latency on H100, A100, and A40; shading shows ±SE.

F Cross-Evaluation and Base Model Selection

For the committed-action environments, we trained four AlphaZero checkpoints at budgets $k \in \mathcal{K}$ and evaluated all 16 (train-k, eval-k) combinations. Here, train-k denotes the action-delay budget used during training, while eval-k denotes the action-delay budget used at test time. For each train-k checkpoint we additionally swept the MCTS simulation budget at evaluation time and report the best raw episode return achieved for each eval-k. The evaluation sweeps were: real-time Tetris with simulation counts $\{32, 64, 96, 128\}$, Pac-Man with $\{8, 16, 32, 64, 96, 128\}$, and Snake with $\{32, 64, 96, 128\}$.

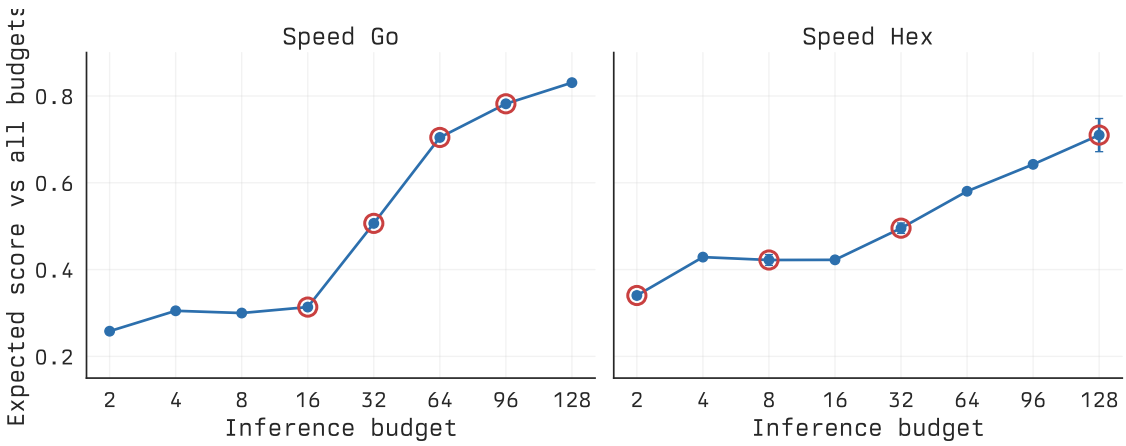

Figure 10: Simulation-option calibration for the two clock environments. Each point shows a budget's average expected score against all other candidate budgets. Red circles mark the options used in our clocked experiments: 16/32/64/96 simulations for Speed Go and 2/8/32/128 for Speed Hex.

Environment	Train-k	Eval-k=1	Eval-k=2	Eval-k=3	Eval-k=4
real-time Tetris	k=1	61.8 @ 128	**85.0 @ 128**	**73.2 @ 128**	**69.0 @ 128**
	k=2	30.0 @ 96	11.4 @ 128	7.4 @ 96	3.6 @ 96
	k=3	**72.2 @ 128**	22.0 @ 128	15.2 @ 128	11.8 @ 128
	k=4	69.6 @ 128	24.0 @ 96	23.2 @ 128	18.8 @ 128
Pac-Man	k=1	**3235.4 @ 128**	925.3 @ 128	**2579.3 @ 128**	**1869.2 @ 96**
	k=2	2458.5 @ 128	630.7 @ 128	866.7 @ 128	574.4 @ 96
	k=3	2749.7 @ 128	1122.3 @ 128	1110.6 @ 64	1051.2 @ 128
	k=4	2904.1 @ 128	**1161.2 @ 128**	1117.2 @ 128	1132.4 @ 96
Snake	k=1	0.77 @ 128	0.33 @ 128	0.50 @ 128	0.27 @ 128
	k=2	2.32 @ 128	0.56 @ 128	0.84 @ 128	0.31 @ 96
	k=3	**2.42 @ 128**	**0.80 @ 64**	**1.13 @ 128**	**0.49 @ 96**
	k=4	0.05 @ 32	0.05 @ 128	0.06 @ 128	0.07 @ 128

Table 1: Cross-evaluation of committed-action base models. Entries report the best raw episode return for each train-k/eval-k combination after optimizing over the evaluation simulation-budget sweep; the selected simulation count is shown after the '@' symbol. Bold marks the best train-k for each eval-k within an environment.

The resulting pattern is environment-dependent but clear. For real-time Tetris, k=1 is the strongest base model overall: although the k=3 checkpoint is best at eval-k=1, the k=1 checkpoint wins at eval-$k \in \{2, 3, 4\}$ and has the highest mean return when averaging the best-per-eval-k scores. For Pac-Man, k=1 also wins overall, dominating eval-$k \in \{1, 3, 4\}$ and achieving the highest average performance across eval budgets, while k=4 is only slightly better at eval-k=2. Snake differs qualitatively: the k=3 checkpoint is the strongest across all four eval budgets and therefore serves as the frozen base model in that domain.

These results match the intuition that low-delay training is often causally cleaner, since less delay-induced mismatch enters the value targets. In addition, committed actions during delay steps are drawn from the base model's policy, so a stronger base policy also produces better intermediate states before the final MCTS action lands. Accordingly, we use the k=1 checkpoint as the frozen base for Pac-Man and real-time Tetris, and the k=3 checkpoint for Snake.

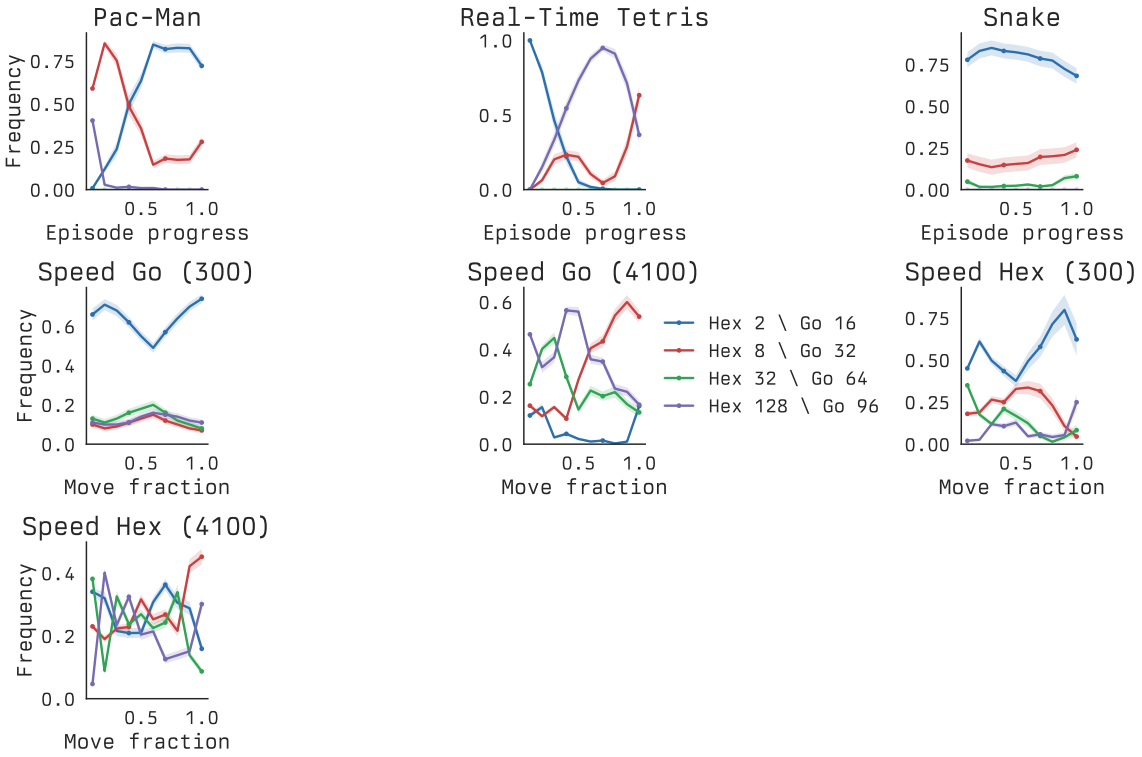

Figure 11: Different environments induce different allocation profiles. Pac-Man becomes more reactive later in the episode, real-time Tetris shifts toward deeper planning as the board densifies, Snake remains mostly reactive with occasional deeper planning in constrained states, and both clock games become less reactive when more time is available. This figure complements Figure 6 by showing the full cross-environment strategy profiles rather than only the clock-game comparison.

G Strategy Profiles Across Environments

This section expands the main-text analysis by showing the full learned budget-allocation profiles for each environment (Figure 11).

H Strict-Timeout Speed Hex Control

The main Speed Hex setup allows the game to continue after the acting side exhausts its remaining clock budget, so the experiment measures *resource allocation* rather than a pure timeout-avoidance game. To verify that this distinction matters, we ran a strict-timeout control in which overspending the selected search budget causes an immediate loss and there is no fallback behavior.

Figure 12 shows the resulting per-budget head-to-head curves under the unique-game metric used in the main paper, evaluated on a wider clock sweep than the main benchmark. The learned gate stays above parity against every opponent *on average*, and against five of the eight opponents it is above parity at *every* evaluated budget. Even the hardest fixed-budget baseline under this metric, always-32, averages only 0.690 against the gate, while policy-only averages 0.952 and random allocation averages 0.904. The only sub-parity points occur for the more clock-aggressive baselines at the largest budgets: fixed 128-simulation play dips to 0.490 at T=5700, midpeak dips to 0.491 at T=4100, and greedy dips to 0.487 at T=5700. This pattern is the opposite of what we observe in the main benchmark. Once timeout is an immediate terminal event, stronger search is no longer reliably beneficial; it often just increases the probability of burning too much clock. In other words, the game becomes easier

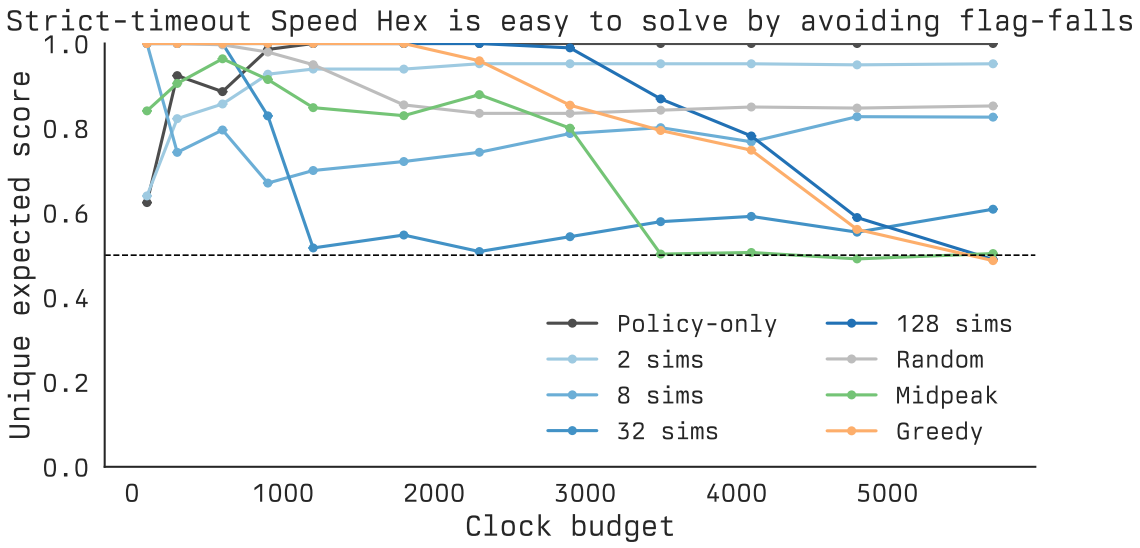

Figure 12: Strict-timeout Speed Hex is substantially easier than the main Speed Hex benchmark. Using the unique-game expected-score metric, the learned gate remains above parity against every opponent on average, with means of 0.952 against the policy-only opponent, 0.903 against fixed 2-simulation play, 0.904 against random allocation, and 0.893 even against the fixed 128-simulation opponent. At large budgets, the more aggressive opponents become the weakest because they overspend clock and lose on time.

because the gating policy can win largely by learning a conservative time-management rule rather than by solving the underlying Hex positions more effectively.

I Implementation Details

I.1 Gating network architecture

Spatial branch. The game grid is processed by a 1×1 convolution lifting to 64 channels, followed by three residual blocks with Layer Normalization and global average pooling to a 64-dimensional spatial representation. Layer Normalization is used because the gating policy operates on a single environment instance per rollout step, making batch statistics undefined.

Time / clock embedding. For committed-action environments, the normalized step count is encoded as a 2-vector and projected through a small MLP before being added to the spatial features. For clock environments, the remaining clock fraction is similarly embedded and added.

Fusion and heads. The spatial representation (64-dim), planner trunk features (128-dim), and planner value (1-dim) are concatenated and passed through dual MLP heads: a policy head producing logits over k, and a value head producing a scalar baseline.

Speed Hex gating network. Uses a GRU backbone rather than a feedforward network, maintaining a recurrent hidden state across moves within each game to track temporal context.

I.2 Training hyperparameters

Clock-environment training details. The clock-game training runs use a separate self-play stack built on `pgx`. For Speed Go, the base AlphaZero planner is trained by pure self-play on 1024 parallel games with 32 MCTS simulations per move, a 256-move horizon, training batch size 4096, Adam with learning rate 10^{-3}, and 800 training iterations; evaluation against the `pgx` baseline occurs every 5 iterations and checkpoints are saved every 200 iterations. The recurrent Speed Go gate then loads the frozen 16-simulation base checkpoint matched to the same seed

Table 2: PPO hyperparameters for committed-action environments.

Hyperparameter	Pac-Man	real-time Tetris	Snake
Num. parallel envs	32	32	32
Meta-steps per rollout	384	384	384
PPO epochs per rollout	4	4	4
Num. mini-batches	16	16	16
Meta-reward mode	raw	raw	raw
Discount γ	0.99	0.99	0.997
GAE λ	0.95	0.95	0.95
PPO clip ϵ	0.2	0.2	0.2
Entropy coefficient	0.01	0.01	0.05
Learning rate	3e-4	3e-4	3e-4
Sim options	32, 64, 96, 128	32, 64, 96, 128	32, 64, 96, 128

and is trained in pure self-play for 500 PPO updates on 256 parallel environments. Each PPO update collects 32768 rollout steps (128 per environment), chunks them into GRU sequences of length 64, and uses 4 PPO epochs with batch size 512, γ=0.99, λ=0.95, PPO clip ϵ=0.2, learning rate 3×10^{-4}, value-loss coefficient 0.5, entropy coefficient 0.01, and global gradient clipping at 1.0. The gate chooses among simulation options $\{16, 32, 64, 96\}$, domain-randomizes episode clocks over $\{100, 300, 600, 900, 1200, 1800, 2300, 2900, 3500, 4100, 4800, 5700\}$, and saves checkpoints every 25 updates. Although the codebase contains hooks for richer opponent schedules, the experiments reported here use pure self-play throughout.

I.3 JAX implementation and batching strategy

The main computational challenge is that our system nests a meta-policy on top of an AlphaZero-style planner, so a naive implementation would repeatedly invoke MCTS inside an outer RL loop with substantial Python overhead. Our implementation avoids this by pushing nearly all control flow into JAX primitives.

Base planner training. The frozen planners are trained with our own Gumbel-AlphaZero stack built on top of the Jumanji environments, using `mctx.gumbel_muzero_policy` (DeepMind et al., 2020) for search. Within each planner step, the MCTS simulations are executed inside a compiled `lax.scan`; leaf expansion and backup are batched across parallel environments with `vmap`; and the full self-play/training pipeline is wrapped in `jit`. In the multi-device setting, the leading batch dimension is sharded with `pmap`, so each accelerator runs the same compiled self-play/update program on its local slice of environments.

Gating-policy training. For the gating layer, we again compile whole rollouts rather than individual steps. Each PPO rollout is one `jit`-compiled `lax.scan` over meta-decisions, and advantage computation is another reverse `lax.scan`. The recurrent Speed Hex gate uses the same pattern: the GRU unroll is expressed as a scan, while PPO operates on sequence chunks rather than on Python loops over timesteps.

Evaluating all budget options in parallel. The key batching trick is that, during gating training and evaluation, we compute the transition for every available budget option at each meta-step and only then select the branch corresponding to the gate's chosen action. Concretely, for $k \in \{1, 2, 3, 4\}$ we run all four `meta_step_one_k` branches, stack the resulting rewards, discounts, next states, and next observations, and use a batched selection operator to pick the chosen branch per environment. This spends extra compute relative to evaluating only the selected option, but it preserves static shapes, keeps the whole computation batched, and avoids recompilation or Python-side control flow. In practice this systems tradeoff is what makes the meta-MCTS stack fast enough to train routinely rather than as a one-off systems effort.

Practical effect. These engineering choices — `mctx` (DeepMind et al., 2020) for search, `vmap` for environment parallelism, `jit` for whole-program compilation, `pmap` for device parallelism, and `lax.scan` for both MCTS inner loops and PPO rollout loops — reduce wall-clock training time dramatically. In our internal runs, the end-to-end training workflow dropped from roughly two weeks in an earlier less-batched pipeline to roughly six hours in the optimized JAX implementation. The full implementation is available at https://aneeshers.github.io/realtime-rl/.

J Detailed Two-GPU Deployment Breakdown

We ran the full asynchronous deployment grid for all 45 settings: 3 environments (real-time Tetris, Pac-Man, Snake) × 3 GPUs (H100, A100, A40) × 5 frame rates (8–12 FPS). Figure 13 gives the complete breakdown by environment, GPU, and FPS for return, deadline miss rate, and p95 slack to deadline.

Several patterns are clear. First, H100 is consistently reliable across the entire tested range. In real-time Tetris its miss rate stays at 0.014% for all FPS values, with p95 slack remaining positive from +291.0 ms at 8 FPS to +123.6 ms at 12 FPS. Snake is even easier: H100 and A100 stay at 0.001% miss rate throughout, and A40 stays at or below 0.005%.

Second, the deployment boundary appears where slack collapses. Real-time Tetris on A40 is the clearest example: p95 slack decreases monotonically from +162.2 ms (8 FPS) to −4.1 ms (12 FPS), and the miss rate jumps from 0.046% to 50.669%. By contrast, real-time Tetris on A100 remains usable even at 12 FPS, with p95 slack still positive at +24.5 ms and miss rate only 0.188%.

Third, Pac-Man is the most timing-sensitive environment. Even with positive p95 slack throughout, A100 and A40 exhibit non-trivial miss rates across all FPS values. At 12 FPS, Pac-Man reaches 30.369% misses on A100 and 13.073% on A40, whereas H100 stays much lower at 3.927%. This indicates that for Pac-Man, the far latency tail beyond the p95 threshold matters operationally even when the central mass remains in-budget.

Finally, return remains comparatively stable over a broad deployment range despite these timing differences. Real-time Tetris returns remain in the 41–50 range on H100/A100 and in the high-30s to low-40s on A40 except at the most constrained settings. Pac-Man degrades more gradually with hardware and FPS than its miss-rate curve might suggest, while Snake is nearly flat across all tested settings. Together these results support the main-text conclusion: the committed-action training protocol transfers robustly to asynchronous hardware deployment, and the observed failures emerge in predictable deadline-limited regimes rather than as qualitatively new behaviors.

K Additional Results and Ablations

We also ran input-ablation evaluations for the committed-action environments, in which the gating policy received zeroed observation features, zeroed time features, zeroed planner trunk features, or a zeroed planner value input while the environment dynamics and MCTS planner remained unchanged. Across environments, ablating planner trunk features was consistently disruptive, while zeroing the scalar value input had only minor effect; in real-time Tetris, zeroing either the board observation or planner trunk substantially reduced return, while in Pac-Man and Snake the learned behavior was more robust to removing any single input channel.

Environment	Baseline	Zero obs	Zero time	Zero trunk	Zero value
Pac-Man	2245.6	2379.2	1571.2	1812.6	2189.8
real-time Tetris	44.0	34.4	37.6	33.2	44.0
Snake	15.18	14.92	15.94	15.66	16.14

Table 3: Input-ablation results for the committed-action gating policies. Entries report mean episode return under each ablation condition while the environment and MCTS planner remain unchanged.

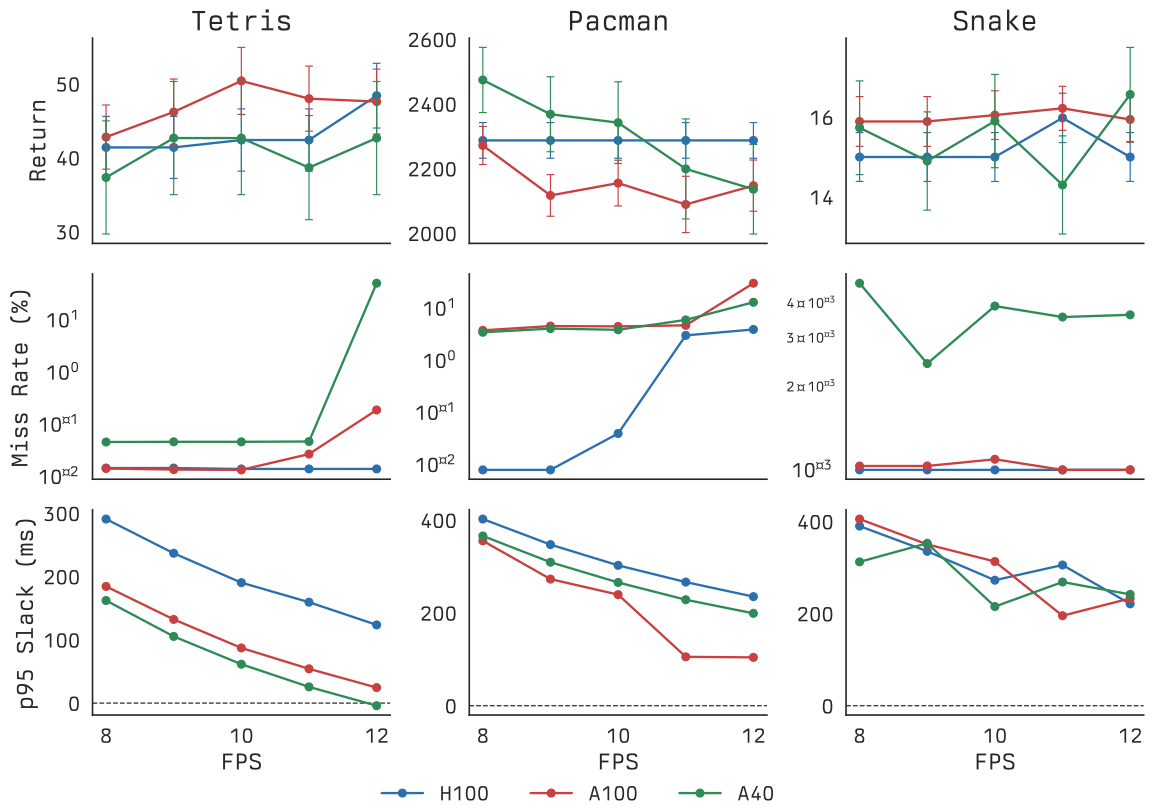

Figure 13: Detailed two-GPU deployment breakdown across real-time Tetris, Pac-Man, and Snake. Top: return vs. FPS. Middle: deadline miss rate. Bottom: p95 slack to the k=4 deadline. Snake remains robust, real-time Tetris fails only near the tightest A40 regime, and Pac-Man is the most latency-sensitive.

L Real-Time RL, SMDPs, and Committed-Action Training

This appendix expands Section 3.2 by focusing on the distinction between the discrete-time SMDP used in training and the continuous-time SMDP approximated in deployment.

The RTRL framework and its single-step restriction

Section 3.2 already gives the RTMDP equation and explains why Ramstedt et al.'s fixed one-step delay formalism is not the natural representation for our variable-budget committed-action setting. The additional point needed here is that, once delay is represented as budgeted options, training and deployment instantiate two closely related SMDPs with different time models.

Training is a discrete SMDP; deployment is a genuine SMDP

Simulation. In training the environment is synchronous: MCTS runs in zero simulated time. As described in Section 3.2, each meta-step proceeds by choosing k, executing $k-1$ committed argmax steps, and then applying the MCTS result. From the meta-policy's perspective this is a *Semi-Markov Decision Process* (Puterman, 1994) with *deterministic* holding time $\tau = k$, so successive meta-decisions are spaced k environment steps apart and the appropriate discount is γ^k. This is the discrete-time SMDP solved during training.

Deployment. In real-time deployment the holding time between meta-decisions equals $k \times T_{\text{frame}}$: MCTS runs concurrently on GPU 1 while the environment executes the k reflex frames on GPU 0,

so the two durations *overlap* rather than sum. Given k, the elapsed wall-clock time is $k \times T_{\text{frame}}$ regardless of MCTS latency, provided search completes within the deadline. The holding time is stochastic only because k is chosen stochastically by π_{gate}; this is a true SMDP in the continuous-time sense (Puterman, 1994). Our measurements confirm that MCTS latency stays well within the frame budget[2] (H100: mean 122 ms, p95 205 ms; A40: mean 197 ms, p95 338 ms vs. a k=4 frame budget of 444 ms at 9 FPS), so the deterministic simulation discount γ^k matches the deployment discount closely, which is why sim-to-real transfer succeeds without retraining.

How committed-action training avoids state augmentation

The central reason the k-step RTMDP state augmentation is unnecessary in our framework is that *the committed action at each intermediate step is computed from the current state, not from a state k steps ago*. Specifically, the committed action at frame t is $a_t = \arg\max_a \pi_\theta(a \mid s_t)$, evaluated fresh in $\sim$2 ms—well within the 111 ms frame budget. The MCTS search is initiated at s_0 (start of meta-step), but the tree explicitly rolls forward the same $k-1$ committed steps that will be executed in the environment and selects its terminal action for the resulting future landing state. This delay is therefore explicitly modeled in training: the $k-1$ committed steps actually execute in the environment before the MCTS action lands, so the policy learns to handle the k-step gap without any state augmentation.

In the RTRL vocabulary, the committed-action component operates in the *turn-based* regime (computation time $\ll$ frame time, so the environment effectively pauses during action selection), while the MCTS component operates in a k-step delay regime whose effects are absorbed by the training curriculum rather than by augmenting the state space. The key property that makes this possible—and that the standard RTMDP does not exploit—is that the intermediate policy π_{reflex} is a *deterministic function of the current state*, so its actions do not need to be carried in the state to recover the Markov property.

[2]Note the distinction between *frame budget* and *planning budget* k. The frame budget is the time between frames at the chosen FPS (e.g., 111,ms at 9,FPS)