ChatGPT

Tamiyo Next: Obs V3 + Policy V2 Implementation Review

Below we evaluate each phase of the 8-phase implementation plan, identifying potential bugs or oversights and listing concrete validations the developer should perform. These checks focus on correctness (off-by-one errors, dtype mismatches, masking logic, buffer writes), PPO architectural consistency (e.g. action–value alignment, conditioning), state tracking (to avoid silently stale fields), and rollout/bootstrapping schema changes. We also flag behaviors that might "appear to work" but could misbehave if not carefully validated, and suggest high-payoff quality improvements not explicitly in the guide.

Phase 1: Observation Pipeline Integration (Obs V3)

- **Feature Schema and Indexing:** Verify that the new observation features (Obs V3) are correctly sized and indexed. In particular, check that the base feature length (23) plus `num_slots * 39` per-slot features matches the expected input dimensions. An off-by-one error here would misalign the policy network. The code defines 23 base + 3×39=140 by default [1] – confirm this matches the network's `state_dim`. Also validate that every feature index corresponds to the documented metric (e.g. indices 2–4 are clamped losses [2] and 5–7 are accuracies [3]). A mismapped index (even if it runs without crashing) could silently skew the policy's perception.

- **Stage Encoding Validity:** Perform a dry-run where the `SeedStage` values are all within the valid range, ensuring the stage one-hot encoding never hits the fallback path. The code expects `_stage_to_one_hot` to handle known stage IDs and uses an all-zero vector if an unknown stage appears [4]. **This should "never happen" after initial validation** [4], so actively test that every seed's `stage` is in `_VALID_STAGE_VALUES`. Enabling the debug check (`ESPER_DEBUG_STAGE=1`) will assert on any out-of-range stage [5] – a failure here indicates a bug in how stage is set or reported. Catching this early prevents silent downstream errors where a seed's stage features would be all zeros (appearing as "no progress" to the policy).

- **Blueprint ID Mapping:** Check that the blueprint string IDs align exactly with the one-hot indices expected by the policy. The implementation uses a hard-coded `_BLUEPRINT_TO_INDEX` dict for performance [6]. If the set or order of blueprint IDs in `BlueprintAction` changed, this mapping might be off (e.g. a new blueprint type could shift indices). A quick validation is to compare `_NUM_BLUEPRINT_TYPES` (13) and the highest index used in the enum. A mismatch would cause the policy to encode the wrong blueprint or leave the one-hot vector empty for a valid blueprint [7]. This kind of bug might not crash the code (the one-hot would simply be all zeros or wrong slot), but it breaks the correspondence between the chosen blueprint action and the observation – a subtle correctness issue. An **easy quality win** is to add an assertion that the dict covers all enum values (or regenerate it from the enum) [6].

- **Data Type Consistency:** Ensure that all observation fields use consistent types and ranges. For example, `epoch` and `global_step` are cast to float32 in the feature tensor [8], but things like `available_slots` or boolean flags become floats 0.0/1.0 in features. Confirm that masks and categorical indices are dtype-compatible with the policy network. A dtype mismatch can

cause silent precision issues or slow tensor conversions. In particular, the action masks should be boolean or float 0/1 as expected by `MaskedCategorical`. Validate one end-to-end step where `compute_action_masks` produces masks and they are `.to(device)` without unintended type promotion (e.g. bool→uint8). This prevents a scenario where masking "appears" to work but some actions aren't truly blocked due to type or device mismatch.

- **Telemetry Off/On Paths:** If `use_telemetry` is toggled, verify that feature dimensions adjust correctly (telemetry adds per-seed metrics, e.g. +26×3=78 extra dims, total 218 [9]). An off-by-one in the telemetry feature assembly could corrupt the input vector. While telemetry is likely off during training for performance, running a short rollout with telemetry enabled (setting `use_telemetry=True`) and checking tensor shapes will catch any latent indexing bugs. This is a low-frequency code path that could *appear* fine until someone enables debug telemetry and gets a dimension mismatch.

Phase 2: Normalization & Task Configuration

- **Task Config Calibration:** Validate that the `TaskConfig` constants (loss ranges, max_epochs, etc.) match the actual training task. The implementation defines presets for CIFAR-10 vs TinyStories [10] [11]. If the training scenario differs (e.g. a new dataset or a different max epoch count), using the wrong `baseline_loss` or `max_epochs` could skew normalization. For example, if `max_epochs` is underestimated, features like `epochs_remaining` (if any) or normalized epoch counts will saturate too early. Before training, ensure the chosen `TaskConfig` (passed into Tamiyo) reflects the current run's expected ranges (loss values, epoch count, etc.). This prevents subtle scaling issues where normalized improvements or epoch fractions are off (no outright error, but the policy might behave inconsistently if, say, it thinks 50 epochs is full training when actually 100 are planned).

- **Observation Normalizer Updates:** Confirm that the observation normalizer (running mean/std) is being updated at the appropriate times. The code collects `raw_states_for_normalizer_update` each epoch [12] using the *detached* state tensor (so that normalization stats don't affect gradients during rollout). It's critical to call an update method on the normalizer with these raw states periodically (e.g. at the end of each rollout or episode) – otherwise the normalizer's statistics remain frozen and new observations could drift outside its range. This might not throw errors but would degrade performance (policy sees unnormalized inputs if stats are stale). A validation step is to track the normalizer's mean/std values over training and ensure they converge (or at least change) as more epochs' data is collected. If they stay constant after initialization, it means the update path was missed. In short, **verify that `obs_normalizer.update(...)` is invoked with the cached raw states** at the appropriate interval (perhaps after each PPO update). This ensures that normalization keeps working as training evolves, maintaining stable input scales [13].

- **Clamping and Bounds:** The implementation clamps certain metrics (e.g. losses to ±10, improvement percentage to ±10 points) before normalization [2] [14]. Validate these bounds against the actual data distributions. For example, a language modeling loss might exceed 10 (natural log of large vocab), and clamping at 10.0 [2] would cause many values to sit at the cap. If so, update the clamp range (or the TaskConfig baseline_loss) to avoid constant saturation. Similarly, improvement percentages are clamped to ±10 points [14] – confirm that in practice improvements per epoch rarely exceed this (for CIFAR-10 they shouldn't, but for a highly volatile metric they might). An out-of-range value would just be clipped (so the code runs), but it *silently* loses information (e.g. a huge loss spike or gain would just register as "maxed out"). By logging a

few raw values and their normalized results, the developer can catch if clamping is frequently hitting the limit and adjust accordingly for more faithful representation.

- **Global Step Calculation:** Double-check the formula for `global_step` in the TrainingSignals. Currently, `global_step = epoch * num_train_batches` is used [15], which implies counting **after** finishing the epoch's batches (epoch 1 gives `global_step = num_train_batches`). If the training loop defines global step differently (e.g. starting from 0 at epoch 1 start), this could be off by one batch. It won't crash anything, but any logic that uses `global_step` (e.g. for learning rate schedules or telemetry) might be slightly mis-synced. It's worth printing `signals.metrics.global_step` for the first couple of epochs to ensure it aligns with expectation. Off-by-one here is minor, but since Phase 2 is about getting normalization and tracking right, it's a quick sanity check.

Phase 3: Policy Network & Multi-Head Actions (Policy V2)

- **Action Masking Logic (Hierarchical Heads):** Validate that the action masking handles inter-head dependencies correctly. The policy has multiple heads (slot, blueprint, style, tempo, etc.) and not all heads apply to every decision. For example, when the operation (op head) is "WAIT", the other heads (slot, blueprint, etc.) are effectively no-ops. The implementation uses a **causal masking scheme** during advantage computation to ensure heads with no effect get zero advantage [16] [17]. However, we must also ensure the *sampling* logic doesn't produce illegal or inconsistent combos. Check that `compute_action_masks()` yields masks such that, for instance, if `op=WAIT` is chosen, then any blueprint or seed-selection is ignored or constrained to a dummy "noop" value. The code defines a `"noop"` blueprint ID (index 0) in the mapping [18], which suggests that when blueprint isn't needed the agent should select index 0. Confirm by testing a scenario where no germination is possible (no free slot): the blueprint head's sampled index should always be 0 and the style/tempo heads likewise should pick a neutral default. If you find that in such a scenario the policy sometimes outputs a non-zero blueprint index, that indicates the mask isn't properly preventing meaningless actions. This can *appear* to work (since the environment ignores blueprint when op=WAIT), but it means the blueprint head is flapping around unpredictably, which could inject noise or bias. It's acceptable for the blueprint head to output arbitrary values when not used (since advantage=0 there), but ideally the mask should force a stable choice (noop) to reduce variance.

- **Advantage Attribution per Head:** The PPO implementation splits the advantage across heads so that each head is only trained on outcomes it influenced. Confirm that this **advantage masking is correct for all ops**. The advantage logic specifies, for example, that the blueprint head gets the base advantage only when `op==GERMINATE` [17], the slot head gets advantage for any op except WAIT (including PRUNE, FOSSILIZE, ADVANCE) [19], etc. Pay attention to edge cases: ops like FOSSILIZE and ADVANCE should give zero advantage to heads like blueprint, style, etc., which appears to be handled by the masks (neither germinate nor set-alpha, so those masks are false). One area to validate is the **"ADVANCE" op**: by design it uses only the slot head (which seed to advance), but we should confirm that style/alpha heads truly get zero advantage. Since the code didn't explicitly list an `is_advance` mask, it relies on the default `~is_wait` for slot and the absence from other masks to cover ADVANCE. This works implicitly – ADVANCE is not WAIT, so slot_advantage = full advantage, and it's not germinate/set/prune, so blueprint/style/alpha advantages all become 0. A quick unit test: force an ADVANCE decision in a controlled setting and ensure the policy gradient update only affects the slot head and op head (others should see zero gradient). If any head that *should* be masked isn't, it would learn from noise and potentially destabilize training (e.g. the alpha_speed head shouldn't get credit during FOSSILIZE

decisions). The code logic looks consistent, but it's complex enough that a controlled validation is worthwhile.

- **Joint Policy-Value Consistency:** Check for any **mismatch between the policy's factorization and the value function's conditioning.** The value head outputs V(s) (state-value) unconditioned on the chosen action, while the policy is making a structured decision (s → op + other heads). PPO's theory assumes V(s) is a baseline for all actions from state s. This works as long as the state features include all relevant info. One subtle risk is if the policy's decision has components that significantly determine return but aren't reflected in state until later. For example, if the value function doesn't "know" which blueprint was chosen this step, can it still accurately predict the return? In our setup, the new seed's blueprint will affect future loss/accuracy, but the blueprint choice doesn't influence *immediate* rewards except via what happens in future epochs (which V(s) at the time of decision is trying to estimate). This is inherently a hard prediction, but not a bug – just the nature of an unconditioned critic. We want to ensure no coding bug exacerbates this (like accidentally feeding the next state or action info into the value head). The architecture uses a shared LSTM and separate heads [20] [21], so we're fine there. What the developer should do is monitor for any systematic bias: e.g. check if certain ops consistently lead to big positive or negative advantages – if so, the baseline might be misestimating for those. One concrete validation: evaluate the trained value function on states where a germination just happened vs. not happened (all else equal). The value shouldn't wildly mispredict in one case. If it does, that may indicate the need for better conditioning or reward tuning. This is not a immediate bug but an architectural check to ensure the **PPO invariant (advantage = Q(s,a) - V(s)) holds reasonably** and that no action has an unseen "side channel" effect that V(s) missed.

- **Entropy and Exploration per Head:** It's easy to overlook whether each action head maintains healthy exploration. Because some heads (like blueprint or tempo) are used only in specific circumstances, they receive gradient updates infrequently. Validate that entropy regularization is indeed preventing those heads from collapsing. The config defines a minimum entropy coefficient [22] and the code tracks per-head entropy history [23]. During training, monitor the entropy of rarely-used heads (e.g., blueprint head entropy when few germinations have occurred). It should remain high (near uniform) if the agent hasn't had cause to tune it much. A pathology to watch for: a head might become nearly deterministic simply due to lack of feedback (e.g., always picking blueprint "conv_light" by default because it never gets a gradient to change that). This *appears to work* (no immediate error), but when a germination eventually happens, the agent would not explore other blueprints, hurting performance. To validate, force a few germination decisions (or examine the first few that happen) and see if the blueprint choices are diverse or at least not locked to a single value early on. If you find a head's entropy dropping to zero long before it's ever used meaningfully, that's a sign the entropy bonus or mask logic isn't functioning as intended. One possible cause could be masked actions: if a head is masked out (no valid moves) in many steps, its entropy might not be computed or rewarded properly. The developer should ensure that *even unused heads preserve entropy by design*. The code comment indicates entropy is normalized per MaskedCategorical internally [24], which is good – just verify it in practice (e.g. blueprint head should stay at max entropy (~ln(N) if N valid) whenever op != GERMINATE).

Phase 4: Signal Tracking & Lifecycle Gating

- **Embargo/Cooldown Enforcement:** Validate that the "cooldown after cull" logic truly prevents immediate regermination of a just-pruned slot. In the new design, **slot availability is determined by** `seed_slots[slot].state is None` rather than simply checking if an active

seed is present [25]. This subtle change ensures that a slot still holding a recently pruned seed's state (e.g. one scheduled to disappear in a few epochs) is *not* counted as available for germination. To test this, simulate a prune action and then the very next epoch's germinate decision. The expectation: the mask should treat that slot as occupied (no germination allowed) until the embargo period passes. Concretely, if `embargo_epochs_after_cull = 5`, one could prune a seed and confirm that for the next 5 Tamiyo decisions, `model.seed_slots[that_slot].state` remains non-None (likely a "PRUNED" stage or similar) so that any germinate attempt is masked out. After the embargo, the state should be cleared, allowing use. A bug to watch for is if the seed state is cleared immediately on prune (making the slot appear free) – the code suggests scheduled prunes will keep it around, but an immediate prune (`speed_steps=0`) might remove the seed at once [26]. In that case, the slot becomes truly empty with no state, and without additional logic the agent *could* germinate a new seed next epoch. That would violate the intended cooldown invariant. The developer should verify how immediate prunes are handled: the plan might rely on the agent choosing a non-zero `alpha_speed` to get a cooldown. If that invariant isn't enforced by design, consider whether to add a mask rule or slight penalty for back-to-back prune→germinate cycles. At minimum, test the extreme: prune immediately and see if the agent can add a new seed right away. If it can, does the reward discourage this (see Phase 7)? It might "work" but lead to thrashing if not explicitly discouraged.

- **Lifecycle Stage Invariants:** Check that all gating conditions on lifecycle operations hold true in both the masking logic and execution. For example, **a seed should only be prunable after it's sufficiently mature and in HOLDING stage.** The code enforces this in two places: (1) in `_parse_sampled_action`, it sets `action_valid_for_reward=False` for PRUNE unless the seed is in HOLD with `alpha_mode==HOLD` and has spent ≥ `MIN_PRUNE_AGE` epochs [27]; (2) in the actual execution, it only performs the prune if the seed meets those criteria (and even then uses schedule vs. immediate) [26]. These dual checks (mask vs execution) were added because of a past bug (see "BUG-020 fix" comments) – essentially ensuring the mask's invariant is actually mirrored at runtime. To validate, create edge scenarios: e.g., try to prune a seed that's only 0 or 1 epochs old. The agent's raw output might suggest a prune (especially since we lowered `MIN_PRUNE_AGE` from 10 to 1 [28] to allow learning flexibility), but the system should convert that to a NO-OP (wait) or refuse to execute. Confirm that in such a case either the mask already blocked the choice or the post-decide logic flips it to WAIT (the parse function does `action_for_reward = WAIT` if invalid [29]). This is important because if a prune slips through too early, you'd break a PPO assumption that the action taken was allowed by the policy distribution (leading to an advantage miscalc since that transition wouldn't match any valid action probability). It might "appear" to work (the code would just not remove the seed or remove it and cause a weird reward), but it corrupts the training data subtly. Similarly, validate gating for GERMINATE (slot must be empty) [30], FOSSILIZE (seed must be in HOLDING) [30], and ADVANCE (seed exists in a stage < HOLDING) – these should be enforced consistently. Essentially, for each lifecycle op, one can write a small test: set up the preconditions incorrectly and ensure the outcome is a safe WAIT. If any illegal action is neither masked out nor turned into WAIT, that's a bug that could break Tamiyo's logic or PPO's trust region (by introducing unreachable actions).

- **Stabilization & Plateau Signals:** The SignalTracker's behavior should be checked after introducing multi-seed dynamics. For instance, `signals.is_stabilized` is a latch that never resets once true [31]. In a long multi-seed run, once the host stabilizes the first time, that flag stays true (to prevent repeated seed blasts). The developer should confirm that this design still makes sense – e.g., if a major structural change (like many seeds fossilized) occurs, do we ever want to reset stabilization? The spec says no (by design, latch is sticky) [32], but be mindful: if

stabilization triggers very early (maybe incorrectly), Tamiyo might stop germinating entirely thereafter. It's worth validating that the initial stabilization threshold (3% over 3 epochs by default) isn't tripped by normal noise early on. This is more of a configuration concern: run a baseline with no Tamiyo actions and see when `signals.metrics.stable_epochs` starts counting. If by epoch 3 or 4 it latches stable due to a minor plateau, the policy might never germinate seeds (thinking the host "converged"). This would **appear to be fine** (no errors, just no actions) but defeats the experiment. The fix might be adjusting `STABILIZATION_THRESHOLD` or requiring a minimum epoch before stabilization can latch. Not explicitly in code here, but the developer should manually monitor `plateau_epochs` and `is_stabilized` early in training as a guardrail.

- **State Field Freshness:** Monitor fields like `gradient_health_prev` or similar epoch-to-epoch metrics if they exist in the SignalTracker or SeedState. In the code, after each epoch they call `seed_state.sync_telemetry(...)` with current gradient stats [33]. Notably, they pass a constant `gradient_health=1.0` (since detailed health isn't computed) [34] and flags for vanishing/exploding gradients. If the system intends to use a trend of gradient health (comparing `gradient_health_prev` to current), this won't be meaningful – it's always 1.0 unless an extreme triggers a flag. The developer should ensure no logic explicitly depends on a decrease in gradient health from epoch to epoch. If there is (perhaps in some "seed is stuck" detection), it would silently never trigger. As a validation, search the code for any use of `gradient_health_prev` or similar; if found, ensure it's either removed or updated with a real computation in future. Similarly, if `epochs_since_counterfactual` is tracked (e.g., how long since a seed's last solo evaluation), verify that it increments/reset correctly. A seed in HOLDING might carry a counterfactual contribution from when it entered holding; if it stays holding for many epochs, there's a risk that metric becomes stale. The current code doesn't show an explicit field, but if such exists in SeedState, it should reset to 0 whenever a new counterfactual eval is done (like at fossilization or perhaps periodically). If it's never updated, a policy check like "prune if >N epochs since last counterfactual and still no improvement" would never fire. Essentially, any *time-since* or *previous value* fields should be validated by printing them across a few epochs to ensure they change as expected. Stale fields won't crash the training loop – they just yield inert or misleading data. This phase is about catching those silent issues.

Phase 5: Rollout Buffer & Bootstrapping Changes

- **Buffer Schema Alignment:** The rollout buffer (likely TamiyoRolloutBuffer) now stores many new fields (multiple actions, log-probs, masks, etc.). It's crucial to verify that the buffer's internal storage and sampling logic handle all of them consistently. After collecting a rollout, retrieve a sample (or the whole buffer) and check that each field (op, slot, blueprint, style, tempo, alpha_target/speed/curve, value, reward, done, masks, hidden states, etc.) has the correct shape and corresponds to the same timestep. For example, if `buffer.slot_action` for index i is 2, then `buffer.blueprint_action` at i might be 7, etc., and those together form the joint action taken at that step. A common bug would be misaligned arrays – e.g., if one of the new fields wasn't appended, all subsequent data could be off by one index. The add call lists all fields in order [35] [36]; if the buffer class's constructor or `add()` implementation didn't perfectly line up with this, you'd get a subtle corruption. The developer should run a mini-rollout (say 2 envs, 2 epochs) and print buffer contents to ensure each transition's data is coherent. Even something as small as a dtype mismatch (e.g., masks stored as int8 but read as bool) can cause silent errors during PPO updates (like incorrect masking). So, verify: actions and log_probs are stored as floats/ints as expected, masks are booleans or 0/1 floats, hidden states are stored and sliced

correctly (check that `hidden_h` and `hidden_c` from LSTM are properly carried – the buffer should not mix up which hidden state goes with which transition).

- **Sequence End Handling:** Confirm that episode termination and bootstrapping logic work hand in hand. The code marks each transition with `done` and a separate `truncated` flag [37]. Here, "truncated" means the episode continues (i.e., not truly done at environment level), so we need a bootstrap value for the next state; conversely, if `done=True` (episode ended), we use 0 as the bootstrap (no next value) [37]. Test this by ensuring that at the final epoch of an episode, the stored transition has `done=True` and `truncated=False`, and that its `bootstrap_value` is 0.0 in the buffer. For earlier transitions, you should see `done=False, truncated=True` and a nonzero `bootstrap_value` corresponding to the value function's estimate of the next state [37]. A likely pitfall: if an environment ends exactly at `max_epochs`, some code might mistakenly mark it as truncated instead of done. The logic in Phase 1 loop likely handles this, but it's worth cross-checking: after epoch 25 (if max_epochs=25), does `signals.done` come through correctly? A misflag here wouldn't crash anything but would cause the bootstrap logic to use a value when it shouldn't or vice versa, skewing advantage calculation. In practice, the values would then be slightly off and could cause the critic to learn incorrectly. So, it's a silent bug to guard against. If a discrepancy is found, fix how `done` is set in the environment or post-processing.

- **Bootstrap Indexing:** When multiple envs run in parallel, verify that the ordering of bootstrap values matches the transitions. The code collects `all_post_action_signals` for each env that isn't done [38], computes their values in one batch, then iterates through `transitions_data` and assigns bootstrap values to each truncated transition in order [37]. This assumes the sequence of `transitions_data` entries corresponds to the order of bootstrap values. It looks like `transitions_data` is built in env index order (in the Phase 1 loop), and `all_post_action_signals` is appended in env order as well [38], so the assumption holds. A quick validation is to instrument the code: log the env_id for each computed bootstrap value and the env_id for each transition as you assign it. They should match one-for-one. If they don't, then the wrong value could be assigned to a transition – a severe bug that would break advantage calculations in subtle ways. This is unlikely given the code structure, but since parallel loops can sometimes diverge, it's a good sanity check especially if any refactoring was done. Essentially you're ensuring `bootstrap_values[k]` truly corresponds to the k-th truncated transition in the list.

- **LSTM Hidden State Bookkeeping:** The buffer also stores LSTM hidden states for each transition (`hidden_h`, `hidden_c`) [39]. Ensure that these are used correctly when training the policy. For example, some PPO implementations with RNNs will reset the LSTM state at episode boundaries or carry it over for truncated rollouts. Our case likely treats each episode from epoch 1 to max_epochs as one sequence per env (since episodes aren't extremely long). The developer should confirm that at update time, if sequences are truncated (none likely are, unless using smaller rollout chunks than full episode), the training code either resets hidden state or splits sequences appropriately. A concrete test: if you reduce `max_epochs` for a quick run (say 5) and use `num_envs > 1`, does the PPO update function process the sequences without error? Pay attention to how `done` is used: typically, one multiplies the next-state value by (1 – done) when computing returns. We already handle that via bootstrap. Also, if doing multi-step advantage estimation, the code should treat done transitions as sequence breakpoints (which GAE inherently does by zeroing out beyond done). This is mostly consistent, but an oversight would be forgetting to reset the LSTM at episode end in the next rollout. Because here we likely end training after one episode per env, it's moot. However, if we ever loop back (multiple episodes),

make sure to call `agent.policy.reset()` or provide an initial hidden state for new episodes. Otherwise, a new episode could inadvertently start with a leftover LSTM state from last episode, which **won't throw an error** but could confuse the policy. Since the plan is single-run training, this might not occur, but it's a quality check if Phase 5 included "rollout buffer schema changes," perhaps anticipating multi-episode usage.

- **Reward and Value Normalizers with Buffer:** If using a reward normalizer (the code does maintain one for intrinsic stability [40] [41]), ensure it's applied consistently in storing and updating. For instance, they normalize the immediate reward before adding to `env_state.episode_rewards` [40]. Check that the normalized reward is what goes into the buffer (it should, as presumably `transition["reward"]` is that normalized value). If unnormalized rewards accidentally went into the buffer but the PPO update expects normalized (or vice versa), advantage calculations will be off. It's easy to miss because all values would just be scaled, not causing a code crash. So print a sample transition's reward from the buffer and confirm it matches the normalized value printed in logs. Similarly, if using a value normalization or advantages normalization (they do normalize advantages later), ensure those are just for training stability and not double-applied. No immediate bug stands out, but mixing normalized vs raw in buffer could quietly reduce training signal if done wrong.

Phase 6: PPO Update Loop & Clipping Behavior

- **Policy Ratio Computation:** Examine how the PPO ratio is computed with the multi-head actions. The code defines PPO ratio anomaly thresholds (e.g. ratio > 5.0 as explosion) [42], implying they calculate a **single ratio** for each timestep. It's likely they combine the heads' log-probabilities into one joint probability (since the overall action is the tuple of all head choices). We need to confirm that this joint ratio is calculated correctly. A potential oversight is if the new code uses only the op-head log-prob for the ratio or mixes up units. The safe way is: `ratio = exp(sum(new_log_probs_all_heads) - sum(old_log_probs_all_heads))`. Because old log-probs per head are stored in the buffer [36], and we can get new log-probs by running the current policy on the states (with the same masks). Validate that the implementation does exactly this: fetch each head's log-prob from the buffer and current policy, sum them to get the joint log-prob difference. If instead it only took the op log-prob (or averaged them or something), the trust region enforcement could break – e.g., the policy might radically change a rarely-used head (causing a big joint KL change) undetected if we were only watching the op probability. Given the anomaly detection code and that they pass all masks and heads through, they probably do it correctly, but it's crucial. One way to test: intentionally cause a big change in a secondary head (say blueprint) between iterations (maybe by tweaking a weight) and see if the reported ratio reflects it. If the code only considered op, it wouldn't catch it; if it considers all heads, the ratio would spike. This check ensures **PPO's clipping invariant (no large policy update step)** holds for the *joint* policy. A bug here could "appear" fine for primary decisions but allow uncontrolled drift in the other heads (violating the trust region). The developer should also check that the probability of masked-out actions is treated properly (likely masked logits yield fixed log-prob for the noop action). If the ratio computation inadvertently included impossible actions, it could produce NaNs or zeros – but since MaskedCategorical normalizes over valid actions, that's handled.

- **Per-Head vs Joint Advantage Usage:** The algorithm splits advantages per head, but uses a single optimizer to update the network. It's important to verify that the loss function correctly aggregates the per-head surrogate losses. For each head, they compute a surrogate loss term `L_head = w * min(ratio, clipped_ratio) * adv_head` and sum these up (plus value

and entropy losses). The code snippet for computing `surr1 = ratio * adv` and so on [43] suggests they do loop over heads and sum the contributions. Validate that the clipping is applied properly: the ratio should be clipped *per timestep* based on epsilon (0.2 by default) to form `surr2`, and then each head's adv is multiplied. One thing to watch: if `adv_head` is zero (masking), then that head contributes nothing regardless of ratio – which is fine. However, consider a scenario: one head changed drastically (ratio 10) but had zero advantage (unused), and another head had small ratio 1.0 with nonzero advantage. The joint ratio would be ~10, which would be clipped to 1.2 perhaps, and then the second head's loss is computed with that 1.2 * adv. This effectively means an irrelevant head's change caused the other head's update to clip when it otherwise wouldn't have. Is that happening? Or do they perhaps clip per head using that head's own ratio? If they compute a single joint ratio, then yes, a big change in a no-advantage head could prematurely trigger clipping globally. This is subtle and might *not* be what they intended. The safer approach would be to compute ratios per head independently. The code is a bit unclear; if `ratio` is calculated once for all heads, we have the coupling described. The developer should inspect this: run a controlled update where a secondary head's parameters are perturbed (simulate an extreme change) while the primary head's advantage is known, and see if the primary head's gradient is being clipped when it shouldn't. If so, consider adjusting the ratio calc to ignore heads with no advantage (since they had no effect on returns). The current design might tolerate it because the entropy term and lack of gradient on that head means such changes are rare, but it's an architectural quirk. **In summary, check whether PPO clipping is triggered appropriately and not too aggressively due to unrelated head changes.** The anomaly logs (ratio explosion/collapse) [42] should also be monitored – if you see frequent "ratio explosion" warnings that don't correspond to actual reward jumps, it could be this effect. This is something that won't break the code, but it may silently slow down learning by over-clipping.

- **Advantage Normalization and Stats:** The code normalizes advantages after GAE [44]. Ensure this is still beneficial with per-head advantages. Since a lot of advantage values will be zero (for heads that were masked on many steps), the mean/std computation should exclude those zeros (the code uses `valid_mask` for advantages [45], which likely filters out non-used transitions or perhaps non-done?). Double-check that "valid_advantages" in the stats excludes entries with zero advantage due to masking, otherwise the std will be underestimated (lots of zeros lower variance) [45]. It looks like they only compute stats on actually used advantages [45], which is good. The developer can verify by printing the advantage mean/std and the fraction of positive advantages metrics they log [46] – ensure these make sense (e.g. ~50% positive, mean ~0 by construction after normalization). If anything looks off (like advantage_mean is not ~0 or positive_ratio is extreme), it could indicate an error in how advantages are filtered or normalized (e.g., including dummy zeros). This would not stop training but would be a quality regression (advantages might not be properly normalized, affecting learning rate). Also, check that the number of advantages in the buffer matches expected transitions count per epoch * envs; if not, some transitions might have been dropped inadvertently.

- **Anomaly Detection Sensitivity:** The plan includes detecting entropy collapse (per-head entropy thresholds) and ratio anomalies [47]. Treat these not just as telemetry – test that they aren't being triggered spuriously. For instance, if the blueprint head is unused and outputs nearly random logits, its entropy might be naturally high or fluctuate, possibly triggering a false "entropy collapse" if it randomly becomes confident about one unused action. The code likely monitors entropy over time; the developer should ensure it's looking at meaningful data (e.g. maybe they only consider entropy on decisions where that head was active). If not, an alert could fire saying "entropy collapsed" when in reality it's just that the head was constant due to always being masked. Similarly, for ratio: a head with no impact could change a lot (big ratio) and set off the alarm even though the actual policy didn't effectively change for the task. Review the logging or

debug output for any such warnings during test runs. If found, this is more of a tuning issue – the fix might be to refine the detection logic (like ignore masked-out heads or require consecutive occurrences). It doesn't break training, but it could mislead debugging efforts. So the validation here is: run a few iterations with high debug verbosity and confirm that any "policy anomaly" diagnostics correspond to real issues (e.g. a bug causing an actual ratio explosion or a genuine entropy collapse in a used head). If they don't, adjust the criteria or at least document that those alerts can be safely ignored if conditions X, Y are true.

Phase 7: Reward Design & Hindsight Credit

- **Reward Component Verification:** The reward function for Tamiyo is complex, combining multiple terms (improvement, "rent" for added params, "shock" for rapid changes, scaffold hindsight credit, etc.). The developer should validate each component's calculation to ensure it matches the design intent:

- **Seed Contribution vs Total Improvement:** When a seed is pruned or fossilized, ensure the reward reflects the true contribution. E.g., for PRUNE actions, a seed's *negative* impact should yield a positive reward for removing it. The code uses a baseline accuracy with that seed disabled (`baseline_accs`) to measure contribution [48] [49]. Verify that baseline is correctly computed (the fused validation pass sets `baseline_accs[env_idx][slot]` for each seed's removal). A bug here might be if baseline accuracy isn't recorded for a seed (e.g. if a seed never got a counterfactual eval because it wasn't in BLENDING yet). The code guards `if target_slot in baseline_accs` [50], meaning if not, `seed_contribution` stays None and likely no reward for that. This could *appear fine* (no crash, just zero contribution), but it means the first time a seed is pruned (before any blending/hold stage), the reward might not capture its effect. The developer might consider that acceptable (we enforce MIN_PRUNE_AGE=1 so at least one epoch of data exists). It's still worth testing: prune a seed immediately after it finishes training stage (no counterfactual yet) and see if the reward makes sense (likely slightly negative because val_acc drops due to losing a seed with no baseline to compare – the code's `compute_reward` might handle None by treating contribution as 0). If it looks wrong, that might be an oversight to refine in reward calc.

- **Rent and Parameter Penalty:** Check that adding a seed incurs the intended "rent" cost. The code calls `compute_rent_and_shock_inputs` to get `effective_seed_params` and an `alpha_delta_sq_sum` for shock [51]. These feed into `compute_reward` via `reward_args` [52]. One should verify that for a GERMINATE action, `effective_seed_params` is positive (the new params) and thus the reward includes a negative term (penalty) to discourage adding too many parameters. If the agent germinates a huge seed, the reward should reflect a bigger penalty. You can test two scenarios: germinate a small blueprint vs a large blueprint and confirm the reward difference aligns with the blueprint's param count. If this isn't the case, there may be a bug or mis-scaling in how rent is calculated or normalized. The `reward_summary` accumulation in the code will tell you how much of the total reward came from `compute_rent` [53] – check those values for sanity (e.g., –0.1 vs –0.5 for small vs large seed, whatever expected). Any sign that the rent term is not changing or is positive would be a red flag.

- **Shock for Rapid Changes:** The `alpha_delta_sq_sum` represents the magnitude of sudden blending changes (e.g., pruning or retargeting causes abrupt shifts). Ensure that when a prune is scheduled slowly vs immediately, the shock term differs. For an immediate prune, we expect a larger shock penalty (because alpha goes from e.g. 1 to 0 instantly). If the agent uses a gradual schedule, shock should be smaller per step. This encourages smoother changes. Validate by comparing reward_components.alpha_shock for a fast prune vs slow prune (the code

accumulates `alpha_shock` in the summary [53]). A subtle bug would be if shock is computed but not actually included in the final reward due to a weighting issue. By printing `reward` and the individual components, you can see if `reward = bounded_attribution + compute_rent + alpha_shock + ...`. If something is consistently zero when it shouldn't be, that's a sign of a logic bug (maybe the shock wasn't added if no "last action" or similar).

- **Bounded Attribution & Total Improvement:** The term "bounded_attribution" likely refers to the seed's contribution limited by some range (perhaps clamped to avoid huge outliers). Check that when a seed is fossilized, the reward uses the **total improvement** that seed provided (if any). A known risk (the "ransomware seed" scenario) is that a seed could have high *counterfactual* value (removing it drops accuracy) but zero net improvement to the final model (maybe it just siphoned credit). If the reward only uses counterfactual contribution, the agent might fossilize seeds that actually weren't beneficial overall. Ideally, the reward should incorporate *total* improvement as well. The code hints at this: they check `seed_info.total_improvement` when marking a fossilized seed as contributing [54], but do they penalize if total_improvement is negative? Not explicitly in the snippet. As a check, if a seed got fossilized and had negative total_improvement, what happens? Possibly the baseline (host without seed) would be higher accuracy, meaning fossilizing it should be good (and the agent likely got negative contribution when it was active, so maybe it pruned earlier). This scenario might not occur often, but it's worth ensuring that the reward for FOSSILIZE isn't positive unless the seed's presence was indeed helpful. In testing, if you encounter a case where agent fossilizes a seed that made things worse (perhaps due to noise) and still gets a reward, that suggests the reward function might be overly trusting counterfactual. While fixing this might be complex (it's a design issue), being aware allows the developer to monitor if the agent starts fossilizing everything (thinking it's always good because counterfactual >0 by definition when it's allowed).

- **Scaffold Hindsight Credit:** The guide added a hindsight credit for scaffolding: when a seed is fossilized, seeds that "boosted" it (scaffolds) get some reward credit. The code calculates this by iterating `scaffold_boost_ledger` for the fossilized seed's slot [55], summing up a discounted credit up to `MAX_HINDSIGHT_CREDIT` (0.2) [56] [57]. This is a complex feature – verify it with a controlled scenario. For example, if scaffold seed A gave a boost of X to seed B's performance, when B is fossilized, seed A should get some credit (thus encouraging the agent to use scaffolds). In practice, ensure `env_state.pending_hindsight_credit` accumulates the expected values [58] and that the reward for the *scaffold seed's* actions includes this credit eventually. It might be applied as an addition to that seed's reward at the moment of fossilization (perhaps via the `pending_hindsight_credit` being added to the next reward calculation). The potential bug or oversight here: if the credit doesn't end up being delivered (e.g., if pending_hindsight_credit isn't used anywhere later), then scaffolding actions might be undervalued. The developer should search for where `pending_hindsight_credit` is read and added into the reward (likely in `compute_reward` for subsequent actions or final episode reward). If it's missing, that's a silent omission – everything runs, but the agent never gets the intended scaffold bonus. A quick test: if you know a scaffold event happened (check the summary's scaffold_count and scaffold_delay metrics [59]), see if any transition around that time had a reward bump corresponding to hindsight credit. If not, that's an oversight to address (perhaps by adding the pending credit to the scaffold seed's next ADVANCE or something).

- **No Thrashing Incentive:** Confirm that the reward structure discourages pathological behavior like rapidly germinating and pruning seeds (thrashing). Thrashing might *locally* bump accuracy (new seed gives a tiny boost, then is removed to reset plateau, repeat) but would stall long-term progress. The heuristic design prevented this via embargo and blueprint penalties, but for the RL policy we rely on rewards. Several components fight thrashing: (a) The rent penalty makes

germination costly if done too often (lots of params added); (b) a germination with no plateau improvement yields little positive reward (since no plateau trigger means probably `signals.metrics.plateau_epochs` wasn't high, so maybe the reward for germinating in non-plateau is low or negative); (c) a prune with minimal seed age yields a small positive reward (removing a seed that hadn't improved much doesn't give much, or might even be negative due to lost potential). The developer should simulate a mini-run where the agent tries a germinate immediately followed by prune of the same seed in the next epoch. Check the sum of rewards for those two actions. It should be negative or zero – meaning the agent gains nothing by doing that. If instead the reward ends up slightly positive (e.g., a germinate gave +0.1 for trying, and prune gave +0.05 for clearing, net +0.15 each cycle), the agent might exploit that loop. Such an outcome would "appear to work" in that training runs, but the agent would learn an unintended strategy. If detected, one might need to tweak reward scaling (maybe increase the rent cost or require a minimum plateau improvement for germination reward). The key validation is ensuring the **PPO policy doesn't find a degenerate reward hack**. Keep an eye on the learning curves: if you see oscillating seed count with no overall accuracy gain, suspect thrashing. For pre-release testing, deliberately provoke this: run with a high stabilization threshold (so Tamiyo always thinks plateau) and see if it starts adding and removing seeds frequently. The expectation is that the accumulated reward for pointless add-remove cycles is ≤ 0, thereby naturally disincentivizing thrash. If not, that's an oversight to fix in the reward design.

- **Telemetry and Counters:** As a final check on Phase 7, verify that all telemetry counters (like `seeds_fossilized`, `contributing_fossilized`, `seeds_created`) are incremented correctly and reset appropriately. These don't directly affect training but inconsistencies could hint at hidden problems. For example, `env_state.seeds_created` increments on each germination [60]; ensure it resets to 0 if a new episode begins (if episodes even repeat). If not (and episodes don't repeat), it's fine – it just monotonically increases. `seeds_fossilized` and `contributing_fossilized` increment on each successful fossilize [61]. They should not increment for prunes. Confirm that in a run with multiple seeds, the counts match the actual actions taken. If any of these counts is off, it means some action wasn't logged correctly (e.g., forgetting to update a counter on a certain code path). It won't break the algorithm, but it could confuse analysis later. Given the focus on reward, also verify that the **episode return reported** (if any) equals the sum of per-epoch rewards. The code appends normalized rewards to `env_state.episode_rewards` [40] – summing that list at episode end should give the total return for that episode. It's useful to log this and ensure it aligns with expectations (e.g., if the agent did one germinate that gave +0.5 reward and nothing else, the episode return should be +0.5). Any discrepancy could indicate an accounting bug (like forgetting to normalize one reward or double-counting something).

Phase 8: Final Testing & Edge-Case Validation

- **End-to-End Episode Test:** Before considering it "done," run a full episode (e.g., 25 epochs) in a variety of settings and visually inspect the key outputs (decisions, rewards, metrics) at each phase. This is a holistic validation covering all phases together. For instance, track one environment through a successful germinate→blend→fossilize sequence and confirm that at each step: the masks made the correct heads available, the buffer stored all needed data, the advantage for each head was reasonable, and the rewards reflected the outcome. Doing this end-to-end can catch integration issues that unit tests might miss – for example, a certain combination of events might reveal a state field not being reset. Look especially for anything that "looks odd but doesn't crash": e.g., a **seed remaining in a slot longer than expected** (did a pruned seed actually get removed after its schedule? If a seed was scheduled to prune over 3

epochs, verify it actually disappeared on time). If a seed's state persists beyond the intended epoch (say a bug where `schedule_prune` didn't finalize), the slot would never free up (Tamiyo would never germinate there again). This would be a **silent stall** – training would go on but one slot is effectively dead. Check logs or internal state to ensure scheduled prunes complete.

- **Memory and Resource Checks:** Although performance tuning is out of scope, correctness includes not leaking resources. The code made fixes like clearing optimizers on rollback [62] and after fossilization [63]. It's wise to monitor GPU/CPU memory during a long run. For example, ensure that after many epochs, the number of seed optimizers in `env_state.seed_optimizers` equals the number of active seeds, no more. They explicitly pop optimizers on germination (to re-init) and on removal [64] [65]. A subtle bug would be if an optimizer isn't cleared for some path (perhaps if a seed is pruned immediately, is it popped? Yes, they do in both immediate and scheduled cases [66] [67]). Similarly, ensure that no large tensor is growing each epoch – e.g., the `reward_summary_accum` or `scaffold_boost_ledger` might need resetting each epoch or episode. The code resets `scaffold_boost_ledger` entries when scaffolds are cleared [68], and `reward_summary_accum` per epoch. Just verify these don't accumulate indefinitely across epochs (they shouldn't, but a quick memory profile confirms it). While these issues wouldn't surface as exceptions, they can degrade performance or eventually OOM – catching them early is valuable.

- **Graceful Shutdown Behavior:** The training loop checks for a shutdown event to break out early [69]. Test this pathway to ensure it doesn't corrupt the buffer or leave partial updates. For example, if we stop training mid-epoch, do we properly handle the last partially filled batch and not compute advantages on incomplete data? The code breaks out of the epoch loop then a batch-level loop, which should trigger the PPO update on whatever was collected (or maybe skip if not enough). It's not core to learning, but a half-implemented shutdown could result in, say, an attempt to compute advantages with empty `bootstrap_values`. Simulate a shutdown after a few epochs and confirm no errors (and maybe that the model still saves if intended). It appears the design is such that it will stop after finishing the current batch gracefully [69] – just ensure that indeed `all_post_action_signals` etc. are not computed on a partially filled batch if break occurs. This is a minor edge case but easy to test.

- **File Sync and Integration Points:** Since this is a major refactor, double-check integration with any external components (not fully shown in code but implied by spec). For example, ensure Tamiyo's new policy outputs (the `TamiyoDecision` or `AdaptationCommand`) still work with Kasmina and Leyline. The heuristic Tamiyo used to produce decisions via `TamiyoDecision.to_command()`; in the RL case, we apply actions directly. But if any telemetry or external logger expects to receive an AdaptationCommand event, are we sending it? It looks like we execute actions directly in the environment model (calls like `model.germinate_seed()` etc. replace issuing a command). This is fine, but then `Nissa` telemetry for `TAMIYO_INITIATED` might not fire since that was tied to the heuristic's detect-stabilize event. The RL loop manually updates SignalTracker and could fire that event when stabilization happens (was that kept? Possibly in `signal_tracker.update()` if it flips `is_stabilized` true for first time, it might emit event). The developer should verify that any such side-effect (event emission) still occurs. This doesn't affect training correctness, but missing telemetry could be considered an oversight if analysts rely on it. If it's missing, add a one-liner to emit when `signals.is_stabilized and not already_emitted`.

- **Documentation and Defaults:** Lastly, verify that all new default constants (like `DEFAULT_MIN_FOSSILIZE_CONTRIBUTION`, `DEFAULT_GAMMA` for credit, etc.) are tuned as

expected. These are set in code (we saw references like `DEFAULT_MIN_FOSSILIZE_CONTRIBUTION` around fossilize logic [70]). If any differ from the design docs, update them or note it. Also ensure the configuration (CLI flags, etc.) correctly propagate – e.g., `--plateau-epochs N` sets plateau_epochs_to_germinate (should be wired in TamiyoPolicyConfig). These aren't code bugs per se, but if a default is wrong (say, MIN_FOSSILIZE_CONTRIBUTION was meant to be 1% but is 0.0 in code [71]), it could break an invariant (fossilize might trigger with no improvement at all). According to spec, it's 0.0 by default (which is quite lenient) [72], so be aware that the RL policy might fossilize seeds even if they contributed nothing (again, presumably the reward would not reward that if no improvement, so it's okay). Just double-check if any such threshold should be non-zero to avoid nonsense actions.

- **Easy-to-Miss Quality Wins:** Incorporate some final small improvements:

- Add assertions where assumptions are strong. For example, after Phase 0 validations, it might be worth asserting in code that `_BLUEPRINT_TO_INDEX` covers all blueprints (so a future dev adding a blueprint gets immediate feedback) – this reduces silent errors.
- Use the debug flags (like `ESPER_DEBUG_STAGE`) regularly during testing. Perhaps introduce similar debug checks for mask validity: after sampling an action, assert that if `action_for_reward` was set to WAIT due to invalid, the mask for that action was indeed zero. This would catch any inconsistency between mask and parse logic during development.
- Monitor training metrics like advantage skewness/kurtosis [73] and ratio diagnostics – these are already computed. The developer should actually use these to detect issues (the code computes and stores them [74]). For instance, if advantage skewness is hugely negative consistently, it could mean value function is lagging (lots of positive advantages, indicating underestimation). That might not be a "bug" but something to address (e.g., reduce value loss weight or learning rate). It's beyond pure correctness, but since the user is experienced, this insight can be valuable for phase-wise validation: *Is the training behaving as theoretically expected?* Any major deviation might hint at an implementation mistake earlier.

By systematically following these checklists for each phase, the developer can be confident moving forward without letting silent bugs or oversights accumulate. Many of the items above won't throw immediate errors if wrong – which is why this deep validation is necessary. Each phase's careful verification will ensure the Obs V3 + Policy V2 integration is **correct and robust**, preserving PPO invariants and Tamiyo's design principles throughout the implementation.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 18 20 21 22 23 24 25 26 27 28 29 30 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 73 74 TamiyoV3.txt
file://file_00000000d8e071f894c899f90eec7edd

16 17 19 advantages.py
https://github.com/tachyon-beep/esper-lite/blob/a28e21a19a6289ea34f8ddc1d5711e6ccb6e638b/src/esper/simic/agent/advantages.py

31 32 71 72 tamiyo.md
https://github.com/tachyon-beep/esper-lite/blob/a28e21a19a6289ea34f8ddc1d5711e6ccb6e638b/docs/specifications/tamiyo.md