ChatGPT

A) System Understanding

- **Control Loop & Environment Steps:** The RL agent (Tamiyo) interacts with an environment that simulates the training of a host model over epochs. Each **environment step corresponds to completing one or more training epochs** of the host network, during which seeds may be introduced, trained, blended, or removed. At the end of each step (epoch), the agent receives an observation summarizing training metrics and seed states, then selects an action for the next interval [1] [2]. The agent's **policy acts at the granularity of epochs** – e.g. an action can direct the system to germinate a new seed or adjust a blending parameter before the next set of epochs runs.
- **Policy Controls:** The policy controls a **factored action space** with multiple categorical "heads". In the current implementation, it chooses a target slot, a seed blueprint type, a blending algorithm, and a lifecycle operation [3] [4]. (Planned extensions add heads for blend tempo, alpha target values, etc.) The operation (`op`) can be **WAIT, GERMINATE, PRUNE (CULL), FOSSILIZE,** or advanced parameter tweaks like setting a seed's alpha target or forcing stage advancement (if enabled). For example, a GERMINATE action spawns a new seed of the chosen blueprint in the selected slot (if empty), and a FOSSILIZE action attempts to finalize integration of a blended seed. The agent thus **controls when to add seeds, which type to add, how to blend them, and when to remove or finalize them**.
- **Environment Transitions:** Each action is applied and then the host training proceeds for some epochs (the "tempo" head, if present, could specify 3/5/8 epochs to advance before next decision). This defines the state transition. **Observations** include global training progress (loss, accuracy, etc.) and per-slot seed status [1] [2]. Notably, the observation is a flat vector of base features (e.g. current epoch, recent loss/accuracy trends, etc.) plus per-slot features (seed stage, blend alpha, improvement, blueprint type one-hot) [5] [2]. Optionally, detailed **seed telemetry** (like gradient norms, health scores, time in stage) can be appended to each slot's features [6] [7]. All observations are normalized online for stability (using running mean/std) [8] [9], ensuring inputs to the policy network remain roughly zero-mean, unit-variance.
- **Action Masking & Fallback:** The environment applies **action masks** to disallow physically impossible moves – e.g. it masks germination if a slot is occupied, masks fossilize if no seed is ready, masks blueprint choices that aren't valid for the model topology (CNN vs Transformer) [10] [11]. The policy still outputs a probability for each categorical choice, but masked options are given near-zero probability (large negative logits) so they won't be selected [4]. If the agent somehow produces an illegal combination (due to function approximation error or a masking bug), the environment simply **ignores the invalid part and treats the action as a no-op (WAIT)** [12] [13]. This silent failure means the agent effectively loses a turn with no progress. Such fallbacks prevent crashes but can **poison learning signals** – the policy might not get a clear negative reward for an invalid action, just the default outcome of doing nothing, making it harder to learn the correct constraints.
- **Episode Horizon & Non-Stationarity:** An episode typically spans an entire training run of the host model (e.g. 25 epochs by default [14]). This is the true credit assignment horizon – an action early in training (like germinating a seed) may only yield rewards (e.g. improved accuracy) many steps later when the seed blends in. The environment is highly **non-stationary**: the underlying supervised model is learning during the episode, so the state dynamics change over time. For instance, the same action (adding a seed) can have different effects depending on the epoch (early vs late training) and the current host model state. Additionally, each new seed changes the state space (adding parameters, altering loss/accuracy trajectory), violating the i.i.d. assumption underlying stationarity. The **policy must handle long-term dependencies** (hence an LSTM

memory is used [14]) and a shifting reward landscape as the episode progresses. Key contributors to non-stationarity include the host's improving accuracy (which makes incremental gains harder later on), the changing number of active seeds, and distributional shifts in observations (e.g. loss deltas shrink as the model converges). This complexity means **credit assignment is challenging** – the agent relies on reward shaping and memory to connect actions with eventual outcomes.

B) Likely Failure Modes and Why (with Symptoms & Mitigations)

1. **Reward Hacking via PBRS:** The agent may learn to game the shaped reward signals (Potential-Based Reward Shaping) instead of truly improving model performance. For example, it might rush seeds through lifecycle stages to collect shaping bonuses – **"fossilization farming"** – without actually yielding a better model [15]. *Symptoms:* The policy frequently germinates and immediately blends/finalizes seeds that contribute little to accuracy (e.g. many seeds with minimal improvement, just enough to pass thresholds). Episode rewards remain high (from stage bonuses) even if final validation accuracy stalls. You may see many seeds reaching FOSSILIZED with exactly the minimum required contribution (e.g. 0.5%) repeatedly. *Telemetry:* Log the breakdown of reward components per episode – if **stage potential rewards dominate** while actual accuracy gain is flat, that's a red flag. Also monitor the number of seeds fossilized with low improvement. *Mitigations:* Increase the **minimum improvement threshold** for fossilization (e.g. >0.5% [16]) or raise the penalty for fossilizing unproductive seeds. Reduce the weight of PBRS shaping relative to the main objective, especially toward the end of training (see reward scheduling in section E). Introduce a **counterfactual check**: only give the final fossilization bonus if the seed's removal would significantly hurt accuracy (ensuring the reward aligns with true contribution). These ensure the agent only gets big rewards for genuinely helpful seeds.

2. **Inactivity (Always WAIT Policy):** The agent might learn to do nothing to avoid penalties or risk. If the shaped reward is mis-tuned – e.g. if adding a seed often causes a transient drop in accuracy (and hence negative reward) or incurs a rent penalty – the safest strategy could be to never germinate at all. *Symptoms:* The agent predominantly selects the WAIT action. Few or zero seeds are ever germinated in an episode (seed utilization stays near 0 [17]). Training performance then mirrors the baseline (no seeds) behavior. Episode returns might be low but steady, with the agent avoiding both positive and negative extremes. *Telemetry:* Track the distribution of `op` actions – if **WAIT commands >> others**, especially early in training, that's a problem. Also monitor the "compute_rent" penalty component: if the rent cost (parameter overhead) is large relative to potential gains, the agent may decide seeds aren't worth it. *Mitigations:* Lower the **compute rent penalty** weight so that adding a seed isn't immediately "too expensive" [18] [19]. Provide an initial positive reward for germination (e.g. via PBRS +1 for GERMINATED [20]) so that the agent at least tries. Curriculum can help: start with scenarios where seeds have a guaranteed benefit (to build trust in germination) before introducing full penalties. Also ensure the **initial host model** plateaus so that doing nothing yields near-zero reward – the agent should see that *some* intervention is needed to get higher returns.

3. **Entropy Collapse in Certain Action Heads:** The multi-head policy might prematurely become deterministic in some parts of its action (losing exploration). For example, the slot head might always pick slot 0, or the blueprint head might collapse to always selecting a particular blueprint type, even if suboptimal. This often happens if one choice yields slightly better early rewards and the entropy bonus is not enough to maintain exploration. *Symptoms:* **Per-head entropy** metrics will drop near zero for one head while others remain higher. You may observe repetitive actions: e.g. every seed is placed in the same slot or the same blueprint is used for all germinations. If

the blueprint distribution becomes one-hot (except maybe a "noop" blueprint), the agent isn't trying alternatives. *Telemetry:* Log entropy for each action head and the frequency of each categorical choice. An entropy below ~10–30% of max for a head is a warning [21]. Also, the telemetry might raise an `ENTROPY_WARNING` or `ENTROPY_COLLAPSE` event if thresholds are crossed (e.g. policy nearly deterministic) [21]. *Mitigations:* Use **per-head entropy coefficients** to keep exploration longer on critical heads. For instance, increase entropy weight on the blueprint head so it continues trying different blueprints. Gradually anneal entropy rather than a sudden drop, and consider a minimum entropy floor per head [22]. If a head collapses anyway, reset it via higher entropy or even adding stochastic noise to logits temporarily. Ensuring the reward signals are well-shaped for all choices (so that trying a different slot/blueprint isn't consistently punishing) is also key – e.g. randomize slot ordering or provide symmetric rewards so no slot is "always bad".

4. **Invalid-Action Loop (Masking Issues):** If the agent frequently chooses actions that are masked (invalid), learning can stagnate. Although impossible actions are masked out, the agent might still assign probability mass to them and effectively end up executing a WAIT (no-op) without understanding why. This can happen if the policy network doesn't properly condition outputs (e.g. selecting an operation that makes a certain slot or blueprint choice invalid). *Symptoms:* High **mask hit rates** – e.g. telemetry shows a large fraction of decisions had at least one head masked [23]. Specifically, many actions result in `action_success=False` due to invalid combinations (logged as "action fails silently" in the code) [12]. The agent might appear to choose an op like GERMINATE, but nothing happens because it chose a slot that's full or a blueprint that's not allowed. This looks like frequent wasted turns. *Telemetry:* Examine the mask statistics and the `emit_last_action` logs – if `masked` flags are often true for certain heads, the agent is fighting the action mask. Also track how often `action_success` is false or falls back to WAIT implicitly. *Mitigations:* Strengthen **action representation** (see Section F) so that the policy naturally avoids invalid combinations – e.g. use a hierarchical action decode (choose op first, then only relevant sub-actions). In the interim, add a small **penalty for invalid actions**: if `action_success=False`, provide a negative reward signal so the agent learns to avoid those choices. Also verify the masking logic: it should be complete (cover all impossible cases) and applied *inside* the policy distribution (so the agent's entropy and gradients account for mask). If necessary, simplify the action space early in training (e.g. temporarily remove rarely-used ops) to reduce the chance of confusion, then add them back once basic competence is achieved.

5. **Observation Normalization Instability:** The observation features have very different scales and dynamics, which can lead to unstable normalization if not handled properly. For instance, **"improvement" is a raw percentage point value** that could swing significantly [24], and accuracy is fed as 0–100. If many features are unbounded or poorly scaled, the RunningMeanStd normalizer might adapt slowly or clip values, causing non-stationary inputs. *Symptoms:* The policy loss or value loss might spike sporadically (as the network encounters out-of-distribution inputs). Training curves might show oscillations. The critic might output inconsistent values (low explained variance). You might also see debug warnings or TODO comments about normalization (indeed the code flags an audit for improvement scaling [24]). *Telemetry:* Monitor the **mean and variance of observations** (the running stats) over time. If certain features (e.g. accuracy_delta or improvement) have a high variance relative to others or keep drifting, normalization might not keep up. Check if observations frequently hit the clip limits (±10) after normalization [25] – that indicates outliers. *Mitigations:* Manually **scale critical features** into a consistent range. For example, express accuracy as 0.0–1.0 instead of percentage [7], clip or log-scale the total_params, and definitely normalize "improvement" to a reasonable range (perhaps divide by 100 or use a tanh). This reduces the burden on RunningMeanStd. Additionally, use a slower momentum (the code uses 0.99 by default) for the normalizer if needed to adapt to

gradual shifts without overshooting [26] [27]. Finally, ensure that new seeds or stage transitions don't introduce abrupt observation jumps without corresponding normalization updates – if they do, consider updating normalization immediately after such events.

6. **Per-Slot Permutation/Identity Issues:** The agent may struggle to generalize behaviors across slots because the observation treats each slot position separately. The three (or N) slots are fed in a fixed order in the state vector [5], which can break symmetry. For example, the policy might arbitrarily favor slot0 for germination simply because it always appears first in the input vector. Likewise, when a seed in slot1 gets pruned and a new seed appears in slot1 later, the agent might not associate that it's a "new" seed – it just sees some features changed in slot1, potentially confusing past and present seed identity. *Symptoms:* The agent's behavior might be biased to specific slot indices (e.g. it only uses the first slot heavily while others remain under-utilized). Alternatively, it could mis-tune actions for a slot because it "remembers" the previous seed's performance in that slot via LSTM state. If a seed is removed and a new one germinated in the same slot shortly after, the policy's memory might cause suboptimal decisions (this is a partial observability issue – it can't directly observe seed identity). *Telemetry:* Compare seed usage across slots – ideally they should be roughly even, unless the environment inherently makes one slot special. Also track if slot-specific metrics (like improvement) reset to 0 when seeds restart; if not properly reset, the agent could be misled. *Mitigations:* Introduce **slot symmetry** in training: randomize the slot ordering in observations each episode, or explicitly permute slot feature order periodically so the policy can't latch onto a fixed position. Another approach is to augment observations with a unique seed identifier per slot (perhaps as one-hot encoded ID) so the agent can differentiate seeds – however, this could blow up input size and doesn't solve the permutation bias. As a simpler fix, enforce a usage of all slots in early curriculum (e.g. require or reward using different slots) so the agent explores them. Architecturally, one could use a permutation-invariant network module (like attention over slots) – see Section F for more – but that's a larger change.

7. **Partial Observability & Memory Limits:** The agent may face partial observability because not all relevant information is in the current state vector. For example, the decision to fossilize might depend on whether a seed's improvements are sustained over several epochs, or whether a seed has been attempted before – information that only an LSTM memory or additional features (like `epochs_in_stage`) can provide. If the LSTM or temporal window is too limited (or not used effectively), the policy could make myopic decisions. *Symptoms:* The agent might oscillate actions or make inconsistent choices that imply it "forgets" past outcomes. For instance, it might germinate a seed, then prune it a few steps later, then germinate a similar seed again – suggesting it didn't recall that blueprint was tried and failed. Another symptom is needing an excessive number of timesteps to decide to fossilize even when the criteria were met, because it lacks confidence from memory. *Telemetry:* Check the **LSTM hidden state usage** (if accessible, e.g. magnitude of hidden activations) and the episode length relative to memory span. If episodes are long (25+ steps) and credit assignment requires remembering something from step 1 at step 20, ensure BPTT (backprop through time) isn't truncating this (the code sets chunk_length equal to episode_length to avoid breaking sequences [14]). Look at **stage transition counts** – if seeds often time out at MAX_PROBATION_EPOCHS (auto-cull after 5 epochs [28] [29]) rather than being proactively fossilized or culled by the agent, it might indicate the agent isn't recalling how long a seed has been waiting. *Mitigations:* Increase the **LSTM hidden size or sequence length** if memory is a bottleneck (default 128 hidden for 25 steps is usually fine [14], but complex tasks might benefit from more). Provide **explicit features for time** – e.g. "epochs in current stage" or a timer until auto-cull – so the agent doesn't have to infer it. The SeedTelemetry features already include `epochs_in_stage` normalized [6]; if not using telemetry, consider adding a simpler

feature. Essentially, offload some memory burden to observations (making state more Markovian) wherever possible.

8. **Unbalanced Multi-Objective Signals:** The shaped reward combines many terms – counterfactual contribution, proxy improvements, PBRS potentials, rent cost, blending warnings, action penalties, etc. If these are not balanced, the agent might chase one objective at the expense of others. For example, if **compute_rent is too low**, the agent might add too many seeds and hurt final accuracy (over-parametrization) without enough penalty. Conversely, if rent or penalties are too high, we get the inactivity problem (#2). Another conflict: **blending warning vs contribution** – the agent might prematurely cull a seed at the first sign of trouble to avoid the blending penalty, even though patience could yield a net positive contribution. *Symptoms:* Weird policy behaviors that correlate with one reward component. E.g. the agent prunes seeds very aggressively as soon as any validation loss uptick occurs – possibly over-weighting the blending warning signal [30]. Or it might never prune at all because the contribution reward outweighs the warning (leading to performance crashes). Another symptom: large oscillations in value estimates if one term (like rent or stage bonus) changes abruptly when a threshold is crossed. *Telemetry:* Logging the **reward breakdown per timestep** is crucial. Watch for one component consistently dominating the reward magnitude. If, say, `bounded_attribution` (the main contribution reward) is an order of magnitude larger than others, the agent may ignore the rest. If `blending_warning` or `probation_warning` frequently maxes out to -10 [31] [32], the agent is experiencing strong negative signals that might override positive ones. *Mitigations:* Manually **rebalance reward weights** (see Section E) – e.g. reduce contribution weight if it dwarfs shaping, or vice versa. Implement a scheduling where some penalties ramp up later in training (when they matter more). Additionally, consider making certain signals binary or capping them, so they act as gentle nudges rather than overwhelming forces. For instance, instead of a harsh -10 penalty for waiting in probation, one could cap it at -1 in early training and only use full strength once the agent consistently reaches probation stage, to avoid scaring it off from ever using probation.

9. **Value Function Overestimation / Instability:** With high-dimensional observations and non-stationary rewards, the critic (value network) might struggle to approximate returns, leading to high bias or variance. In a complex shaped reward setting, the critic could latch onto easy-to-predict surrogate signals (like stage count) and **misestimate true future returns**. This manifests as high variance in advantages, policy updates overshooting (KL spikes), or even diverging value loss. *Symptoms:* The value loss might remain large or oscillatory while policy seems to improve, indicating value predictions can't keep up. The **explained variance** of the value function is low or even negative late into training (meaning the value predictions barely correlate with actual returns). We might see PPO clipping a lot of value updates if value_clipping is used (to prevent destructive updates). In worst cases, the policy might exploit value errors (e.g. appearing to maximize a flawed value function instead of actual returns). *Telemetry:* Monitor **KL divergence per update** and whether PPO hits the `target_kl` early stop often [33] – if yes, the updates might be unstable (possibly due to inconsistent advantages from a wobbly critic). Also track the ratio of episodes where the critic's predictions of final reward are way off (e.g. predicted high reward but got low). *Mitigations:* Use a **conservative learning rate and clipping** for PPO updates. Ensure `gamma` and `gae_lambda` are set appropriately for the episode length (they are: γ=0.995, $\lambda \approx 0.97$ by default [34] [35], which is good for ~25 steps). You might increase the critic weight (`value_coef`) slightly to focus on value fit, or apply a small L2 **weight decay to the critic** (the code allows that [36]) to prevent it from overfitting to noise. Another safeguard is the RewardNormalizer on returns [37] [38], which keeps returns scaled – ensure it's enabled so that large returns don't blow up the value loss. Finally, consider simplifying the reward during

initial training (reduce complexity) so the value function has an easier job (addressed via curriculum in section C and reward shaping in E).

10. **Catastrophic Forgetting / Mode Switching:** The environment's multi-phase nature (first improve host, then integrate seeds, etc.) might cause the policy to oscillate between strategies if not trained carefully. For instance, the agent might sometimes pursue a "seed-heavy" strategy and other times a "seed-light" strategy, depending on slight differences, and have trouble settling. This is partly due to the non-stationarity and partly because the reward landscape might have multiple local optima. *Symptoms:* Training curves with alternating periods of high and low performance, corresponding to different policies. One run the agent might germinate many seeds and then suddenly in another run (or after a policy update) revert to no seeds. You may notice high variability between training seeds (high standard deviation in number of seeds used per episode). The policy entropy might increase unexpectedly (agent exploring a new mode) after being low, indicating it "gave up" on the previous mode and is trying something else. *Telemetry:* Compare segments of training – e.g. a moving average of seed count or returns per episode. If there are clear regime shifts unrelated to changes in hyperparameters, the agent might be toggling between strategies instead of refining one. Also, see if any external triggers (like hitting a `MIN_IMPROVEMENT_TO_FOSSILIZE` threshold [16] or encountering a governor rollback event) correspond to these shifts. *Mitigations:* Introduce **curriculum phases** or reward shaping that guide the agent through a consistent strategy. For example, fix a strategy in early training (like only allow 1 seed at a time) so it experiences that thoroughly, then relax constraints slowly. This prevents the policy from jumping chaotically between extremes. Additionally, use **checkpointing and regularization**: if the agent finds a decent strategy, apply a bit more KL penalty or lower learning rate to gently improve it rather than allowing large policy moves that could erase it. Ensuring a continuous evaluation on a fixed policy (without updating) can detect if a seemingly good policy was just overfitted – if performance plunges once learning stops, it was oscillating rather than converged.

C) Training Curriculum Plan

Phase 0 – Sanity Check with Heuristic: *Before training the agent, verify the environment with the existing Tamiyo heuristic controller.* Ensure that the heuristic can complete episodes and achieve reasonable performance (e.g. final accuracy improvements) so we know the task is solvable. This sets a reference for reward signals and identifies any immediate issues (if even the heuristic fails, the RL agent will too). No learning yet – just run Tamiyo and log its behavior and reward breakdown for one run.

Phase 1 – Learn Lifecycle Grammar (Basic RL with heavy shaping): In this stage, we simplify and strongly guide the agent to learn the "grammar" of seed lifecycles – i.e. when to germinate, advance, fossilize in the correct sequence. - **Environment Simplifications:** Start with a **single seed slot** (or one seed at a time) so the agent doesn't worry about multiple concurrent seeds. Limit max_seeds to 1. This reduces complexity: the agent only focuses on the lifecycle of one seed from birth to death. Use a **shorter episode length** (maybe 15 epochs instead of 25) so the agent experiences an entire lifecycle within an episode, reinforcing start-to-finish credit assignment. - **Reward Mode:** Begin with the **"SHAPED" reward mode (contribution family)** [39] to provide dense feedback. This mode includes PBRS stage bonuses, contribution and proxy signals, and warnings – making it much easier for the agent to see which actions are immediately good or bad. The dense reward will "coach" the agent: e.g. +1 on germination, +1.5 on blending, etc. [20]. We also temporarily **relax penalties** that might discourage exploration – set rent penalty very low or zero and tone down blending/probation warnings. The idea is to **remove reasons to not use seeds** in this phase. - **Telemetry Usage:** Start **without telemetry features** for simplicity. The base 23 + slot features are enough to learn the basic sequence. Additional telemetry (gradient norms, etc.) might be noisy and not necessary to just complete a

lifecycle. We'll introduce it later when the agent needs finer judgments. (However, do enable basic metrics like `improvement_since_start` which are already part of the slot features [40] [41].) - **Action Space Constraints:** Disable rarely-used ops if any (for instance, if "SET_ALPHA_TARGET" or "ADVANCE" are in the space but not critical initially, we can mask them out in this phase). Focus on **GERMINATE → (train) → FOSSILIZE** as the primary operations, plus WAIT and CULL as needed. This teaches the agent the core loop: wait for plateau then germinate, blend, and finalize. - **Entropy and Exploration:** Use a relatively **high entropy coefficient** initially (e.g. 0.05 as default [22], or even a bit higher for certain heads) to ensure the agent tries all operations. In particular, ensure the op head and blueprint head explore – but since we have only one slot and maybe one blueprint for now, exploration there is naturally limited. - **Expected Outcome:** By the end of Phase 1, the agent should consistently germinate a seed (instead of always waiting), train it through blending, and fossilize it successfully. We expect it to get the shaped rewards for stage progression and not to get stuck. *Success criterion:* >90% of episodes the agent goes through GERMINATE→BLEND→FOSSILIZE at least once, and the average reward is close to the theoretical maximum shaping sum for one seed (~6.0 scaled by PBRS weight [20]). If it's germinating and then immediately culling or doing nonsense sequences, iterate on this phase by further tweaking reward or adding rule-based interventions (e.g. force it to wait a minimum period after germination before any other op).

Phase 2 – Multi-Seed Management (Increasing complexity gradually): Now that the agent can handle one seed, we introduce more slots and concurrent seeds, teaching it to make choices about *when* and *which* seed to grow. - **Environment:** Increase to 2 slots (then 3 slots in later trials). Allow at most 1 active seed per slot initially, and perhaps cap total seeds (e.g. max_seeds=2 for 2 slots, to avoid constantly filling/emptying both). This teaches the agent to **decide which slot to use** and manage seeds in parallel up to a small number. - **Blueprint Variety:** Introduce a second blueprint type in the action space. Ideally, choose two blueprint options that have noticeably different effects (e.g. one "easy" blueprint that reliably gives a small boost, and one more risky blueprint that could give a bigger boost or fail). This forces the agent to learn blueprint selection. Keep the blueprint set small at first (maybe 2–3 types out of 13) to avoid overwhelming the choice. - **Reward Mode:** Remain on **shaped reward** but we can start shifting toward a slightly sparser version once basic behavior emerges. For example, switch from SHAPED to **SIMPLIFIED** reward mode [42] after a certain number of episodes. The SIMPLIFIED mode (per code comment) still gives PBRS and a terminal bonus but might drop some dense terms like warnings [43]. This gentle reduction in shaping will test if the agent can still perform without every crutch. During this phase, we also **turn back on moderate penalties**: e.g. enable compute_rent with a low weight (0.01–0.05) [44] and blending warning at -0.1 steps [30]. The agent should learn that seeds have a cost and that bad seeds should be removed, *but* we keep these penalties mild to not discourage trying. - **Telemetry:** Still keep telemetry off to reduce observation size. Instead, rely on simpler signals: improvement and loss deltas per slot should suffice for the agent to tell if a seed is helping. We will add telemetry next phase for fine-grained control. - **Entropy Schedule:** Gradually decrease global entropy coefficient over time to let the policy consolidate. For example, over, say, 500k timesteps, anneal entropy_coef from 0.05 down to 0.02 linearly. This way early exploration transitions to exploitation. Also consider **entropy per head**: if in Phase 1 we noticed one head collapsed, we might set that head's entropy weight a bit higher here to keep it exploring (e.g. blueprint head might get 0.03 while others 0.02 once we anneal). - **Expected Outcome:** The agent should now handle multiple seeds: e.g. germinate a second seed after the first one blends, or decide which slot to allocate a new seed. We want to see diversified actions: sometimes both slots in use, sometimes one left empty if not needed. Also, it should start to learn to **cull failing seeds**: if one blueprint doesn't improve accuracy (val_accuracy delta goes negative), the agent should invoke PRUNE/CULL on it, leveraging the blending warning signal. *Success criteria:* The average number of seeds used per episode >1 (showing multi-seed usage), and final model accuracy surpasses what a single-seed policy achieved. Also the agent's actions per seed should align with improved contributions (if it consistently fossilizes seeds that had negative impact, that's a failure – it should have pruned them instead).

Phase 3 – Refine Architectural "Taste" (Performance-focused): Now we aim for the agent to outperform the heuristic, by making smart decisions on *which seeds to germinate and when*, not just completing lifecycles. We up the realism and difficulty. - **Full Blueprint Set & Telemetry:** Enable the **full range of blueprint types** (all 13) and turn **telemetry features ON** for each slot. The agent now gets detailed signals like gradient health, vanishing/exploding gradient flags, and per-seed accuracy improvement stats [6] [7]. These 10 extra features per slot will help it discern subtle differences in seed quality (e.g. if a seed's gradients look unhealthy, it might decide to cull it early). The blueprint one-hot already indicates the type [45], so the agent can associate telemetry patterns with blueprint types over time. - **Environment:** Use the full 3-slot setup, and allow up to `max_seeds=3` or more (if hardware can handle it). Now the agent can potentially germinate seeds in all slots and even replace them sequentially. Remove any extra simplifying constraints – e.g. if earlier we disallowed ADVANCE or SET_ALPHA_TARGET ops, now include them if their implementation is ready. The agent should now manage the complete action space as designed. - **Reward Mode Progression:** In this phase, consider **annealing from shaped to sparse rewards** over training iterations. For example: for the first N episodes in this phase, use the SIMPLIFIED shaping (PBRS + terminal). Then switch to **SPARSE mode** where the only reward is at episode end: perhaps final validation accuracy minus a penalty for total parameters [46]. We might also try a **hybrid schedule**: start each training run with shaping on, then once the policy is stable, turn shaping off for a few episodes to see if it still performs (this can reveal if it was exploiting shaping artifacts). The curriculum could alternate between shaped and sparse to make the policy robust. The goal is to ensure the agent ultimately optimizes the true objective (accuracy minus complexity) and not just the shaping proxies. - **Entropy & Learning Rate:** By now, exploration should focus on fine details (which blueprint when, etc.), so keep entropy low but not zero. Possibly maintain a small entropy bonus on blueprint and op heads to avoid converging to a single architecture solution – we want some continued exploration of architectures. The learning rate can be lowered in this final phase to fine-tune (to perhaps 1e-4 if it was 3e-4) for stability. - **Expected Outcome:** The agent develops "taste" – e.g. it learns which blueprint patterns lead to higher final accuracy. We expect it to choose, say, more effective blueprints for the given task (maybe it learns that **attention modules are better later in training, while conv seeds help earlier**, etc.). It should time germinations smartly (not germinate too early when host is still improving on its own, and not too late either). Ideally, it surpasses the heuristic Tamiyo's performance by using seeds more optimally – perhaps achieving higher final accuracy or same accuracy with fewer seeds (less overhead). *Success criterion:* Consistently higher final reward in sparse mode than the heuristic baseline. Also, qualitative inspection: the agent's sequence of decisions should look sensible (no obviously wasteful actions). For example, one might see it germinate a small "conv_light" seed early for a quick gain, later add a heavy seed once the model saturates, and avoid using known bad blueprints (maybe it rarely uses one that historically gave poor results, showing it learned an architectural preference). If these behaviors emerge, we've successfully taught the agent the full task.

Throughout all phases, maintain checkpoints of the policy. The curriculum can be iterative: if in Phase 3 the agent collapses (e.g. stops using seeds because sparse reward was too hard), one can backtrack – reintroduce some shaping or reduce blueprint variety – and try again. The final output of the curriculum is an agent that has seen progressively more complex scenarios and has the knowledge encoded (via its network and LSTM memory) to handle them all.

D) PPO Hyperparameter and Architecture Plan

PPO Hyperparameters:

- **Learning Rate:** Start with a moderate LR ~ `3e-4` (the default) [47]. This is typically stable for PPO. Because our environment is high-variance and non-stationary, we don't want to go too

high. We can schedule LR decay if needed – e.g. halve the LR after certain number of episodes once the policy has learned the basics (to fine-tune stability in later training). If we observe policy oscillations or divergence, we might lower LR to `1e-4`. Conversely, if progress is very slow in early phase, a short period at `5e-4` could be tried, but cautiously.

- **Clip Ratio & Target KL:** Use `clip_ratio ≈ 0.2` initially [47], which is standard. But monitor the KL divergence each update – if we often hit a KL > ~0.015 (target) and trigger early stopping [33], it means updates are too aggressive. In that case, either reduce `clip_ratio` to 0.1–0.15 or reduce LR. The code has `target_kl = 0.015` by default [36] – keep that. If the policy is very sensitive (e.g. we see occasional ratio explosions >5) we could even lower target_kl to 0.01 to be safer. Essentially, we want the PPO updates to be small trust-region steps; given the complex reward, small updates help avoid sudden policy flips.
- **Discount Factor (γ):** Keep `gamma = 0.995` as set [34]. This high gamma makes sense because the episode (25 epochs) is relatively long and we want to credit late outcomes (fossilization, final accuracy) back to earlier actions. Using the same gamma for both environment rewards and PBRS is critical (they do this by design) [48], so we won't change it. If we ever extend episodes (say to 50 epochs for longer training), we might increase gamma slightly (0.999) to maintain long-horizon credit.
- **GAE Lambda:** Use `λ = 0.97` (slightly higher than the default 0.95) [35] to reduce bias in advantage estimation. A higher lambda leverages the long trajectory by smoothing advantage over many steps, which is good because many rewards (especially contribution improvements) are delayed by several epochs. We don't go to 1.0 (which would be full return) to keep some variance reduction. If we find advantages are still noisy, we might experiment with λ up to 0.99.
- **PPO Batch and Epochs:** Each PPO update will use a batch of experiences from parallel environments. By default, with `n_envs=4` and episode_length=25, one batch = 100 steps (before minibatching). We set **mini-batch size** to 64 [49] and **PPO epochs** to 10 [50]. This means each step's data gets seen ~15 times per update, which is a good balance of learning vs. not overfitting. We'll keep these defaults initially. If training is unstable (e.g. policy oscillations), we might *reduce* PPO epochs (to 5) to update more conservatively per batch. If we need more sample efficiency and things are stable, we could increase to 15 epochs. Another parameter is `ppo_updates_per_batch` (if defined differently from epochs, but likely the same concept here) – effectively we are doing 10 epochs per batch, that's fine.
- **Value Function Coefficient:** Keep at `0.5` (standard) [51]. This gives equal scaling to critic loss relative to policy loss (since advantages are normalized). In case we see the value loss dominating or critic diverging, we might raise this to 0.7 to prioritize fitting value (stabilizing training), or lower to 0.3 if we think the critic is overfitting and dragging policy updates. We'll monitor explained variance to decide.
- **Max Grad Norm:** Stick to `0.5` for gradient clipping [52]. This is fairly strict (clipping grads if norm >0.5). It will help prevent any sudden spike from blowing up network weights – a nice safety for our complex environment. If we find training very slow and stable, we could relax to 1.0, but 0.5 is a good starting point to avoid numerical issues especially with an LSTM in the network.
- **Entropy Coefficient (Global and Per-Head):** Start with a **global entropy coef ~0.03–0.05**, as mentioned. The defaults are 0.05 decaying to 0.01 [22]. Implement a schedule: for example, linear decay from 0.05 at start to 0.01 by halfway through training. Now per-head adjustments: the action space heads have differing sizes and importances.
- The **op head** (LifecycleOp) has only a few options (4–6 ops). We'll give it a moderate entropy weight – enough to try all ops, but since ops have big consequences, we don't want random oscillation forever. The default 1x global is fine here.
- The **slot head** depends on number of slots (3). We should encourage trying different slots, but since slots are symmetric, the policy should naturally explore them if not biased. No special weight needed beyond global.

- The **blueprint head** (if 13 categories) could use a slightly higher entropy push because we want thorough exploration of architectural choices. We can scale its entropy bonus by a factor (e.g. 1.5x the normal coefficient) for the first part of training. That means effectively a ~0.075 initial coef on blueprint logits before decaying. This helps avoid converging on one blueprint too early.
- The **blend algorithm head** (3 options: linear, sigmoid, gated) might also benefit from a bit more exploration weight, as the differences might be subtle. But we can keep it same as op head since it's small.
- For any additional heads like **tempo or alpha target** if/when they're enabled: those are numerical parameters with discrete choices. They likely have an optimal default (e.g. maybe medium tempo is often best). We still want the agent to try extremes initially (fast vs slow tempo, etc.), so ensure they have some entropy. We'll treat them akin to blueprint: give them time to explore then let entropy decay.
- **Value Clipping:** Use **value function clipping** to stabilize critic updates. The code sets `DEFAULT_VALUE_CLIP = 10.0` [53], meaning the value estimate change per update is limited to ±10 (which is high relative to typical reward scales, but since rewards are normalized, it effectively prevents huge swings). We will enable this. This prevents the Q-value of a state from changing too drastically in one go, which is important if, say, a seed's outcome suddenly produced a big return – the critic will update gradually. We might tighten this if needed (maybe clip to 5) if we see value overshoot.
- **Weight Decay:** By default, weight decay is 0 for policy and critic (the code only allows a decay on critic if set) [36]. As a plan, we'll keep weight decay off initially for the policy network to not impede learning nuanced behaviors. However, for the critic, a small weight decay (like 1e-4) could help prevent overfitting to the shaped reward peculiarities. We leave it at 0 initially and watch the critic's performance. If we notice signs of value overfitting (e.g. very high value predictions for certain states that don't materialize), we can introduce a slight decay on the critic network in later tuning.

Policy Network Architecture:

- The current architecture is a **Factored Recurrent Actor-Critic** with an LSTM core [54] [55]. It presumably consists of an input layer (taking the normalized feature vector of size ~74 or more), some hidden layers, an LSTM (size 128 hidden by default [56]), and then separate output layers for each action head plus a value head. We will use this default architecture as our baseline.
- **Hidden Layer Sizes:** The default feature embedding is 128 dims [53], and LSTM hidden 128. This seems reasonable given the state size (74–100+ features) – it can capture interactions. If we find the policy isn't capturing complex interactions (like which blueprint works in which stage), we might increase the hidden size to 256 (which in fact may already be in use as per PPOAgent default showing hidden_dim=256 [57] – possibly the actor and critic MLP sizes). We should confirm the network's actual layer sizes and match them to complexity. Starting with 256 hidden in the fully-connected layers and 128 in LSTM is fine.
- **Activation & Initialization:** Use default activations (likely ReLU or Tanh in LSTM). No special changes unless we encounter dead neurons or saturations, in which case we might switch to a smoother activation (like Swish) – but that's experimental.
- **Hierarchical Action Structure:** The architecture currently outputs all heads in parallel (with masking to invalidate combos). In later improvements (Section F) we suggest making it hierarchical or auto-regressive. But for baseline, we keep it parallel for simplicity and speed. The implication is the network must learn the dependencies (like that if op=WAIT, blueprint choice is irrelevant) on its own. This is doable but slower. We mitigate by providing clear mask feedback and possibly special loss terms (not in base plan, but if needed we could, for example, not update blueprint head on steps where op=WAIT to avoid noise).

- **Exploration Constraints:** Ensure the MaskedCategorical distribution is used so that masked actions don't get probability [4]. This is already in place. We might also incorporate an **entropy penalty per head** as discussed. If a particular head (say blueprint) isn't exploring enough, we can temporarily boost its entropy coefficient. Conversely, if one head is too random for too long (maybe the slot head jumping around causes instability), we could reduce its relative entropy earlier.
- **LSTM usage:** The recurrent state is essential for remembering things like how long a seed has been in a slot or the last action taken (since the observation doesn't directly encode the last action). We'll make sure the PPO training uses full sequence unrolling (no truncation mid-episode). The code's DEFAULT_EPISODE_LENGTH and chunk_length alignment ensures that [14]. We will keep that to fully leverage memory. If training is slow, one might be tempted to chunk the episodes, but that risks losing memory of early actions which is crucial; so we avoid truncation.
- **Normalization Layers:** We already normalize inputs. We might consider **output normalization** for value predictions (they do clipping). Another subtlety: if the policy logits for different heads have very different scales (especially with independent heads), we might want to normalize advantages per head (some implementations do that to balance multi-discrete action learning). The code hints at computing per-head advantages [54] [55] – we should verify if that's done. If not, we might implement a small adjustment: e.g. normalize advantage by the number of heads when updating, or ensure each head's gradient impact is scaled appropriately. For now, assume their implementation handles it (perhaps via combined loss).
- **KL and Anomaly Detection:** We will enable any built-in anomaly detectors (they have ratio explosion/collapse thresholds [58] and entropy collapse warnings). If those trigger, our plan is to respond by adjusting hyperparameters (like backing off LR or increasing entropy as needed). The training loop should log if a ratio explosion is detected (that often indicates a need to reduce LR or clipping).

Trade-offs specific to Esper environment: This environment has **vectorized parallelism** (so many environments running, giving diverse data each update), which helps stability – we will use at least 4 envs as default [59]. The rewards are shaped and mostly bounded in [-10, 10] after normalization, which PPO can handle. The multi-categorical action means the policy loss is sum of losses for each head. We must ensure one head's domination doesn't destabilize updates – e.g. if blueprint head has 13 classes and is highly uncertain, its entropy gradient could be larger than the slot head's. That's where per-head entropy tuning helps. Also, **action masking** means the effective policy dimensionality changes with state (some actions not available in some states). PPO can handle this, but we should monitor if the policy shows any unnatural bias when masks change (for example, if a certain op is rarely available, its probability estimation might be poor – but since it's masked, that's okay). We also consider the **multi-objective reward**: because of normalization and clipping, PPO should naturally focus on the combined reward. However, if the reward goes through phases (e.g. big spike at end), the advantage estimation might be tricky. Our hyperparams (γ, λ, etc.) are chosen to smooth that out.

In summary, the initial hyperparameter set is: *lr=3e-4, clip_ratio=0.2, target_kl=0.015, γ=0.995, λ=0.97, n_epochs=10, batch_size=64, value_coef=0.5, max_grad_norm=0.5, entropy_coef_start=0.05 (decay to 0.01), separate head entropy scaling (higher for blueprint), recurrent architecture with 128 LSTM, full observation normalization, reward normalization on, value clipping 10.0*. We will adjust these as needed based on training feedback but these are a solid starting point given typical PPO defaults and the specifics of this task.

E) Reward Shaping Audit

The current shaped reward is a sum of several components. We dissect which terms might dominate or conflict, and how to adjust them:

- **Dominant vs Secondary Signals:** Likely the **primary contribution reward** (based on counterfactual or improvement) and the **PBRS stage potential** are the largest terms. For example, a seed that significantly improves accuracy can yield a big positive reward (contribution_weight is 1.0, so a +5% accuracy jump gives +5 reward before normalization), whereas stage bonuses are capped (the total from Dormant to Fossilized is 6.0 in potential) [60] and scaled by pbrs_weight (default 0.3) [61] giving at most +1.8. So **if a seed does very well, contribution reward dominates; if seeds do poorly or nothing, the stage bonuses (and penalties) dominate.** Also, negative penalties like rent and warnings are relatively small per step (blending_warning around -0.1 to -? escalating, probation_warning up to -10 in worst case [62], rent maybe -0.1 or -0.5 depending on growth). However, those can accumulate if the agent stalls.
- **Conflicting Terms:**
- *Potential Shaping vs. True Objective:* Stage potentials (PBRS) are potential-based and theoretically shouldn't alter optimal policy [48], but in practice they can encourage behavior that's suboptimal for final accuracy. E.g., a seed can be rushed to BLENDING to grab that +1.5 shaping [63] even if its contribution to accuracy hasn't fully materialized. There's some conflict between "take time to train the seed well" (accuracy reward) and "advance stage quickly" (PBRS reward). The small terminal bonus for fossilizing is intentionally tiny (0.5) to reduce conflict [63], but the larger BLENDING bonus might still incentivize seeds reaching blend stage perhaps earlier than ideal.
- *Counterfactual Contribution vs. Proxy Improvement:* When a seed is not blended yet, the reward uses **accuracy_delta as a proxy** (scaled down) [64]. This could conflict with actual contribution measured later. For example, the agent gets a positive proxy reward because validation accuracy improved after germination, but later the true counterfactual shows the seed had minimal effect. This mismatch can confuse the policy – it was rewarded early but might get less reward or even penalty later for the same seed. They mention an attribution discount/ratio to avoid over-crediting such "ransomware" seeds (ones that piggyback on host improvement) [65] [43]. Still, this is a delicate balance: if the proxy is too rewarding, agent will germinate seeds all the time to get transient bumps; if too low, the agent might not see any reward until blending which is delayed.
- *Compute Rent vs. Contribution:* Rent penalizes adding parameters regardless of outcome [19] [18], while contribution rewards improvements. If a seed gives a small accuracy improvement but is large, these can directly conflict (e.g. +0.5% accuracy but +30% params might yield a net negative reward if penalty outweighs gain). Conversely, if rent is too lenient, the agent might spawn many seeds that collectively improve accuracy a bit but bloat the model – not truly optimal. So finding the weight of rent (0.05 vs 0.5 etc.) is about balancing resource cost vs performance benefit.
- *Blending Warning vs. Exploration:* The blending_warning term gives an early negative reward if a seed in blending is making things worse [30]. This helps the agent learn to cut off failing seeds, but it might also discourage any risky seed. If the agent experiences a strong penalty the moment a seed's contribution dips, it might prematurely prune seeds that could recover (over-conservative). That conflicts with maximizing eventual contribution (sometimes patience yields more final reward). Similarly, probation_warning penalizes waiting too long at the end [62] – it conflicts with the agent's desire to wait for maybe a marginal improvement; it forces it to decide quickly (fossilize or cull). The conflict here is between *thorough evaluation* vs. *speed*. These warnings are shaping intended to push a timely policy, but they might overshoot.
- *Action Costs (Interventions) vs. Doing Nothing:* If there are small costs for certain ops (not explicitly detailed, but possibly "intervention cost" mentioned in SIMPLIFIED reward), that could conflict with exploration. For instance, if every germination had a -0.1 cost just for trying, that

discourages germination unless a seed is very promising. We need to ensure any action cost is low enough that a potentially good seed's positive reward dwarfs it. Otherwise, agent might lean to inaction.

- *Terminal Bonus vs. Episode Return:* There's a final reward for ending an episode with seeds fossilized (if using sparse mode, maybe accuracy – baseline accuracy, etc.). If we use a terminal reward like "final accuracy minus initial accuracy", it aligns with the objective, which is good. But if we still have shaping on, that terminal reward might be relatively small compared to accumulated shaping (depending on scenario). In sparse mode, it's all that matters. In shaped mode, it could be overshadowed by stage bonuses. We should ensure the terminal/ground-truth reward is significant enough that the agent ultimately cares about final accuracy, not just mid-run bonuses.

Reward Component Adjustments (Ablations & Reweights):

1. **Ablation 1 – Remove PBRS Shaping:** Run an experiment without any stage potential rewards (set pbrs_weight = 0) [61]. In this condition, the agent is rewarded purely for actual performance improvements (contribution) and penalized for costs, but no bonus for moving seeds through stages. *Hypothesis:* The agent might be less "eager" to generate seeds because the immediate +1 from germination or +1.5 from blending is gone; it will only do so if it expects a contribution. This tests whether PBRS was causing reward hacking. *Expected observation:* If performance does not drop too much and the agent still uses seeds, then PBRS wasn't strictly necessary. But if the agent becomes very passive or slow to act, PBRS was providing needed incentive. This ablation tells us how dependent the current training is on shaped signals for exploration.

2. **Ablation 2 – Disable Blending/Probation Warnings:** Remove or zero out the `blending_warning` and `probation_warning` terms [66] [67]. This means the agent no longer gets penalized for letting a bad seed linger or for waiting in probation. *Hypothesis:* Without blending warning, the agent might tolerate seeds that degrade performance longer (perhaps hoping they turn around or just not noticing the slight loss), potentially hurting overall accuracy. Without probation penalty, it might leave seeds in probation indefinitely to farm contribution (though the PBRS potential stops increasing after fossilize stage 7 is reached). This ablation checks if these warnings are crucial for the agent to learn culling and timely fossilization, or if the natural terminal reward and counterfactual signal are enough. *Expected observation:* We might see the agent become slower to cull – final accuracy could drop because it doesn't remove bad seeds promptly. If so, these warnings are doing their job. If not, maybe the agent can self-correct via final rewards, meaning we could simplify the reward function by dropping these.

3. **Ablation 3 – No Rent Penalty:** Set `compute_rent_weight = 0` (no parameter cost penalty). This effectively makes the agent ignore model size and focus solely on accuracy. *Hypothesis:* The agent might spawn more/larger seeds than optimal, possibly improving accuracy a bit more but at the cost of model bloat (which isn't penalized). It might also discover strategies like germinate many seeds since there's no downside except maybe interference (the environment's stability could suffer). This tests how much the rent term was restraining the agent. *Expected observation:* We would likely see an **increase in seeds utilized** (seed_utilization approaches maximum) and possibly higher raw accuracy, but if we measure "accuracy-minus-param" externally, it might not be better. If the agent's returns go up without rent, it means rent was holding it back a bit (maybe too strict). If returns don't change or policy stays similar, maybe the agent wasn't hitting the rent penalty much anyway (maybe it naturally used few seeds). This informs whether to tune rent weight.

4. **Reweight Schedule 1 – Anneal Shaping -> Sparse:** As mentioned, gradually reduce shaping over training. Concretely: start with `pbrs_weight = 0.3` (default) and `contribution_weight = 1.0` during early training, then after X episodes, linearly decrease pbrs_weight to 0 over the next Y episodes, and possibly also drop proxy contribution usage in favor of true contribution only. Simultaneously, increase reliance on the **terminal sparse reward** (final accuracy outcome). By the end, the policy should be optimizing mostly the sparse objective. *Rationale:* This reduces Goodhart risk by ensuring the policy that worked with shaping is forced to also work when only real outcome matters. If it was exploiting shaping quirks, it will get corrected when shaping is removed. This schedule ensures exploration initially (thanks to shaping) and optimality finally (thanks to sparse reward). We expect some performance dip during the transition (the agent may need to adjust), but if things are well-aligned, it should recover and perhaps do even better on final accuracy.

5. **Reweight Schedule 2 – Increase Punishment for Bad Seeds Late:** Early in training, we want the agent to try seeds even if some are bad. Later, once it knows what's good, we can get stricter about bad seeds. For example, **schedule the blending_warning to ramp up over time**: start with a mild penalty (e.g. -0.05 per bad epoch) and increase it to -0.2 or more in later training. Similarly, the probation_wait penalty could be minimal early on (just a gentle nudge) and become the full -1,-3,-9 exponential later. This way, initially the agent isn't too scared to experiment, but as it gains confidence, we enforce that it should not waste time with failing seeds. This reduces the risk of the agent intentionally keeping seeds around just to see if they might improve (once it should know better). In code, this could be achieved by dynamically adjusting `ContributionRewardConfig` or through a custom schedule outside the PPO (like a callback that changes config after N episodes).

6. **Reweight Schedule 3 – Emphasize Final Accuracy after Convergence:** In the final stages of training or fine-tuning, we might explicitly multiply the final performance component. For instance, if using RewardFamily.CONTRIBUTION, we could add an extra term at the very end of an episode: `+ α * (final_val_acc / 100)` (with α maybe 1.0 or 2.0). Or if using RewardFamily.LOSS, something like `-val_loss` scaled. The idea is to ensure that among all things, the agent ultimately maximizes validation accuracy. This is a check against Goodhart: no matter what it did for shaping, the final big reward comes from the actual metric we care about. If the agent somehow learned a policy that excels in shaping but yields subpar accuracy, this final emphasis will create a gradient pulling it toward fixing that. Essentially, it's like a guaranteed "ground truth" reward at the end. We have to be careful to normalize it similarly (maybe using RewardNormalizer so it doesn't dwarf other rewards unpredictably). But done properly, this can align training with evaluation metrics.

7. **Ablation 4 (Counterfactual vs Proxy):** As a deeper experiment, compare training with **true counterfactual contributions** vs with only proxy improvements. If we have the capability, run one version where we *disable counterfactual measurement* – so the reward always uses improvement since germination as the metric (no direct A/B test of seed). Another version, always use counterfactual when possible and maybe ignore proxy even early (i.e. simulate we can perfectly measure seed contribution every time). This tests the reliability of the counterfactual mechanism. If the agent trained with true contributions learns much better (likely yes, because it gets accurate credit), it means our proxy/discount approach in the default shaped reward might need tweaking. If there's little difference, our proxy was sufficient. This experiment is more on the research side to validate the reward design.

8. **Goodhart Mitigation Plan – Counterfactual Guardrails:** We propose adding a **verification step** in reward calculation: for any seed that is fossilized, perform a final counterfactual

evaluation (e.g. measure validation accuracy without that seed at the end) – if it turns out the seed's contribution was negative or below a threshold, retroactively penalize or nullify the earlier rewards given for that seed. Essentially, "claw back" rewards if the seed didn't actually help in the end. This is complex to implement online, but one approach: log each seed's cumulative contribution reward and if final contribution < 0, apply a negative bonus equal to that log (taking back what it gained). This reduces the incentive to do things that only temporarily boosted reward. Scheduling-wise, you might introduce such a mechanism later in training once the agent can handle it. This is a qualitative strategy to avoid the agent exploiting the proxy signals.

9. **Reward Mode Experiments:** Try the different RewardMode options explicitly:

10. **SPARSE:** Only give reward at episode end: `final_accuracy - baseline_accuracy - β*(total_params)` (like a true evaluation metric). The agent will get zero reward for intermediate steps except maybe a tiny negative for culls (if using minimal). This is the hardest setting, but it tests if the policy can still learn with pure outcome-driven signal. It likely won't learn from scratch (the search space is huge), but it's a good sanity check for a well-initialized policy (after shaped pre-training).
11. **MINIMAL:** As coded, gives terminal accuracy-minus-cost and an early cull penalty [46]. This encourages not leaving seeds idle. It's simpler than full shaping. We'd observe if the policy still performs without all the bells and whistles.
12. **SIMPLIFIED:** (likely PBRS + small costs + terminal) which we partially used in curriculum. Confirm that this indeed yields similar final performance with less complexity – if so, it's a safer long-term reward to use (less risk of weird incentives). By comparing these, we ensure our shaping approach isn't hiding a fundamental issue – e.g., if a policy trained with shaping fails completely when switched to sparse, we know it was overfit to shaping signals.

From these audits, one likely conclusion is to adopt a strategy of **progressive shaping reduction**. Start with robust shaping to get the agent off the ground, then gradually remove artificial incentives. The final policy should ideally do well on a mostly sparse or simplified reward, indicating it's truly optimizing the objective (and won't fall apart when deployed without shaping). We also expect to adjust weights like: - Lower **PBRS weight** as training progresses (from 0.3 down towards 0 or very low). - Possibly lower **contribution weight** if it overshadows everything (though 1.0 seems okay, we might try 0.5 and see if agent still learns – if yes, that means it doesn't need huge reward for improvements and can pay more attention to other terms). - Increase **rent weight slowly** to gently nudge the agent to be parameter-efficient after it learned to gain accuracy. E.g., start at 0.0 in Phase 1, 0.01 in Phase 2, 0.05 in Phase 3 [44], and maybe 0.1 in final fine-tuning, to really ensure it uses minimal necessary capacity. - Adjust **terminal bonus scaling**: DEFAULT_FOSSILIZE_TERMINAL_SCALE = 3.0 [68] which multiplies into the final seed reward at episode end. We could schedule this too – maybe towards the end, increase it, meaning the agent gets a nice bonus for successfully integrating a seed by episode end, encouraging completion of good seeds rather than leaving them hanging (if that was an issue).

Overall, the reward shaping audit suggests being dynamic and cautious: too much shaping can lead to perverse strategies, too little and learning stalls. By ablations and schedules above, we find a sweet spot: *enough shaping to guide, but not so much to misguide.*

One concrete Goodhart risk mitigation: **ensure any shaping term corresponds to a genuine desired effect.** For instance, PBRS is fine because it's potential-based (doesn't alter optimal theoretically) [48]. The warnings and rent are more ad-hoc – we'll keep an eye that they don't create unintended incentives (like the agent trying to game the rent by temporarily pruning seeds to reduce params at end – unlikely since rent is continuous and small). If any weird trend appears (like oscillating seeds just to reset potentials), we'll adjust or remove that component. By the final training runs, we anticipate using a

simplified reward function (maybe just contribution + small PBRS + final outcome) that is robust and aligned with the actual performance metric.

F) Observation and Action Representation Improvements

Observation Feature Improvements:

- **One-Hot or Categorical Encoding for Stage:** Currently stage is provided as an integer 0–7 in the slot features [2]. This treats stage as a numerical value, which might mislead the network (it might think stage 4 is "twice" stage 2, etc.). It's better to encode stage as a one-hot vector of length 7 (or 8 including 0) or as separate binary indicators for key stages. This is an easy patch: replace the single `stage` float with a 7-dim one-hot. It will slightly increase obs dimension but ensure the policy can distinctly recognize DORMANT vs TRAINING vs BLENDING, etc., without confusion. This change is low-risk and straightforward.
- **Normalize Accuracy to [0,1]:** As noted, training/validation accuracy are currently raw percentages in the base features (e.g. 85.0 for 85%) [69]. We should scale those to 0–1 range (by dividing by 100) before feeding to the policy (similar to how SeedTelemetry does [7]). This keeps all features roughly on comparable scales and makes normalization easier. It's a trivial change in the feature construction.
- **Log-scaling for `total_params`:** The total parameter count can vary by orders of magnitude if seeds are large. Instead of feeding the raw count (which could be, say, 1e6+), we can feed `log10(total_params)` or `total_params` normalized by a baseline. The code currently just casts total_params to float [70], which could be huge. At minimum, if using RunningMeanStd, it will slowly adjust. Better to give a more stable representation: e.g. parameters as a fraction of some reference (like fraction of baseline model params) or log-scale. This is an easy patch. It helps the agent understand changes in model size in a bounded way.
- **Feature for "time since X":** We should add explicit features for time-related aspects that are currently implicit. Two helpful ones: **epochs since germination per seed** and **epochs since last seed operation**. The telemetry has `epochs_in_stage` [71] which covers some of this (for seed). We can include that even without full telemetry by computing it or adding a counter that resets on stage change. Another could be a global counter of epochs since any germination or cull – this might inform the agent about host training progression and pacing (like "it's been 5 epochs without a new seed, maybe consider germinating if plateaued"). These are moderate difficulty (need to track events), but not too hard – the environment state machine can easily provide such counters.
- **Include Mask/Availability Summary:** The observation could include a summary of what actions are currently possible. For example, a binary feature "slot_available" (1 if any free slot to germinate) or "active_seeds_count". We do already have seed_utilization fraction [17], which is good. We could add a feature for "number of seeds currently blending" or something if that matters. Also, if blueprint choices depend on topology, perhaps a feature indicating model type (CNN vs Transformer) which might already be known (TaskConfig.topology) – ensure it's input to the policy. A simple way: an input bit for "is_cnn_topology" so the policy can learn that certain blueprint indices are effectively invalid (if masking doesn't handle it fully).
- **Telemetry Feature Integration:** When telemetry is enabled, ensure those 10-dim features per slot [72] are appropriately normalized (they mostly are in [0,1] or -1 to 1 already). Potential improvement: some telemetry signals like gradient_health or vanishing/exploding flags could be combined or filtered if noisy. If the 17 extra dims per slot (as the question mentions original plan) are too many, consider selecting the most useful telemetry signals. E.g. gradient_norm (normalized) and accuracy_delta are likely important; maybe the raw accuracy (which is duplicative of global val accuracy) could be less needed in per-slot telemetry. However, since it's already compact (10 dims) and normalized, we can feed all.

- **Bounded Improvement Feature:** The `improvement` per slot currently might be unbounded (or at least not explicitly bounded) – it's the counterfactual contribution or accuracy delta (in percentage points) [40] [41]. We should clamp or scale this feature to a sensible range. For example, improvements beyond ±10% could be rare; we might cap it at ±10 or normalize by dividing by 100. Better, express it as a proportion of some target (if target accuracy is e.g. 90%, a 1% improvement is significant). In code there was a TODO to audit this [24]. So implementing that TODO: maybe set improvement feature = tanh(actual_improvement/5) or something to compress outliers.
- **Slot "identity" feature:** If we want the agent to distinguish seeds beyond just blueprint and stage, we could include an **ID or age for each seed**. For example, a seed index or spawn order number (normalized) could be given. Or even simpler: include the `seed_id` hash as an embedding (not straightforward for an arbitrary string). This is more of a research idea – it could help the agent not confuse one seed with another. However, with 3 slots and seeds not moving between slots, using slot index plus stage plus blueprint might be enough; we might not need a unique ID. This would be a heavier change and maybe unnecessary if our other adjustments handle the needed info.
- **Combining Historical Info:** Currently, the observation includes recent loss and accuracy history (last 5 values) [5]. This is good for trend detection. We might consider adding a **smoothed trend feature** explicitly – e.g. "plateau_metric" which indicates how plateaued the host is (they do have plateau_epochs count [69]). If not already included, ensure plateau_epochs is there (it is [69]). Possibly add "improvement_rate" (like average accuracy increase per epoch last 5 epochs) as a single feature to summarize trend. But since we have history and plateau count, the network can derive it. This is minor.
- **Simplify Ranges:** Ensure all features roughly map to [-1, 1] or [0, 1]. For instance, epoch number goes from 0 to max_epochs (say 25), which is fine but we might normalize epoch to [0,1] by dividing by max_epochs (TaskConfig knows max_epochs) – this is doable when constructing obs (could replace obs['epoch'] with epoch/max_epochs). Global step similarly can be normalized by max_steps if known. These small tweaks make it easier for the network to generalize to different training lengths if needed.

Many of the above are "easy patch" – e.g., one-hot encoding stage, rescaling numbers, adding obvious counters can be done in the feature extraction code without major refactoring (the features are constructed in Python in `signals_to_features` and `obs_to_multislot_features` [73] [74]). The more "researchy" improvements are adding unique seed IDs or advanced features like "expected improvement" – those require more design and testing.

Action Representation Improvements:

- **Conditional Action Decoding (Hierarchical Policy):** Instead of producing all 9 action components in parallel, we can structure the policy to make decisions in a sequence, reflecting the logical dependencies:
- First choose `op` (the high-level operation).
- Based on the op, choose relevant sub-actions. For example, if op=GERMINATE, then choose `slot` and `blueprint` (and blend algorithm). If op=SET_ALPHA_TARGET, choose which seed/slot and the target value and speed, etc. If op=WAIT, no further choices needed.

This could be implemented by an architecture where the policy first outputs an op distribution, then conditional layers output other distributions. In practice, one can still sample all at once but mask out unused ones – however, the policy network can be trained with an auxiliary loss to only pay attention to blueprint logits when op=GERMINATE, etc., or use a two-stage sampling. This is a **research-level change** because it alters how PPO experiences are gathered and increases complexity (the logprob of

an action becomes a sum of logprobs of each component, which is already the case, but now some components are omitted in certain branches).

However, the benefit is significant: it reduces the effective action space the agent must search at once and eliminates meaningless combinations. It likely speeds up learning. If time permits, we can prototype a simplified version: e.g. a network that outputs `op` separately, and for other heads we zero out gradients on those outputs in transitions where they weren't used (this way, the network doesn't learn spurious correlations for unused heads).

Feasibility: Moderate. The codebase already has separate outputs per head, so implementing a hierarchical conditioning might require custom sampling logic. Perhaps easier: train as is but post-process the buffer so that for WAIT actions, we don't update blueprint/blend logits (effectively not penalizing whatever random blueprint it had since it didn't matter). That alone might improve learning signal.

- **Auto-Regressive Action Sampling:** A variant of above: sample `slot` first, then condition on chosen slot's state sample `op` (maybe some ops not available if slot empty, etc.), then if op=GERMINATE, sample blueprint and blend, etc. This is like modeling $\pi(slot) * \pi(op|slot) * \pi(params|op,slot)$. It can be implemented via a single network by feeding the already sampled decisions back in (like a sequential policy). This ensures internal consistency of actions. This is more complex to implement with PPO but doable with a custom environment step that asks the policy multiple times per decision. Probably not needed immediately – we rely on masking to enforce consistency for now. This is a research project if the simpler hierarchical gating doesn't suffice.

- **Head-Specific Network Modules:** The action heads have very different semantics. We could give each head (or group of heads) its own small network tower (sharing the lower layers). For example, one could have a part of the network specialized in choosing blueprint (maybe taking as input the slot's blueprint penalty or performance) and another specialized in lifecycle op (taking more of the global training state into account). The current architecture likely shares one LSTM for all. We could consider multi-head attention or context-specific parameters. This is an advanced improvement; not necessary if performance is okay, but might help if we see one decision type lagging.

An easier version: **action masking as input augmentation**. We can feed the mask values into the network as features. For instance, a feature vector that indicates which ops are currently valid, which slots are free, etc. The network could learn to naturally ignore masked actions (since it knows they're invalid). In effect, this is redundant with masking at output, but it could help the network *understand* the game rules better. This is relatively easy: we know in each state which moves are allowed, so append say 1/0 per op validity as part of obs. However, we must be careful not to create a circular logic (the mask is deterministic from state, so it's fine). This is more of a training speed hack.

- **Expand/Refine Action Options:** Some improvements might involve making some discrete actions continuous or higher-resolution, but since the question specifically mentions action parameterization, consider:
- **Alpha Control granularity:** Instead of discrete alpha_target {0.5,0.7,1.0}, perhaps a continuous action or more bins (if needed). A *simple extension* could be adding one more target level like 0.3 or 0.9 if we find the need. But going continuous might be overkill and complicates PPO (would need a Gaussian param, etc.). We probably stick to discrete for now.

- **Tempo as a decision:** If tempo (the number of epochs to run before next action) is a head, we could consider removing it and just always stepping one epoch (the agent can always just choose WAIT multiple times to simulate longer wait). However, having a tempo head is interesting because it allows saying "don't bother me for 5 epochs". If it's implemented, we might improve its usage by *coupling it with stage decisions*. For example, perhaps tempo is only relevant when a seed is blending (like "blend for 5 epochs then check back"). If so, we could condition the tempo head on the op (similar to hierarchical).
- **Blend Algorithm vs Alpha Curve:** If both are present (maybe alpha_curve means the shape of ramp and alpha_algorithm means gating vs continuous), we could simplify by merging them. Possibly "blend algorithm" already captures linear vs sigmoid, and "alpha_curve" might refer to like ease-in/ease-out specifics. We could propose to simplify action space by not exposing too fine-grained options initially – e.g., maybe remove alpha_curve head if it's not crucial, letting the agent focus on more impactful decisions (blueprint, timing). This is an easy patch: mask or fix one of those heads at a default.

- **Action Space Reduction for Feasibility:** If certain combinations are rarely useful (e.g., maybe tempo=8 epochs is too long to ever be optimal, or alpha_speed=0 is trivial), we can either remove those options or at least mask them in situations where they truly make no sense. This reduces the branching factor. For example, alpha_speed=0 (no change) might effectively be a WAIT disguised; if it confuses the policy, we might just drop it or encourage the policy not to use it except certain cases.

- **Blueprint Penalty Integration:** The environment uses a blueprint penalty mechanism (penalize blueprint on cull) [75] [76]. Right now, that likely influences future blueprint availability (maybe by mask or by affecting training signals indirectly). We could explicitly feed the current penalty for each blueprint as part of observation or internally adjust the policy's logits for blueprint head (like bias them). Alternatively, incorporate it in the mask (skip blueprint if penalty beyond threshold [77]). It seems they already plan to skip if penalty >3. We should ensure the policy is aware when a blueprint is effectively "banned" via mask to avoid it trying. This might already be done via mask using `BLUEPRINT_PENALTY_THRESHOLD` [77]. If not, an improvement is to do so or feed that info to the network.

- **Ease-of-Implementation vs Impact:**

- *Easy patches:* one-hot stage, normalized metrics, additional time counters, adding mask info to obs. These can be done with minimal risk and will likely improve learning stability and speed.
- *Medium effort:* hierarchical action gating by simply not training certain heads for irrelevant ops, or adding a small network gating. This requires some careful coding but could yield cleaner learning. It's somewhat moderate in complexity – doable within a few days of work and testing.
- *Major changes (research projects):* full auto-regressive policy, a different neural architecture (like graph-based or attention to treat seeds as a set), continuous action parameters (like directly outputting alpha target as a number) – those would demand more extensive experimentation and aren't guaranteed to outperform without tuning.

We should prioritize the easy and moderate improvements first, as they will likely address the known issues (observations being a bit crude, and action heads producing meaningless outputs sometimes). For instance, implementing the one-hot stage and improvement scaling audit from the TODO [24] is straightforward and directly targets an identified stability issue. Similarly, providing the network with knowledge of which slot/blueprint combos are valid via features will help it not waste capacity figuring that out.

In summary, **immediate changes**: one-hot encode categorical features, normalize all numeric features to comparable ranges, include critical time counters. **Strategic changes**: adopt a form of hierarchical action selection to avoid coupling issues – an easy start is to *structure the mask and loss* so that, e.g., blueprint logits only receive gradient on GERMINATE steps. That's a coding change (ensuring loss=0 for those logits otherwise) but not altering the runtime logic. This alone can prevent the agent from having to "guess" an irrelevant blueprint during WAIT actions (which currently is just noise). **Long-term research changes**: if baseline performance plateaus below expectations, explore a more radical policy architecture that inherently respects the action hierarchy (like an RNN that makes decisions stepwise in a single time-step). Also possibly consider a **meta-controller** approach: one network decides high-level strategy (how many seeds to use, etc.) and another handles low-level control – but that may be overkill here.

We'll implement the easy ones right away in the training code (these do not break compatibility with PPO). We will schedule the more complex ones as future improvements if needed once we see baseline results.

G) Experiment Plan

We propose a series of experiments to validate and refine the training strategy. Each experiment has a clear hypothesis, the changes to test, metrics to track, expected outcome, and stopping criteria for evaluation.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
1. Baseline Shaped (3-slot) *Setup: Default parameters, shaped reward.*	*Hypothesis:* The base configuration with full shaping will learn a reasonable policy that at least matches heuristic performance. This sets a baseline. *Change:* Use RewardFamily=CONTRIBUTION, RewardMode=SHAPED (default). 3 slots, all features normalized except the planned improvements (apply one-hot stage and improvement scaling fixes before run).	- Episode return (average total reward). - Final validation accuracy vs heuristic's. - Policy health: entropy per head, KL divergence, clip fraction. - Behavior: avg # of seeds germinated, # fossilized per episode; action distribution (how often each op). - Reward breakdown: average contribution vs PBRS vs penalties.	We expect the policy to learn to use seeds and complete lifecycles. By ~100 episodes, entropy should start dropping and the agent should outperform doing nothing. Possibly it may not yet beat the heuristic, but it should use at least 1–2 seeds effectively.	**Stop training** when either: - Policy converges (e.g. reward plateaus for ~20 episodes and entropy is low). - OR if after 200 episodes, the agent is still mostly waiting or random (then baseline failed – need fixes). At stop, record the performance and diagnostics as baseline.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
2. Single-Slot Curriculum (Phase1) *Setup: 1 slot, heavy shaping.*	*Hypothesis:* Reducing to one slot makes it easier for the agent to learn the seed lifecycle. It should quickly learn GERMINATE→BLEND→FOSSILIZE with dense reward. *Change:* slot_config = 1 slot; max_seeds=1. Keep RewardMode=SHAPED, maybe increase PBRS weight to 0.5 to strongly reward stage progress. Remove rent penalty entirely for this phase.	- Time to first successful full seed lifecycle (in episodes). - Frequency of germination by the agent. - Stage progression counts (how often reaching blending, fossilizing). - Reward components (should be mostly positive PBRS if done correctly). - Invalid action rate (should be near zero since one slot simplifies choices).	Expect agent to reliably germinate a seed after the host plateaus and then fossilize it. Probably within ~50 episodes it gets this pattern. If it does, that confirms curriculum works. If not, something else is wrong (maybe observations or reward signals confusing it).	Train until success criterion: e.g. >80% of episodes the agent uses a seed. Stop when performance saturates (no further increase in reward for ~10 episodes). If it fails after 100 episodes (no seeds used), stop and troubleshoot (likely reward or mask issue).

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
3. Multi-Slot Introduction (Phase2) *Setup: 2 slots, moderate shaping.*	*Hypothesis:* The agent can extend learned behavior to managing two seeds in parallel. Some shaping (PBRS) will help coordinate multi-seed use. *Change:* slot_config = 2; max_seeds=2. Use shaped reward but drop PBRS weight to 0.2 (since agent knows to germinate now). Enable a small rent penalty (0.01). Telemetry still off. Entropy per head slightly increased for blueprint to encourage exploring using both slots and different blueprint combos.	- Seed usage in both slots: do we see episodes where 2 seeds are active concurrently? - Improvement vs overhead: track final accuracy and growth_ratio (parameters) [18]. - Policy metrics: entropy, KL to see if increasing complexity spikes them. - Action distribution: how often uses second slot, how often prunes one seed while training another. - Invalid actions: e.g. attempts to germinate when both slots full (should be masked, ideally agent avoids it entirely).	Expect the agent to occasionally use both slots, especially when the first seed's improvement plateaus. We should see an increase in total seeds used per episode compared to single-slot. Possibly slight instability at first (more complex decisions), but should stabilize with shaping's guidance (maybe after 200–300 episodes). We anticipate final performance (accuracy) improves over single-slot agent.	Run for enough episodes to reach stable multi-seed behavior or until diminishing returns in reward. Stop criteria: when the average number of seeds per episode stabilizes (~1.5-2 seeds) and rewards plateau, or if training diverges (entropy collapses unexpectedly or reward drops – then stop and adjust before proceeding). Typically ~200 episodes might suffice for this phase.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
4. Blueprint Diversity Test *Setup: 3 blueprint types available.*	*Hypothesis:* Agent will learn to differentiate blueprint effectiveness. If one type consistently yields better improvements, it should favor it. If one is bad, it should avoid after some trials (especially with blueprint penalty in effect). *Change:* Allow say 3 blueprint IDs (like conv, mlp, attention). Possibly implement blueprint penalty on cull (as per constants) to see if agent responds by avoiding failed types. Keep 2 slots for now.	- Blueprint selection frequency vs their empirical performance. - Count of culls per blueprint type (does one type get culled more?). - Final accuracy: does using the better blueprint lead to higher accuracy than using a mix or the worse one. - Reward breakdown by blueprint if we can segregate (or at least per-episode outcome by blueprint used). - Entropy of blueprint head over time (should initially be high then concentrate on best).	Expectation: the agent will try all 3 blueprints early (due to entropy). Suppose conv and mlp do well and attention fails in this environment; we expect it to start avoiding attention seeds after a number of bad outcomes (especially if they are penalized). We might see the blueprint head's distribution shift heavily to the better ones, and the penalty count increase for the bad blueprint, eventually masking it out (if above threshold) [77]. That would validate that it's picking up architectural taste.	Train until blueprint usage stabilizes (i.e. one blueprint dominates or a stable mix). If after X episodes the agent still uses a clearly inferior blueprint frequently, something's wrong (maybe it isn't perceiving difference – could indicate need for telemetry features). Stop once we see a clear preference pattern or if performance stops improving further by trying different blueprints.

5. Add Telemetry Features *Setup: 2–3 slots, telemetry on.*	*Hypothesis:* Providing per-seed telemetry (grad norms, health, etc.) will improve the agent's ability to prune or foster seeds. It will especially help differentiate seeds beyond just immediate accuracy change (e.g. a seed with vanishing gradients can be dropped sooner). *Change:* `use_telemetry=True` in features. All 10 telemetry features per active slot included [72] . No other changes, but note obs dim increases. Possibly need to double LSTM hidden size (from 128 to 256) to handle extra input info.	- Compare decision-making with vs without telemetry: e.g. does the agent prune bad seeds faster? Track **seed lifespan** when gradient_health is poor. - Performance metrics: final accuracy, number of unsuccessful seeds (germinated but pruned) – telemetry should reduce wasted efforts. - Policy inputs usage: check feature importance (maybe via ablation or just correlation). If gradient_health correlates with prune actions after enabling telemetry, that's a good sign. - Any training stability change: adding many features could slow learning or cause initial dip while network learns to use them. Monitor entropy and loss for any anomalies when turning this on.	Expected outcome: The agent's pruning/ culling decisions become more precise. For instance, if a seed's gradient_norm is very low (not learning), the agent should cull it quickly, whereas before it might wait for a drop in accuracy to manifest. Telemetry should thus lead to slightly better final accuracy (by avoiding losing time on hopeless seeds) or at least more efficient use of epochs. We also expect a possible increase in learning time needed (more inputs to learn from), but overall better asymptotic performance.	We would run this for a similar length as prior experiments and compare metrics to the no-telemetry run. Stop criteria: if after adding telemetry the reward or accuracy plateaus or improves relative to previous best. If we see no improvement or even degradation after sufficient time, we may stop and analyze (maybe the agent got overwhelmed – might need feature selection or more training). Otherwise, stop when improvements level off.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
6. Entropy Schedule Experiment *Setup: Dynamic entropy coefficient.*	*Hypothesis:* A tailored entropy decay will yield faster convergence and prevent premature convergence in any head. *Change:* Implement a schedule: e.g. keep entropy high (0.05) for first 50 episodes, then decay to 0.01 by episode 300. Additionally, if we observed any head collapsed early in previous runs, apply a head-specific boost (say blueprint head entropy *1.5 for first 100 episodes).	- Entropy per head over training time (should smoothly decline, no sudden collapse). - Policy KL divergence: with higher entropy early, KL per update might be larger (due to more exploration) but controlled; as entropy lowers, KL should also reduce as policy gets confident. - Final performance vs previous runs: did this schedule improve final reward or stability (maybe fewer spikes in reward curve)? - Exploration outcomes: e.g. did the agent discover use of a third blueprint it previously ignored? (A sign that more exploration helped.)	We expect a more stable training curve: early randomness prevents local optima lock-in, and later focus improves polish. If previously a certain action wasn't explored, now it might be (e.g. agent might try the gated blend algorithm more, if earlier it never did). Ideally, final results improve slightly (or at least variance across training seeds is lower).	Run with the schedule through convergence. Stop when it's comparable length to baseline runs. We'll particularly compare to baseline run (Exp1) to see if learning was faster or final performance higher. If not much difference, we might not need complicated schedules. If clearly better, we'll adopt this approach. Stop early only if we see signs of divergence (which would be surprising, as entropy should only help exploration).

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
7. Sparse Reward Transition *Setup: Switch to sparse after N episodes.*	*Hypothesis:* The policy trained with shaping can maintain performance when switched to a sparse reward (final accuracy-based). Essentially, test if it truly learned to improve accuracy or was just exploiting shaping. *Change:* Take a well-trained policy (e.g. from Exp5 or Exp6) and continue training it but with RewardMode = SPARSE (or SIMPLIFIED) – meaning only terminal reward = final_acc - baseline_acc - param_penalty. Train for some additional episodes.	- Immediate impact on total reward and behavior upon switching. We'll see a drop in reward magnitude (since dense rewards gone), but watch **whether the agent's behavior changes**. Does it still germinate seeds and manage them well, or does it collapse to doing nothing when shaping is removed? - Final accuracy achieved per episode (since that directly drives reward now). - Any signs of instability: value loss might spike initially because reward scaling changed – watch value function convergence. - Comparison of final accuracy distribution before vs after switch.	Expected: A well-trained policy should continue performing reasonably, maybe with some adjustments. We might see a temporary dip as it recalibrates to the new reward scheme, but it should recover and perhaps even slightly improve final accuracy as it fine-tunes for that explicitly. If, however, we see the agent revert to WAIT (inactivity) or weird behavior, that indicates it was too reliant on shaping – a sign we need to anneal more gradually or incorporate some shaping back.	We'd run this for, say, 50-100 episodes post-switch. Stop criteria: if after those episodes the policy's final accuracy is stable (within few % of before) and not trending down, we consider it a success. If it significantly degrades and doesn't recover within, say, 100 episodes, we stop – that means the switch was too abrupt, and we'd try a more gradual anneal instead of a sudden swap.

8. Hyperparameter Sensitivity (A/B tests) *Setup: Vary one hyperparam.*	*Hypothesis:* Some PPO hyperparams might be suboptimal. We test two in particular: (A) a lower clip_ratio (0.1) vs default 0.2, and (B) a smaller γ (0.99 vs 0.995) to see effect on credit assignment. *Change:* For A: run a training with clip_ratio=0.1 from start. For B: run with γ=0.98 or 0.99. Keep others same as baseline.	- Metric for A: KL divergence trajectory and final performance. A lower clip should lead to smaller KL updates; if baseline was fine, too low clip might slow improvement. We check if training becomes more stable (maybe less variance in reward) at cost of speed. Also check final returns and accuracy – they shouldn't be worse; if they are significantly, clipping might be too tight. - Metric for B: Look at credit assignment-related stats: does the agent still learn long-horizon moves? With lower γ, it might put less value on late-stage rewards (like fossilization). Check if agent perhaps fossilizes less or times seeds differently. Compare final accuracy and number of seeds used. - In both cases, monitor	(A) We expect lower clip ratio to result in more conservative updates – possibly fewer instances of ratio > 1.5 or early stops due to KL. If baseline had any instability, this could fix it. If baseline was stable, this might just slow things a bit. We predict policy quality ends up similar, just maybe needing more episodes. (B) Lower gamma likely harms performance in this context (because 25-step horizon needs high gamma). We'd expect if γ=0.98, the agent might undervalue late rewards like final accuracy; it could, say, fossilize seeds less often or do short-sighted things. We might see slightly lower final accuracy or weird timing.	Each sub-experiment we run for a decent number of episodes (maybe 100-200) and compare to same point in baseline. Stop early if we see clear negative outcomes: e.g., for B, if agent completely fails to use seeds or performance is much worse by mid-training, we can stop as hypothesis confirmed that gamma was too low. For A, if training is unbearably slow (like little progress after same episodes baseline nearly solved), we stop and conclude 0.2 was better.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
		convergence speed (episodes to reach certain reward threshold).		

9. Invalid Action Penalty Test *Setup: Add small penalty for invalid attempts.*	*Hypothesis:* Explicitly punishing invalid (masked) actions will encourage the agent to stay within legal moves, potentially improving learning signal. *Change:* Modify reward: whenever an action is chosen that is invalid (i.e., env fell back to WAIT), add a -0.5 penalty. (This requires tracking action_success flag [12] [13] and adjusting reward before normalization, presumably.) Run a training with this modification.	- Frequency of invalid actions vs baseline. Ideally, it should drop to near zero quickly. - Learning speed: compare how fast the agent learns basic operations relative to baseline (if baseline had some struggle with illegal actions initially, this should alleviate it). - Final policy performance: shouldn't be negatively impacted; possibly slightly better if it freed capacity from exploring invalid spaces. - Potential side effect: watch if the agent learns to game this (e.g. always picking the obviously safe WAIT to avoid penalty – but WAIT itself yields 0 reward plus stagnation, so not beneficial overall). Ensure that doesn't happen excessively (if it does, it means penalty is too heavy relative to positive rewards).	We expect a modest improvement in sample efficiency: early in training, the agent will more quickly learn "don't do invalid moves" rather than learning via neutral outcomes. By later training, the penalty ideally becomes irrelevant (agent rarely makes invalid choices). If this works, invalid action rate should approach 0 faster than in baseline. Also, no large difference in final returns except less variance perhaps.	We train for, say, 100 episodes and observe. Stop if we see unintended consequence: e.g., if the agent's op distribution shifts heavily to WAIT at the expense of exploration (could happen if penalty fear outweighs curiosity). If that happens, we'd reduce the penalty magnitude and retry or scrap this idea. Stop when invalid actions are essentially eliminated and policy is on track, no need to run full training if trend is clear by mid-run.

10. Reward Ablation: No PBRS *Setup: Shaped reward minus stage bonuses.*	*Hypothesis:* The agent can still learn without PBRS; it will just rely on direct contribution rewards. This tests if PBRS was crucial or not. *Change:* Set stage_potential_weight = 0 (effectively no PBRS term in reward) in ContributionRewardConfig. Everything else same as baseline (3-slot, telemetry on if that was default by now, etc.).	- Learning curve (episode reward and accuracy) versus baseline: likely slower start since no immediate +1 for germination. Monitor how many episodes until agent begins germinating seeds consistently. - Final performance: does it catch up to baseline after more episodes? If yes, PBRS might not be needed ultimately. - Behavior differences: agent might delay germination more (because now it only germinates when it expects a true accuracy gain). Possibly fewer but more effective seeds. Compare average seed count and average seed contribution. - Critic stability: without PBRS (which is a potential shaping), rewards are more sparse/ delayed. Check if value loss is	We expect the agent will eventually learn to use seeds, but it might take significantly longer or even stall initially (it might not know to germinate until maybe very late in episode when accuracy stagnates – and doing it so late gives little time for reward). Possibly we'll see a dip in performance relative to baseline at equal training time. If after extended training it reaches similar final accuracy, that's interesting: it means PBRS wasn't strictly necessary, just helpful for speed.	We'll run this longer than baseline if needed (maybe 2x episodes) to see if it catches up. Stop criteria: if the agent fails to learn any seed usage after a reasonable time (e.g. still near-zero seeds by episode 100, whereas baseline had many by then), we conclude it's struggling and stop. If it does learn, we continue until convergence and then compare results with baseline.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
		higher or if advantage estimation gets noisy (that would show up as more variance in returns).		

11. Final Evaluation vs Heuristic *Setup: Evaluate trained policy on test runs.*	*Hypothesis:* The trained RL policy outperforms the heuristic (Tamiyo) in final accuracy or achieves similar accuracy with fewer resources. *Change:* Take the best trained policy and run it on e.g. 100 evaluation episodes (with different random seeds for data order, etc.) without further training. Compare with 100 runs of the heuristic on the same task. No training happening, just simulation.	- **Final validation accuracy distribution** (mean, std) for RL vs heuristic. - **Accuracy-minus-rent** (maybe define as final accuracy - κ*total_params% where κ is some constant, to quantify trade-off) for both. - Number of seeds used on average by each and average epochs at which those seeds were germinated/ fossilized. (This shows if RL finds a more efficient schedule.) - Any failure cases: e.g. % of runs where RL policy leads to worse-than-baseline accuracy (did it ever catastrophically fail?). - Compute time or training epochs used: if RL uses fewer epochs to reach similar accuracy that's a plus (maybe it doesn't always run full 25 epochs if it converges early or something).	We expect the RL policy to achieve higher accuracy on average, perhaps by a margin (e.g. if heuristic got 88% on CIFAR10, RL might get 89-90%). Or, if equal accuracy, RL might have done so with fewer or smaller seeds (so model complexity is lower). The RL might also adapt per-run (some runs adding more seeds if needed, some not, whereas heuristic might follow a fixed pattern), potentially giving it an edge in harder random initializations. If results show parity or slight improvement, that's a win given how hard matching a well-tuned heuristic is. If RL underperforms consistently, then either more training or algorithm tweaks are needed.	This is a one-off evaluation rather than continuing training. We stop once we gather enough runs for statistical significance. If RL is worse, we might not "stop" but rather go back and adjust training (maybe extended training or revisit reward/ hyperparams). But assuming we're at final evaluation stage, we collect these metrics and conclude.

Experiment	Hypothesis & Change	Key Metrics to Monitor	Expected Outcome	Stopping Criteria
12. Ablation: Telemetry Off vs On (Trained) *Setup: Remove telemetry inputs from a trained policy.*	*Hypothesis:* To see how much telemetry features contributed, we can ablate them at test time. If removing them degrades performance significantly, it shows the policy was leveraging that info heavily. If not, maybe they were not crucial. *Change:* Take the final trained policy (which was using telemetry), and run evaluation episodes where we zero out the telemetry part of observation (or use a version of the policy network that doesn't expect them – might need to train a parallel one without telemetry for fair comp). Compare performance.	- Final accuracy drop when telemetry removed. - Changes in agent behavior: does it prune less effectively without telemetry? Possibly measure: in runs without telemetry, average duration seeds remain in blending vs with telemetry (should increase if it can't tell they're bad). - If feasible, look at the internal policy decisions: e.g. the logits for culling might be different if gradient_health is unknown. - Reward achieved in these test runs.	We expect some drop in efficiency without telemetry, but if the policy was robust, maybe not huge. Telemetry likely helps in edge cases, so average performance might only slightly decline, but worst-case scenarios (like a seed that is clearly going to fail due to vanishing gradients) might not be handled as promptly. This ablation confirms the value of those 10 extra features. If we find they don't change much, perhaps the agent was mostly using global metrics anyway (maybe because the host's accuracy already reflected issues).	This is an evaluation, so just run enough episodes to gauge difference (say 30 runs with and without telemetry info). No training needed. Stop once metrics collected. A significant difference would prompt us in future to ensure telemetry stays in use; a negligible difference might mean we can simplify the model by dropping telemetry in final deployment (less overhead).

(The above experiments should be performed in a logical sequence: we'd start with baseline (Exp1) and curriculum (Exp2,3), gradually adding complexity (Exp4,5). We'd incorporate improvements and tuning

(Exp6,8,9) as we progress. Finally, we test generalization and performance (Exp7,11,12). Each experiment's outcome will inform adjustments for subsequent ones.)

Throughout all experiments, we will log the detailed metrics as described to diagnose the policy's learning behavior. We use multiple seeds (random initializations) for key experiments to ensure results are consistent and not due to lucky or unlucky randomness. Stopping criteria are mainly about seeing convergence or clear failure signals to avoid wasting time.

H) Instrumentation Checklist

To effectively diagnose training and agent behavior, we will instrument the code to log a comprehensive set of metrics and traces:

- **Reward Component Breakdown:** For each environment step (or at least each episode), log the values of all sub-rewards. This includes:
- *bounded_attribution (counterfactual reward)*,
- *proxy_improvement reward* (if used),
- *PBRS stage bonus* added,
- *blending_warning penalty* [30] ,
- *probation_warning penalty* [62] ,
- *compute_rent penalty* (and growth_ratio) [19] [18] ,
- *any action cost penalties*,
- *terminal bonus or final accuracy reward*.
  Many of these are available via the `RewardComponentsTelemetry` structure in code, so we'll use `emit_reward_summary` or similar hooks [78] [79] to capture them. This breakdown per step (or averaged per episode) will tell us what's driving the total reward. We will especially monitor if any single component consistently saturates (sign of potential reward hacking or imbalance).
- **Per-Head Action Distributions:** Every few training updates (or every episode), log the distribution of actions for each head:
- *Slot head:* how often each slot index was chosen (as % of decisions).
- *Op head:* percentage of decisions that were WAIT, GERMINATE, PRUNE, FOSSILIZE, etc.
- *Blueprint head:* distribution over blueprint types chosen (especially in germination ops).
- *Blend head:* distribution over blend algorithms used.
  This can be done via telemetry events or a custom logger that inspects the agent's action outputs (the code has an `emit_action_distribution` in telemetry emitters [80] which likely already does something similar). We also track *conditional distributions* — e.g. blueprint distribution given op=GERMINATE (since blueprint is only relevant then). That helps identify if the policy properly branches (e.g., maybe blueprint choice is uniform overall but given it germinated, it actually always picks a certain one).
- **Action Mask Usage Rates:** Log how often each head's mask is active and when it influences the choice:
- E.g. "Mask hit rate – slot: X% (meaning X% of time a sampled slot was invalid and had to be corrected), op: Y% (if any ops masked), blueprint: Z%." Actually, since masking prevents selection, we can measure how many options are masked at a time. Better: use the data the code collects: it accumulates `mask_hits` for each head [23] . We will output for each head the fraction of steps where at least one invalid action in that head was masked. Ideally, over training, these fractions should either be constant (if environment always masks some combos like certain blueprints in CNN) or decrease (if agent learns to not choose illegal combos).

- Also count *fallback to WAIT occurrences* (when an invalid action was taken and the env defaulted to WAIT). We can detect that via `action_success=False` combined with op not WAIT [12]. Logging the rate of this happening will help see if the policy is getting stuck producing invalid actions.
- **PPO Training Statistics:** For each PPO update (each batch of episodes):
- *Policy loss, value loss, total loss*.
- *Approximate KL divergence* for the update (the code likely computes this and checks vs target_kl [81]).
- *Clip fraction* (what % of gradients were clipped by the PPO clamp).
- *Entropy values* for the policy, possibly per head. We can compute entropy of each categorical head's distribution averaged over the batch (the MaskedCategorical likely can give this). This tells us if any head's exploration is collapsing (e.g. blueprint entropy dropping to near 0 while others not).
- *Value function explained variance:* after each update, evaluate how well value predictions match returns. Many implementations log this. It's crucial to gauge critic quality. A high explained variance (close to 1) is good; low or negative means value is off.
- *Learning rate (if it's being annealed or scheduled)*, to keep track of current LR.
- *Grad norm* (average or max) before clipping, to see if we hit the max_grad_norm often. These stats can often be retrieved from the PPOAgent's training loop or by instrumenting the optimizer steps.
- **Episode Outcome Metrics:** At end of each episode in each env, log:
- *Final validation accuracy* achieved.
- *Final training accuracy* (to see if maybe overfit, etc.).
- *Number of seeds germinated in that episode*, *number of seeds fossilized*, *pruned*, etc. (We can get these from env state counters [82] [83]). This effectively logs the agent's "design choices" for that run.
- *The sequence of actions taken* (maybe store a compact history of op and when it happened). This is useful for offline analysis of strategy. If logging every action is too verbose, summarizing as counts or key events is fine.
- *Total reward* of the episode (normalized and maybe raw unnormalized sum).
- *Episode length* (in epochs, though it's usually fixed at 25 unless early termination happens due to rollback).
- **Policy Behavioral Metrics:** These overlap with above but explicitly:
- *Invalid->WAIT ratio:* count how many actions were invalid leading to no-ops vs total actions. Should trend down.
- *Average stage occupancy:* e.g. average number of seeds concurrently in TRAINING/BLENDING at any given time. This can be derived from logs of how many seeds were active each epoch. We could instrument environment to log at each epoch "active seeds = N, stages = {count per stage}". Summarize per episode. This shows if agent tends to run with, say, always 1 seed at a time or sometimes 2, etc.
- *Seed utilization timeline:* For insight, log something like a timeline of seed alpha values or contributions per epoch. Perhaps simpler: at the end of episode, log for each seed that was introduced: its blueprint, how long it stayed, final contribution. This helps correlate decisions with outcomes.
- *Governor events:* If any catastrophic event triggers (Tolaria governor rollback or panic) [84], log that along with reason. These should be rare (they indicate training instability), but if they happen we need to know (since the agent would get a big punishment). So include in logs any TelemetryEvent of type `GOVERNOR_PANIC/ROLLBACK` and its data.
- **Performance Metrics for Final Evaluation:** When running test episodes (no learning), log:
- *Final accuracy, baseline accuracy (initial model accuracy)*, and compute *accuracy gain*.

- *Parameter count at end* (to compute any accuracy-minus-rent measure like accuracy minus 0.1*%param_increase or similar).
- *Perhaps inference time or steps – though likely not needed unless seeds affect runtime).

- *Compare those with heuristic runs side-by-side.* (This is more analysis than instrumentation, but ensure we record necessary info for that comparison).

- **Debug/Anomaly Logs:** Enable logging or even triggering debug mode if:

- *Entropy collapse:* If any head's entropy falls below the warning threshold (0.3 of max) [21], log a warning event (the code might already do this). We'll capture it.
- *Ratio explosion/collapse:* The code has thresholds for ratio >5 or <0.1 [58]. If those triggers fire (perhaps via `RatioExplosionDiagnostic`), log when and maybe save the offending batch for inspection.
- *Value function anomalies:* If value loss is very high or value predictions nan/inf, log that. (We might also implement a simple check: if any value estimate > some large threshold, log instability.)
- *Numerical stability:* Monitor if the RunningMeanStd or RewardNormalizer produce any nan (shouldn't, but if observation has some pathological values, could). Possibly just assert or log at high severity if any nan in obs or grad.

- *Telemetry events from code:* The Telemetry system lists events like REWARD_HACKING_SUSPECTED, etc. [85]. We should subscribe to those if they occur. For example, if an event "REWARD_HACKING_SUSPECTED" triggers (maybe they have some heuristic for that), we immediately know to inspect what the agent did. Logging these events with details can greatly speed debugging.

- **LSTM/Memory Diagnostics:** If we suspect memory issues, we can log the LSTM hidden state norm at the start and end of episodes to see if it saturates or decays. Also maybe monitor if the policy's value of certain states that should be similar diverge if fed with different histories (harder to do live, but maybe in test we can feed same observation with different hidden states to see effect).

- **Logging Frequency:**

- Some metrics like reward components and actions we can log *every epoch (env step)* in debug mode, but that's a lot. Instead, accumulate and log summary per episode to keep logs manageable (with occasional full detail episodes for analysis).
- PPO stats we log every update (which might be every few episodes * n_envs).

- We will also use TensorBoard or similar to plot key metrics over time (episode reward, accuracy, entropy, etc.).

- **Checkpoints and Reproducibility:** Regularly save model checkpoints (maybe every X episodes or when performance improves). Also log the random seeds used for each run. This helps if we need to reproduce a specific failure or anomaly run.

- **Instrumentation of Code Paths:** Ensure that critical code paths have logging:

- When a seed is germinated or fossilized by the agent, log an event (including which blueprint, slot).

- When a seed fails a gate (like trying to fossilize but gate criteria not met [86]), log that along with the reason (not enough improvement, etc.). This is important: if the agent is issuing FOSSILIZE too early, we want to see those failures in logs clearly.
- When the governor rollback happens (if at all), log entire state if possible (since that's a catastrophic event).

- Masking logic: maybe log if a particular blueprint was masked due to blueprint_penalty or topology (so we know agent wasn't even allowed that).

- **Example Log Snippet (for a single episode):**

- Episode 42 (seed=123): FinalAccuracy=0.894 (+4.0% from baseline), SeedsGerminated=2, Fossilized=1, Pruned=1. TotalReward=+8.5.
  Actions: [Epoch5: GERMINATE(slot0, blueprint=conv_light), Epoch5-10: TRAINING, Epoch10: BLENDING started, Epoch13: GERMINATE(slot1, blueprint=attention), Epoch15: PRUNE(slot1), Epoch18: FOSSILIZE(slot0)]
  RewardComp: attribution+6.2, pbrs+1.5, rent-0.3, blend_warn-0.5, term_bonus+1.0.
  Masks: blueprint 'attention' masked after penalty at epoch15.
  PPO: entropy_slot=0.5, entropy_bp=0.2, KL=0.01, V_expl_var=0.65.

This kind of consolidated log (not necessarily this verbose in console, but recorded in data) allows us to trace what happened and why reward was what it was.

In summary, we will **log everything from high-level outcomes to low-level anomalies**. The crucial ones listed – reward breakdown, action distribution, mask usage, PPO stats, and stage transitions – will ensure we can pinpoint if the agent gets stuck (e.g. logs would show it never fossilizes or always masks something) or if a particular reward term is causing unintended behavior. All logs will be time-stamped and episode-indexed for correlation. Using this instrumentation, we can iteratively debug and tune the training process, confident that we'll catch both subtle issues (like slight reward imbalance) and major failures (like entropy collapse) promptly.

Recommended Baseline Config

Using the analysis above, here is the **baseline training configuration** we suggest starting with:

- **Environment & Episode:** 3 slots enabled (full capacity), episode length = 25 epochs (default) [14], parallel environments = 4 [59] for stability. Use a standard task like CIFAR-10 (TaskConfig for CNN) as the training scenario.
- **Observation Features:** Include base training metrics (loss, accuracy, trends) and per-slot features as defined [5], with improvements:
- Stage encoded as one-hot (8-dim for stages 0–7).
- Accuracy values normalized 0–1 (not 0–100).
- Improvement feature scaled (divide by 100) to ensure ~[-1,1] range for typical values.
- Total parameters feature expressed as log10 or normalized fraction of baseline.
- Seed utilization (active seeds/ max_seeds) included [87].
- If using telemetry later, plan to append those, but baseline can start with `use_telemetry=False` to keep it simple initially.
- **Action Space:** All four fundamental heads active (slot, blueprint, blend_alg, op) [4]. Additional heads (tempo, alpha_target, etc.) can be **disabled initially** (e.g. fix tempo to a default like 5 epochs) to reduce complexity; we focus on lifecycle and blueprint decisions first. Masking is on for impossible actions (with `MASKED_LOGIT_VALUE = -1e4` as set [88]).

- **Policy Network:** Factored LSTM actor-critic with input dim ~74 (post feature improvements), one hidden layer of 256, LSTM with 128 units [56], and separate linear outputs for each action head and value. Use orthogonal initialization for outputs (standard for PPO) and ReLU/Tanh activations internally.
- **PPO Hyperparams:** Learning rate = **3e-4** (Adam optimizer) [47]; Clip ratio = **0.2** [89]; Target KL = **0.015** [36] for early stopping; GAE λ = **0.97** [35]; Discount γ = **0.995** [34]; PPO epochs per batch = **10** [50]; Minibatch size = **64** [49] (so each update processes ~100 steps * 4 envs in ~2 minibatches * 10 epochs = 20 passes of each sample); Value loss coef = **0.5** [51]; Max gradient norm = **0.5** [52]. No weight decay initially (0) [36]. Use reward normalization for returns (std-only) [90] and observation normalization (momentum 0.99) [26].
- **Entropy Regularization:** Initial entropy coefficient = **0.05** [22], with a linear decay to **0.01** by halfway through training (to ensure convergence). Apply this globally, but we will monitor per-head entropy – we anticipate blueprint head naturally has lower entropy as it has more categories, but we won't explicitly separate weights unless needed. (We keep the option if one head collapses prematurely.)
- **Reward Function:** Start with **Contribution family, SHAPED mode** [39]. Set `contribution_weight = 1.0` and `proxy_confidence_factor = 0.3` (so proxy weight 0.3) [91]. PBRS stage shaping on with `pbrs_weight = 0.3` [61]. Compute rent on with a low weight: `compute_rent_weight = 0.01` (to slightly penalize parameter bloat) [44] and max_rent_penalty maybe 5.0 (ensuring it never exceeds that) [92]. Blending warning and probation warning enabled at default values (blending_warning starts at -0.1 escalation [30], probation_warning -1 *3^(n-1) up to -10) [62]. Terminal bonus: ensure a small bonus for fossilized seeds (DEFAULT_FOSSILIZE_TERMINAL_SCALE = 3.0, which multiplies ~0.5 base = +1.5 if a seed fossilizes successfully at end) [68]. This shaped setup should provide frequent feedback while still ultimately rewarding accuracy gains.
- **Curriculum:** Initially, do a short warm-up with a simplified scenario (e.g. 1 slot for 20 episodes) to ensure the PPO loop is working and the agent learns to do something non-trivial. Then quickly scale to the full 3-slot environment for the bulk of training. We might maintain a shorter episode length (like 15 epochs) for the first 100 episodes, then increase to 25 for full runs – but this is optional. The baseline will assume full length from start if we skip curriculum for simplicity, but be ready to apply it if we see the agent struggling to get off the ground.
- **Logging & Evaluation:** Use the instrumentation checklist: log reward components, action distribution, etc., to TensorBoard or files every episode or update. Evaluate the policy periodically (every 50 episodes) on a validation set or by running a deterministic episode to see how it actually behaves.
- **Training Length:** Plan for on the order of a few hundred episodes of training. Since each episode is 25 epochs of training data, this is a significant amount of environment interaction, but necessary for convergence. We watch for plateau in performance to decide when to stop.

This baseline config is a conservative starting point – heavily shaped rewards, modest entropy, default PPO stability settings. It should learn the task in a reasonable time without collapsing. From here, we'll iterate improvements.

Recommended Next Two Iterations

After establishing the baseline, we propose the following next iterations to further improve and refine the policy:

Iteration 1: Gradually Reduce Shaping and Introduce Telemetry
Goal: Transition the policy from relying on heavy shaping to optimizing actual performance, and give it more informational tools (telemetry) to make fine-grained decisions.

- **Anneal Reward Shaping:** Over the next training run, slowly lower the PBRS and proxy weights and move toward a sparse reward. For instance, after baseline convergence, run another 100 episodes where `pbrs_weight` is decayed from 0.3 to 0.1 to 0 (linearly every ~30 episodes). Simultaneously, reduce blending/probation penalties if they proved too aggressive (based on baseline observation; e.g. if agent was culling seeds perhaps too early due to warnings, ease them). By the end of this iteration, the agent should primarily be getting reward from actual accuracy improvement (counterfactual and final outcome) rather than shaping. Monitor that its performance (final accuracy) does not drop – ideally it should even improve as it focuses on real objective.
- **Activate Telemetry:** Midway in this iteration (or from start if baseline is stable), enable `use_telemetry=True` so the agent now sees the 10-dim seed telemetry features [6]. This addition will help it discern seed quality. Because the policy is already somewhat trained, it can incorporate new features without losing all progress (the RunningMeanStd will adapt to new dims). Continue training so it can refine its pruning and advancement using signals like gradient health. Pay attention to any immediate transient effect (there might be a slight dip as it explores the new info). By the end, we expect the agent to make smarter decisions (e.g. terminate seeds that show low gradient_health even if their impact hasn't hit accuracy yet).
- **Tuning:** Possibly raise the entropy floor a bit when introducing telemetry to encourage the agent to experiment with using those signals (since it's like giving it new senses, a bit more exploration could help incorporate them). Also, watch value loss – new features can shift the value function; if needed, do a few extra critic updates or lower LR temporarily to let value re-fit.
- **Outcome:** At the end of Iteration 1, we should have a policy that uses nearly fully **intrinsic rewards** (actual contribution) and the rich observation space. This policy should be less prone to any reward hacking, since shaping has been minimized. It should also be more robust – telemetry should reduce variance in outcomes by handling bad seeds consistently. We will evaluate it versus the heuristic on validation: expecting it to beat the heuristic now, as it has fine-tuned control and still retains all it learned with shaping assistance.

Iteration 2: Enhance Action Space Usage and Fine-Tune Hyperparameters
Goal: Push the performance further by leveraging the full action space (like alpha/tempo controls) and fine-tuning PPO parameters for stability and efficiency.

- **Enable Advanced Actions:** If not already, turn on the additional action heads:
- **SET_ALPHA_TARGET / Alpha controls:** Allow the agent to explicitly set seed blend targets and speeds. With a well-trained base policy, it can experiment with these to potentially improve integration smoothness (e.g. set a lower alpha target initially for a risky seed to reduce shock). We will enable the `alpha_target` and `alpha_speed` heads with discrete options as designed (target {0.5,0.7,1.0}, speed {0,3,5,8}). We'll need to incorporate their effects in the environment if not already (ensuring the model honors these commands). We predict the agent might not use them optimally right away, so:
    - **Curriculum for new heads:** Initially, maybe restrict their usage: e.g., allow SET_ALPHA_TARGET op but only after a seed has blended for a few epochs, to give the agent a clue when it's relevant. Or start with alpha_target always =1 (current behavior) and then free it to vary once it's comfortable. Because the agent has telemetry (including alpha and gradient signals), it can learn scenarios like "if adding a seed causes loss spike, maybe set a lower target and slower speed to ease it in."

- **ADVANCE op:** If this op (force advance to next stage) is implemented or meaningful, allow it. It basically tells Kasmina to skip waiting and push a seed to fossilize. The agent could use this if it believes the seed has done all it can. With telemetry and contribution info, it might decide to ADVANCE early some seeds. We have to ensure gating conditions (min improvement etc.) still apply, but agent might learn not to call it unless conditions nearly met (to avoid wasted action). Watch invalid usage; maybe treat ADVANCE similar to how heuristic uses it (only from BLENDING to PROBATIONARY perhaps).
- **Tempo head:** Let the agent pick 3,5,8 epoch tempo if available. This could allow it to say "I'm confident, don't disturb for 8 epochs" or "check again soon in 3 epochs". Evaluate if it uses this effectively. If not, it might default to some median. If it oscillates tempos unnecessarily, we might impose a bit of regularization (like penalty for too short or too long selection to encourage optimal middle). These expansions will increase the action space to the full 9-head vector. Because the base policy is strong, we expect it to incorporate these gradually. We might need to bump up exploration on these new heads initially (like an entropy boost just for them) so the agent tries non-default values.
- **Hyperparameter Fine-Tuning:** Using logs from previous iterations:
- If we saw the policy often hitting the KL limit or oscillating, consider reducing `learning_rate` to 2e-4 or 1e-4, or reducing PPO epochs to 5 for more gradual updates.
- If entropy is too low on some new heads (e.g. agent never uses ADVANCE), add a small intrinsic reward or higher entropy weight for using those (just at start) to encourage exploration.
- Tune `value_coef` if critic lagged – for example, if advantage estimation was noisy, increase to 0.75 to train critic more. Or if critic seems to dominate (value loss >> policy loss and policy improves slowly), maybe lower to 0.4.
- Adjust `max_grad_norm` if we suspect it's clipping too often (check grad norm logs). Possibly raise to 1.0 if gradients were mostly small anyway, to slightly speed learning.
- Set a lower `target_kl` if we want even safer updates (like 0.01) or observe high variance; or slightly higher if training is very slow and stable (maybe 0.02).
- Consider using *adaptive KL* (some implementations adjust LR based on KL divergence) – if needed, we can manually do it: e.g., if KL < half target for many updates, maybe we can increase LR a bit to speed up, and if KL >> target, reduce LR or clip more. We will apply these tweaks gradually and observe their effect on stability and performance. The aim is to maximize final reward (which correlates to final accuracy minus costs) without sacrificing stability.
- **Extensive Testing:** During this iteration, do more test rollouts of the current policy on unseen data or different tasks (if transferring to a new dataset is a goal). This checks generalization. Log whether the policy handles variations well or if it seems overfit to specifics of training runs. For instance, try the policy on a slightly longer episode (say 30 epochs) to see if it generalizes beyond 25 – if it degrades, maybe it was timing-specific.
- **Outcome:** After Iteration 2, we expect the policy to fully exploit the action space to fine-tune training: maybe it uses slower blending for large seeds (via alpha_speed) to avoid shock (fulfilling the "alpha shock mitigation" design), calls ADVANCE to quickly finalize seeds that have plateaued, etc. The hyperparam tweaks should have stabilized any remaining wobbles (ensuring monotonic or at least consistent improvement). We project this iteration yields a policy that convincingly **outperforms the heuristic** in both accuracy and efficiency. For example, it might achieve an extra 1-2% accuracy or use 30% fewer parameters for the same accuracy – a tangible gain from intelligent control.

Finally, with these iterations, we will have a robust training strategy. To summarize, the first iteration focuses on removing crutches and adding state insight (telemetry) – making the policy autonomous and well-informed. The second iteration opens up more action possibilities and polishes the training hyperparameters – making the policy more powerful and stable. Each iteration will be evaluated against

the heuristic baseline to ensure we’re moving in the right direction, with adjustments as needed based on the rich telemetry and logs we collect.

---

1 2 5 17 24 45 69 70 87 features.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/tamiyo/policy/features.py

3 10 11 factored_actions.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/leyline/factored_actions.py

4 14 16 21 22 28 29 34 35 47 49 50 51 52 53 56 58 59 61 68 75 76 77 88 89 __init__.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/leyline/__init__.py

6 7 71 72 85 telemetry.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/leyline/telemetry.py

8 9 25 26 27 37 38 90 normalization.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/simic/control/normalization.py

12 13 23 78 79 80 82 83 84 86 vectorized.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/simic/training/vectorized.py

15 18 19 20 30 31 32 39 42 43 44 46 48 60 62 63 64 65 66 67 91 92 rewards.py
https://github.com/tachyon-beep/esper-lite/blob/be0d59d30d70cae15df1d1a7195f84c035b1f8f0/src/esper/simic/rewards/rewards.py

33 36 40 41 54 55 57 73 74 81 ppo.py
https://github.com/tachyon-beep/esper-lite/blob/00e573eda0ce54ed2bf52327b5b2591a8113fd6e/src/esper/simic/agent/ppo.py