ChatGPT

Keisei Shogi DRL – Evaluation System Refactor Proposal

1. Current Evaluation Pipeline Overview

The existing evaluation pipeline is triggered periodically during training via an `EvaluationCallback` [1]. At a set interval of training timesteps, the callback saves the current model to disk as a **checkpoint** for evaluation, selects a previous model checkpoint as the opponent, and then calls a separate evaluation routine. This routine is currently invoked by assigning the `execute_full_evaluation_run` function (from `keisei.evaluation.evaluate`) to the Trainer and calling it with the file paths of the agent and opponent checkpoints [2] [3]. In effect, the training loop "spawns" an evaluation subprocess logic within the same process by launching this external function with its own logging and WandB setup.

How it works today: When the callback triggers, it uses `trainer.agent.save_model()` to dump the latest model weights to a file (e.g. `"eval_checkpoint_ts{timestep}.pth"`) [2]. It then obtains a random opponent checkpoint from a `PreviousModelSelector` pool of past models [4]. If no past model is available (e.g. at the start of training), evaluation is skipped entirely [5]. Assuming an opponent is found, the Trainer sets its agent's network to eval mode and calls `execute_full_evaluation_run(...)` [6], passing in:

- The path to the just-saved agent checkpoint,
- The opponent type (e.g. `"ppo"` for a previous agent, or `"random"/"heuristic"` for built-in opponents),
- The opponent's checkpoint path (if applicable), and various evaluation parameters (number of games, max moves, etc.) [3].

This function instantiates an `Evaluator` object which loads the agent and opponent from the given checkpoint files, sets up a dedicated evaluation logger and (optionally) a new Weights & Biases run, then executes the core evaluation loop [7]. The **evaluation loop** runs a series of games between the agent and opponent sequentially in a single thread, using the `run_evaluation_loop` function in `keisei/evaluation/loop.py` [8]. Each game is played turn by turn in-memory (no external engine): the agent and opponent pick moves (agent via its neural network, opponent via a policy or heuristic) until the game ends. Results for each game are recorded, and after all games, win/loss/draw rates are computed [9] [10]. This results dictionary is returned to the callback, which logs a summary to console and W&B [11].

Subprocess & file-based communication: Notably, the evaluation is executed somewhat like a subprocess – the Trainer hands off evaluation to an external component. It relies on file I/O to communicate model data: the current model is saved to disk and immediately reloaded by the Evaluator [2] [3]. Similarly, the opponent model is loaded from its checkpoint file. This design, while avoiding a true OS subprocess, mimics one by isolating evaluation in its own context (with a separate WandB run, etc.) and using the filesystem as the intermediary. The WandB integration in eval is handled by re-initializing a new run for each evaluation (with `wandb_reinit=True`) [12] to avoid conflicts with the training process's WandB run.

Elo rating updates: After evaluation, if an Elo ratings registry is configured, the Evaluator updates a JSON-based Elo registry with the new results [13]. The agent's and opponent's Elo are adjusted using the list of game outcomes, and the updated ratings are saved to file via `EloRegistry.save()` [14]. This provides persistent tracking of relative skill, but it occurs at the end of each eval run in a ad-hoc manner.

Summary of current data flow: The training process yields control to the evaluation function periodically:

1. **Trainer/Callback** – saves model checkpoint, selects opponent, and calls eval function [2] [3].
2. **Evaluator** – loads models from disk, sets up logging (file + optional new W&B run), and runs games sequentially [15] [9].
3. **Results** – returned as a dict (win_rate, etc.), which the Trainer logs. Elo ratings file is updated on disk. The Trainer then resumes training.

This design successfully ensures the agent is evaluated, but it has multiple points of brittleness and inefficiency, discussed next.

2. Issues and Limitations of the Current Design

- **Sequential, Synchronous Execution:** All evaluation games run one after another on a single process/thread, causing lengthy evaluation times. For example, evaluating 100 games can take hours [16] during which training is paused. Modern multi-core systems are underutilized (only 1 core does simulation) [17] and the GPU (used for neural net inference) often sits idle while waiting for game logic, slowing feedback loops dramatically.

- **Training Interruption:** Because evaluation happens in the main thread, training is blocked until all games finish. This reduces overall training throughput and makes the system less amenable to real-time or continuous training scenarios. There is no parallelism between training and evaluation – they operate in a stop-start fashion.

- **Inefficient File-based Communication:** The pipeline uses disk I/O to pass data: writing the model to a `.pth` file and reading it back for eval [2] [3]. This is slow and brittle. If disk write or read fails (due to latency, permission, or disk full issues), the evaluation will fail. In-memory model data is not reused directly – even though the Trainer already has the model object, it must be serialized to disk and deserialized, adding overhead.

- **Suboptimal Opponent Selection:** The current logic picks **one** opponent checkpoint at random from a small pool of recent models [4]. This single-adversary evaluation provides a narrow view of performance. A lucky or unlucky draw of opponent can skew the win-rate result, and it doesn't rigorously assess if the new model is better than the overall pool or best model so far. There is no concept of always challenging a "champion" model or testing against multiple opponents; thus model progression tracking is noisy.

- **Brittle Coordination & Logging:** The evaluation routine runs as a quasi-separate workflow with its own logger and WandB setup. It uses `wandb.init` inside the same process to create a new run [18] [19]. This is fragile – misconfigurations or network hiccups can cause WandB errors that need catching [20], and having multiple WandB runs in one process can lead to confusion. Indeed, the code has to explicitly allow re-initialization and then always call `wandb.finish()` after eval [21]. Logging of results is indirect – the callback uses `trainer.log_both` to record a string of results and attaches the results dict for WandB [11], rather than logging structured

metrics to the training run. This indirection makes tracking eval metrics harder and could be unified.

- **Tight Coupling via Global Function:** The Trainer class doesn't have a dedicated evaluation component; instead it injects a function pointer for evaluation at runtime [22]. This is a code smell – it breaks encapsulation. The training loop calls `self.execute_full_evaluation_run(...)` which actually points to a standalone function in another module [22]. This design is brittle: if `execute_full_evaluation_run` is not set or if its signature changes, the callback would break. There's also hidden coupling in expecting the Trainer to have certain attributes (like `policy_output_mapper`, device, etc.) to pass into that function [23]. A more modular design is needed for clarity and maintainability.

- **Minimal Error Recovery:** The current system tends to catch exceptions only to log them and return `None` [24]. For example, if model loading fails or the game loop encounters an error, the Evaluator logs an error and the training simply continues without results. While training isn't halted (which is good for robustness), the system doesn't attempt to retry evaluation or fix the issue. Moreover, if an exception occurred at a bad time (e.g. after setting the model to eval mode but before resetting it), it could leave the Trainer's model in the wrong state. The callback does ensure `agent.model.train()` is called after eval in the normal flow [25], but in the `else` branch if the eval function was `None` or if an unexpected exception propagated, this might be missed. Overall, there's no centralized management of the evaluation process's health.

- **Testing and Extensibility Challenges:** Because evaluation logic is spread across the Trainer (setting function pointer), Callback (orchestrating files and parameters), and Evaluator (running games), it's hard to unit-test in isolation. The use of actual files and global states (like WandB and the filesystem) means tests must use real IO or heavy mocking. Extending the evaluation (for example, to run matches against multiple opponents or to use parallel threads) would require touching multiple pieces of code. The design is not easily extensible to, say, run a tournament between several models, or to integrate a new metric, without significant refactoring.

These issues point to the need for a more robust, integrated approach to model evaluation.

3. Proposed New Architecture

Overview of the Redesigned System

We propose refactoring the evaluation pipeline around two new core components: an `EvaluationManager` and an `OpponentPool`. These will be first-class citizens of the training process, managed by the `Trainer`, replacing the ad-hoc function/callback approach. The new architecture will enable in-process evaluation with controlled **multiprocessing** for parallel games, and support running background **round-robin tournaments** among multiple model checkpoints to continually assess and rank agents.

Key Objectives of the New Design:

- **Integrate Evaluation into Trainer:** The `Trainer` will own an `EvaluationManager` instance that handles all evaluation duties. This manager will encapsulate evaluation configuration, launching games (possibly in parallel worker processes), aggregating results, and reporting metrics. The training loop or callbacks will simply invoke methods on this manager (or notify it of events), rather than dealing with files and function calls directly.

- **Manage Opponents Systematically:** The `OpponentPool` will maintain a collection of past agent checkpoints (and/or their loaded models) to serve as evaluation opponents. It will replace the simple `PreviousModelSelector`. This pool will track each model's identity (e.g., by checkpoint timestamp or iteration), potentially its Elo rating, and provide selection strategies (e.g., the current champion or a diverse set of opponents) for evaluation matches. This creates a more structured approach to choosing opponents, enabling round-robin or league-style evaluations rather than one-off random picks.

- **In-Memory & Parallel Execution:** Instead of writing model weights to disk and spawning a new evaluator each time, the EvaluationManager can leverage the already-loaded model in memory. For example, it can clone the current agent's network weights to create a static copy for evaluation, and directly instantiate an opponent agent in memory from a saved state (or load from disk if not already loaded in the pool). Games will be executed in parallel where possible (using Python multiprocessing to utilize multiple CPU cores). By distributing games across workers, we can achieve the same number of games in a fraction of the wall-clock time, dramatically reducing the evaluation duration (targeting the 10-20x speedups outlined in the project's plan [26] [27]).

- **Asynchronous Background Tournaments:** The EvaluationManager will also support running ongoing tournaments in the background, pitting recent checkpoints against each other in round-robin fashion. This could run on a separate thread or process, continuously updating Elo scores without blocking the main training loop. For example, whenever a new checkpoint is added to the OpponentPool, the manager could schedule a series of games between that new model and some or all existing models in the pool (and possibly between older models as needed) to update the relative rankings. This provides a constantly-updated leaderboard of agents. By running these matches in the background (with lower priority), training can continue without waiting for full tournaments to complete.

- **Reliable Metric Logging:** In the new design, evaluation results will be logged through the existing training logging infrastructure (the `TrainingLogger` and `MetricsManager`) rather than spawning new WandB runs each time. The EvaluationManager can log aggregate stats (win rate, average game length, Elo changes, etc.) directly to the main W&B run as custom metrics (e.g., `eval_win_rate` at a given training step). This simplifies experiment tracking and avoids the overhead and complexity of multiple WandB runs. The Elo ratings will still be persisted (likely via the existing `elo_registry.json` file or a more robust database), but managed by the OpponentPool or EvaluationManager in one place.

- **Improved Fault Tolerance and Testing:** By encapsulating evaluation in its own manager, we can implement better error handling (e.g., if a worker process crashes or a game encounters an error, the manager can catch it, possibly retry that game or record a failure for later analysis, without crashing the whole training). This isolation also makes it easier to write unit tests for evaluation logic (we can instantiate an EvaluationManager with a dummy agent and opponent and simulate games or use a small deterministic environment). The code will be organized such that components like game simulation, result aggregation, and Elo calculation are methods that can be individually tested.

Below is a high-level data flow diagram of the proposed architecture:

```
sequenceDiagram
    participant Trainer
```

```
    participant EvaluationManager
    participant OpponentPool
    participant WorkerProc as Eval Worker(s)
    Note over Trainer,EvaluationManager: **Training loop triggers a periodic
evaluation**
    Trainer->>EvaluationManager: request_evaluation(current_agent)
    EvaluationManager->>OpponentPool: fetch opponent(s) for current_agent
    OpponentPool-->>EvaluationManager: opponent list (e.g. champion or pool
sample)
    Note over EvaluationManager,WorkerProc: **Parallel game execution for
each opponent**
    par for each opponent in list
        EvaluationManager->>WorkerProc: spawn game(s) agent vs opponent
        WorkerProc-->>EvaluationManager: game results (win/loss/draw)
    and
    end
    EvaluationManager: aggregate results (win rate, etc.)
    EvaluationManager->>OpponentPool: update Elo ratings (agent & opponents)
    EvaluationManager->>Trainer: return eval summary (or log metrics)
    Trainer->>Trainer: continue training (async if eval still running)
    Note over OpponentPool: OpponentPool retains new model if it performs
well (or always) and possibly evicts oldest if pool at capacity
```

(Figure: Sequence of interactions in the new in-process evaluation system. The Trainer triggers an evaluation, the EvaluationManager selects opponents from the OpponentPool, spawns parallel game simulations in worker processes, aggregates the results, updates ratings, and reports back to the Trainer.)

3.1 EvaluationManager Component

The `EvaluationManager` is the orchestrator of evaluations. It will be responsible for scheduling and executing evaluation games, either on-demand (e.g., when the training loop calls for a periodic eval) or continuously in the background. Key responsibilities and design features include:

- **Configuration:** Initialized with evaluation settings from the config (e.g., number of games per eval, max moves, evaluation frequency if needed, parallel worker count, etc.). It may also hold references to needed utilities like the game environment or policy mapper.

- **Triggering Evaluations:** Provide methods to start an evaluation of the current model. For example, `evaluate_current_model()` could be called by the training callback. This method will handle preparing the agent and opponent(s) for play and deciding whether to run synchronously or asynchronously. In synchronous mode, it would run the games and return the results. In asynchronous mode, it might spawn a background process or thread and immediately return, later providing results via a callback or a future/promise object. We may start with synchronous for simplicity and add async later.

- **Opponent Selection:** Use the OpponentPool to decide whom to play against. For instance, if the goal is to evaluate against the *current champion*, the manager will request the champion model from the pool. Or it could retrieve a list of multiple opponents (like top N Elo models, or a random sample) to get a broader evaluation. This strategy can be made configurable.

- **Launching Games in Parallel:** The EvaluationManager will implement parallel game execution using Python's `multiprocessing`. There are a couple of approaches:

- **Worker Pool:** Maintain a pool of worker processes (say, size N) that can each run games. The manager would distribute games across these workers. For example, if 20 games are to be played and 4 workers are available, each worker could simulate 5 games. We can leverage the Python `ProcessPoolExecutor` or a custom pool. The workers will need access to the agent's and opponent's parameters – either we send the model weights to each worker or have each worker load the model from disk. A straightforward implementation is to send each worker the file paths for agent and opponent; since the agent's checkpoint is readily available (we can still save a temp checkpoint or have the current weights in memory), the overhead of loading is offset by parallelism. A more advanced optimization is to use `fork` start method so that worker processes inherit a copy of the model from the main process (to avoid serialization), but with PyTorch and CUDA, using `spawn` is safer – so we likely will load models in each worker.
- **Single vs Multiple games per worker:** We could have each worker process play one game and return the result, or have each worker simulate a batch of games and return aggregated results. The latter can amortize the model load cost. For example, spawn 4 workers, each loads the two models once and plays 5 games, then returns stats. The EvaluationManager then combines these stats.

- The game simulation code can reuse the existing `run_evaluation_loop` logic, or we refactor it into a function that plays a single game given agent and opponent (which would be easier to distribute). For clarity, we might make a helper like `play_one_game(agent, opponent, max_moves)` that returns the outcome, and use that in each worker multiple times.

- **Result Aggregation:** After workers complete, the EvaluationManager collects their results. It will sum up wins, losses, draws, and compute aggregate metrics (win rate, etc.) just like the current `ResultsDict` [28] [9]. Because each worker might have partial results, the manager needs to combine them properly. We will ensure thread-safety (e.g., using synchronization if results come back asynchronously). In synchronous mode, we simply wait for all results and then aggregate.

- **Logging and Reporting:** Once the results are ready, the EvaluationManager will log the summary. It can use the Trainer's `log_both` method to print a concise message to console and W&B (e.g., "Eval at step 5000: win_rate=55%, avg_game_length=123 moves"). Additionally, it can log structured metrics via the Trainer's MetricsManager or directly via `wandb.log`. For instance, it could record the win rate under a key like `evaluation/win_rate` with the global timestep. Integrating with `MetricsManager` would allow the training loop's dashboard to display evaluation metrics over time (ensuring the UI can plot them).

- **Elo and OpponentPool Updates:** The EvaluationManager will update the OpponentPool with the results. If the eval involved the current model versus some opponents, we'll update those models' Elo ratings. This could be done by calling an Elo calculation utility (similar to EloRegistry's logic) – likely we will incorporate the EloRegistry usage here. For example, after getting the game outcomes list, call `OpponentPool.update_ratings(agent_id, opponent_id, results_list)`. The OpponentPool (or an EloManager within it) can compute new Elo values for both models. The EvaluationManager should also decide if the current agent should be added to the OpponentPool (perhaps it always is, or only if it meets a certain threshold). In a typical scenario, every saved checkpoint could be added to the pool (with the pool trimming the oldest if over capacity). Alternatively, we might only add checkpoints that were evaluated and performed well (to focus the pool on strong models). This policy can be refined; initially we might

add all to maintain continuity with the current behavior (which effectively adds every checkpoint to previous_model_selector).

- **Background Tournament Scheduling:** In addition to one-off evaluations, EvaluationManager can have a background thread/process that runs longer tournaments. For example, we could have a method `run_round_robin_tournament()` that takes a set of model IDs (e.g., all in the OpponentPool) and schedules games for each pairing. This is computationally expensive (O(N^2) matches for N models), so it might be done infrequently (say, after every K training iterations or once training is done). However, a lighter approach is **ongoing rating adjustments**: whenever a new model is added, run matches between that model and X randomly chosen existing models (or the top models) to place it on the Elo ladder, rather than full round-robin among all older models (which might not be necessary every time). Over time, as models are continually added and pairwise matches happen, Elo ratings converge. The EvaluationManager can manage this by enqueueing match jobs for the background workers. We'll ensure that these background matches do not interfere with the periodic main evaluation (perhaps use separate processes or schedule them when training is doing an update, etc.). If resources are limited (e.g., only a few CPU cores), we may interleave or throttle background games to not starve the self-play workers or training process.

To illustrate, here's a skeleton of what the `EvaluationManager` class might look like:

```
class EvaluationManager:
    """Orchestrates in-process evaluations of the agent during training,
    including parallel game simulation and opponent matchmaking."""
    def __init__(self, trainer: "Trainer", eval_config: "EvaluationConfig"):
        """
        Initialize the EvaluationManager.
        """
        self.trainer = trainer  # reference to access agent, config, logger
        self.num_games = eval_config.num_games
        self.max_moves = eval_config.max_moves_per_game
        self.device = trainer.device  # use same device or eval-specific
device
        self.parallel_workers = getattr(eval_config, "num_workers", 4)
        self.opponent_pool =
OpponentPool(capacity=eval_config.previous_model_pool_size,

default_elo=eval_config.default_elo_rating,
                                         k_factor=eval_config.elo_k_factor)
        # If desired, preload some opponents or initialize pool with a
random/heuristic opponent:
        if eval_config.opponent_type == "random":
            self.opponent_pool.add_builtin_opponent("random")
        # ... similarly for heuristic, etc.
        # Set up a process pool for parallel execution (could also be lazy-
initialized)
        self._worker_pool = None  # will initialize on first use to avoid
fork issues

    def evaluate_current_agent(self) -> Optional[dict]:
```

```
        """
        Evaluate the current trainer.agent against opponents from the pool.
        Returns a results dict with win/loss/draw rates and other stats, or
None if failed.
        """
        agent = self.trainer.agent
        if agent is None:
            return None  # Nothing to evaluate
        # Ensure agent is in eval mode and get a static copy of its policy
        agent.model.eval()
        # Select opponent(s) - e.g., the strongest opponent in the pool
        opponents = self.opponent_pool.select_opponents(strategy="champion")
# could return list
        if not opponents:
            self.trainer.log_both("[INFO] No opponents available for
evaluation.")
            agent.model.train()
            return None

        # Run games (possibly in parallel)
        results = self._run_games_parallel(agent, opponents)
        agent.model.train()  # restore training mode

        if results is None:
            self.trainer.log_both("[ERROR] Evaluation failed or was
canceled.")
            return None
        # Log and record metrics
        win_rate = results["win_rate"]
        self.trainer.log_both(
            f"Evaluation completed: win_rate={win_rate:.1%} vs
{len(opponents)} opponent(s)",
            also_to_wandb=True,
            wandb_data={ "eval/win_rate": win_rate, **results }
        )
        # Update Elo ratings for agent and opponents
        for opp in opponents:
            self.opponent_pool.update_ratings(agent_id=self.trainer.run_name,
                                              opponent_id=opp.id,

results_list=results["game_results"])
        # Optionally add the new agent to the pool (with provisional rating)
        self.opponent_pool.add_agent(self.trainer.agent,
agent_id=self.trainer.run_name)
        return results

    def _run_games_parallel(self, agent: "PPOAgent", opponents: list) ->
dict:
        """
        Run evaluation games in parallel processes. Distribute games among
opponents.
```

```
        """
        # Lazily initialize worker pool
        if self._worker_pool is None:
            import concurrent.futures
            self._worker_pool =
concurrent.futures.ProcessPoolExecutor(max_workers=self.parallel_workers)
        # Prepare tasks: for each opponent, schedule num_games (or num_games/
len(opponents) each)
        games_per_opp = max(1, self.num_games // len(opponents))
        futures = []
        for opp in opponents:
            # Package needed data (could be paths or lightweight state) for
each worker
            opponent_data = opp.get_evaluation_payload()  # e.g., checkpoint
path or serialized weights
            future = self._worker_pool.submit(run_games_worker,

self.trainer.agent.get_state_dict(),
                                              opponent_data,
                                              games_per_opp,
                                              self.max_moves,
                                              self.device)
            futures.append(future)
        # Collect results
        all_game_results = []
        total_moves = 0
        wins = losses = draws = 0
        for fut in futures:
            try:
                res = fut.result(timeout=300)  # wait for worker (with
timeout)
            except Exception as e:
                # If any worker fails, cancel remaining and return failure
                [f.cancel() for f in futures if not f.done()]
                self.trainer.log_both(f"[ERROR] Evaluation worker failed:
{e}", also_to_wandb=True)
                return None
            # Each res could be a dict similar to ResultsDict for that batch
            wins += res["agent_wins"]; losses += res["opponent_wins"]; draws
+= res["draws"]
            total_moves += res.get("total_moves", 0)
            all_game_results.extend(res["game_results"])
        games_played = wins + losses + draws
        # Compute aggregate stats
        win_rate = wins / games_played if games_played > 0 else 0.0
        loss_rate = losses / games_played if games_played > 0 else 0.0
        draw_rate = draws / games_played if games_played > 0 else 0.0
        avg_len = total_moves / games_played if games_played > 0 else 0.0
        return {
            "games_played": games_played,
            "agent_wins": wins,
```

```
            "opponent_wins": losses,
            "draws": draws,
            "game_results": all_game_results,
            "win_rate": win_rate,
            "loss_rate": loss_rate,
            "draw_rate": draw_rate,
            "avg_game_length": avg_len
        }

    # Additional methods for tournaments, etc., could be added here.
}
```

Code Skeleton: Proposed `EvaluationManager` *class.* (This code is illustrative; in practice, it would need to interface with real PPOAgent serialization, utilize actual game-playing worker functions, and possibly handle more opponent selection strategies.)

In the skeleton above, `run_games_worker` would be a module-level function (or staticmethod) that a worker process executes. It would accept the agent's weights or policy, an opponent specification, and play a certain number of games, returning a partial results dict. This function could internally load the models (if given file paths) and then loop over `ShogiGame` instances similarly to `run_evaluation_loop`. In a simplified form, it might call a helper for each game or directly incorporate a loop like in `run_evaluation_loop` [15] but for a limited number of games.

The `EvaluationManager.evaluate_current_agent()` method demonstrates the flow: set up the agent, choose opponents, run games, then log and update records. We ensure the agent model is set back to train mode after evaluation. The method also shows integrating with WandB via `trainer.log_both` – this way, we don't need a separate WandB init for eval; we log eval stats to the existing run (perhaps under distinct keys).

Notice that `EvaluationManager` uses `OpponentPool.select_opponents` to get opponents. Let's detail the OpponentPool next.

3.2 OpponentPool Component

The `OpponentPool` will maintain the set of opponent agents (checkpoints) available for evaluation. It essentially replaces and extends the `PreviousModelSelector` (which only provided random selection from recent checkpoints). The OpponentPool will provide better organization, such as tracking Elo ratings and offering different selection strategies. Key aspects:

- **Storage of Agents:** The pool can store entries that represent an agent snapshot. Each entry might include:
- An identifier (e.g. a name or ID; could be the checkpoint filename or a timestamp or an incremental model number).
- A reference to the model weights: possibly the file path, or a cached in-memory model object. We must be careful with memory – storing dozens of full neural nets in RAM can be heavy. A pragmatic approach is to keep file paths and lazily load models when needed for evaluation. However, for frequently used opponents (like the champion), we could cache the loaded model in memory for faster reuse. We might also store a truncated version (just needed network parameters).

- The current Elo rating of that agent, and perhaps some metadata like the number of games played in eval or last time it was updated.

- **Adding Agents:** When a new model checkpoint is saved (via CheckpointCallback), the Trainer will call something like `opponent_pool.add_agent(path, agent_id)`. This will add the checkpoint to the pool with a default Elo (e.g., 1500) if not specified. If the pool has a fixed capacity (say last 5 or 10 models), we may remove the oldest or lowest-performing model when capacity is exceeded. (The configuration `previous_model_pool_size` can dictate this [29].)

- **Built-in Opponents:** The pool can also hold non-learning opponents like a random-move player or heuristic bot. These can be added with an `add_builtin_opponent(name)` method. For instance, if `eval_cfg.opponent_type` is "random", we include a `RandomOpponent` (implementing the `BaseOpponent` interface) in the pool. This allows evaluating the agent against fixed strategies as well, not just past selves. We could tag such opponents with a special ID (e.g., "random" with infinite Elo or fixed rating).

- **Selecting Opponents:** The pool should provide methods to choose which opponents to face in a given evaluation:

- **Champion Strategy:** select the highest Elo rated model (the current champion) for a head-to-head. This is analogous to the AlphaGo/AlphaZero approach where a new model must beat the champion to take over. Even if we're not doing replacement gating, using the champion as the opponent is a strong test of skill.
- **Top-K or Mixed Strategy:** select a few opponents, e.g., the top N rated models and maybe a random recent one. This gives a broader evaluation and can mitigate variance.

- **Round-Robin (for tournament):** provide all or many models to evaluate in a league format. The strategy can be specified; the EvaluationManager might call `select_opponents(strategy="champion")` or `"sample"` etc. The OpponentPool, knowing each model's Elo or recency, can implement this easily.

- **Elo Rating Management:** The OpponentPool will integrate the EloRegistry functionality. It can either use composition (hold an `EloRegistry` object internally to load/save ratings) or directly manage a dict of ratings. Given EloRegistry is simple, we may merge it: e.g., OpponentPool can have `self.ratings: Dict[str, float]` which it updates and persists to a JSON file. For thread safety, any updates should be done with care (though since updates happen in the main process after collecting results, we can lock around it if needed).

The method `update_ratings(agent_id, opponent_id, results_list)` will calculate Elo adjustments for the two players given a list of results (`"agent_win"`, `"opponent_win"`, etc.). For example, using standard Elo formula: for each game, if agent won, agent's score=1, opponent's=0 (or 0.5 each for draw). Compute expected scores based on current ratings and update: new_rating = old_rating + K*(score - expected_score). We'll use the K-factor from config (already present in MetricsManager or EloRegistry config). After processing all games, update both ratings in the pool's record. If those agents weren't in the pool's rating dict (perhaps if a built-in opponent like random has no rating, we might assign a fixed or treat it separately). The OpponentPool will also handle saving the ratings to the JSON file (to retain state across runs) – likely after each update or periodically.

- **Access to Opponent Data:** When the EvaluationManager is about to launch a game, it needs to supply the opponent's model to the worker. OpponentPool can provide a method

`get_evaluation_payload(opponent_id)` that either returns a file path to the checkpoint or a ready-to-use object. For simplicity, we might use file paths and have workers load the model. Alternatively, if OpponentPool has the model loaded, it could serialize the state dict into a bytes object to send via multiprocessing (which avoids disk, but large IPC payload could be heavy; file might ironically be more efficient if already on disk).

For clarity, a simplified `OpponentPool` skeleton:

```
class OpponentPool:
    """Manages a pool of opponent models (agent snapshots and built-in
opponents) for evaluation."""
    def __init__(self, capacity: int = 5, default_elo: float = 1500.0,
k_factor: float = 32.0):
        self.capacity = capacity
        self.default_elo = default_elo
        self.k_factor = k_factor
        self.pool: Dict[str, OpponentEntry] = {}  # Mapping from model ID to
OpponentEntry
        # OpponentEntry could be a small dataclass with fields: id,
checkpoint_path, rating, loaded_model
        self.ratings: Dict[str, float] = {}
        # Load existing Elo ratings from file if exists
        try:
            self._load_ratings("elo_ratings.json")
        except Exception as e:
            print(f"[WARN] Failed to load Elo ratings: {e}")
        # Initialize built-in opponent container if needed:
        self.builtins: Dict[str, BaseOpponent] = {}

    def add_agent(self, agent: "PPOAgent", agent_id: str, checkpoint_path:
Optional[str] = None):
        """Add a new agent (checkpoint) to the pool with default rating."""
        if agent_id in self.pool:
            return  # already exists (could update? but typically not)
        if checkpoint_path is None:
            # Save agent's weights to a new checkpoint file if path not
provided
            checkpoint_path = os.path.join( ..., f"{agent_id}.pth")
            agent.save_model(checkpoint_path)
        entry = OpponentEntry(id=agent_id, path=checkpoint_path)
        self.pool[agent_id] = entry
        self.ratings.setdefault(agent_id, self.default_elo)
        # Enforce capacity: if too many, remove the oldest or lowest Elo
        if len(self.pool) > self.capacity:
            self._evict_one()

    def add_builtin_opponent(self, name: str):
        """Add a built-in static opponent (like random or heuristic)."""
        if name == "random":
            opp = RandomOpponent()  # assume this is a BaseOpponent
implementation
```

```
        elif name == "heuristic":
            opp = ShogiHeuristicAI()
        else:
            raise ValueError("Unknown built-in opponent")
        self.builtins[name] = opp
        self.ratings.setdefault(name, self.default_elo)
# could assign fixed rating or treat separately

    def select_opponents(self, strategy: str = "champion", num: int = 1):
        """Select opponents for evaluation based on strategy."""
        if strategy == "champion":
            # pick the highest Elo opponent
            if not self.ratings:
                return []
            champ_id = max(self.ratings, key=lambda mid: self.ratings[mid] if
mid in self.pool or mid in self.builtins else -float('inf'))
            return [ self._get_opponent_entry(champ_id) ]
        elif strategy == "top_k":
            sorted_ids = sorted(self.ratings, key=lambda mid:
self.ratings[mid], reverse=True)
            top_ids = [mid for mid in sorted_ids if mid in self.pool or mid
in self.builtins][:num]
            return [ self._get_opponent_entry(mid) for mid in top_ids ]
        elif strategy == "random":
            # return a random opponent from pool (for variability)
            import random
            all_ids = list(self.pool.keys()) + list(self.builtins.keys())
            if not all_ids:
                return []
            rand_id = random.choice(all_ids)
            return [ self._get_opponent_entry(rand_id) ]
        else:
            # default fallback: champion
            return self.select_opponents("champion")

    def update_ratings(self, model_a_id: str, model_b_id: str, results:
list[str]):
        """Update Elo ratings for two models based on a list of game
outcomes."""
        # Ensure both have ratings entries
        if model_a_id not in self.ratings:
            self.ratings[model_a_id] = self.default_elo
        if model_b_id not in self.ratings:
            self.ratings[model_b_id] = self.default_elo
        Ra = self.ratings[model_a_id]
        Rb = self.ratings[model_b_id]
        # Elo expected score for A vs B
        def expected_score(Ra, Rb):
            return 1 / (1 + 10 ** ((Rb - Ra) / 400))
        Sa_total = 0.0  # total score for A over all games
        n = len(results)
```

```
        for outcome in results:
            if outcome == "agent_win":
                Sa_total += 1.0  # agent A won
            elif outcome == "draw":
                Sa_total += 0.5  # draw gives half to A
            # if opponent wins, A gets 0
        if n > 0:
            Sa = Sa_total / n
            Ea = expected_score(Ra, Rb)
            Eb = expected_score(Rb, Ra)
            # Update ratings
            new_Ra = Ra + self.k_factor * (Sa - Ea)
            # Opponent B's score Sb = 1 - Sa (if no draws, or handle draw
accordingly)
            Sb = 1 - Sa if not any(o == "draw" for o in results) else (n -
Sa_total) / n
            new_Rb = Rb + self.k_factor * (Sb - Eb)
            self.ratings[model_a_id] = new_Ra
            self.ratings[model_b_id] = new_Rb
            self._save_ratings("elo_ratings.json")
        # (If needed, we could also update a list of top models, etc.)

    def get_rating(self, model_id: str) -> float:
        return self.ratings.get(model_id, self.default_elo)

    def _get_opponent_entry(self, model_id: str):
        """Helper to retrieve an OpponentEntry or built-in opponent for
evaluation."""
        if model_id in self.builtins:
            opp = self.builtins[model_id]
            return OpponentEntry(id=model_id, path=None, builtin=opp)
        return self.pool.get(model_id)

    def _evict_one(self):
        # Evict a model from pool when over capacity.
        # For example, remove the lowest Elo model or the oldest added.
        if not self.pool:
            return
        # Simple strategy: remove lowest Elo that is in pool (not built-in)
        lowest_id = min(self.pool.keys(), key=lambda mid:
self.ratings.get(mid, self.default_elo))
        self.pool.pop(lowest_id, None)
        # We might also remove its rating entry or keep it for historical
purposes.
        # (For simplicity, could leave rating but it won't be used if not in
pool)
        # Also consider deleting its checkpoint file to save space if not
needed.

    def _load_ratings(self, filepath: str):
        if os.path.exists(filepath):
```

```
            import json
            data = json.load(open(filepath))
            self.ratings.update(data.get("ratings", {}))

    def _save_ratings(self, filepath: str):
        import json
        try:
            json.dump({"ratings": self.ratings}, open(filepath, "w"))
        except Exception as e:
            print(f"Could not save Elo ratings to {filepath}: {e}")
```

Code Skeleton: Proposed `OpponentPool` *class.* (This pseudocode omits some details for brevity, such as definition of `OpponentEntry` dataclass and actual model loading logic. It focuses on how opponents are stored and selected, and how Elo ratings are updated.)

In this design, the `OpponentPool` concentrates the logic for opponent management. The `EvaluationManager` doesn't need to know how Elo ratings work or how to pick a good opponent – it asks the pool for opponents and tells it about results to update ratings. This separation improves modularity (we could swap out OpponentPool's strategy without touching the EvaluationManager).

Integration with Trainer: The Trainer will create an EvaluationManager at startup, e.g.:

```
self.evaluation_manager = EvaluationManager(self, config.evaluation)
```

instead of setting `execute_full_evaluation_run`. The `EvaluationCallback` will become much simpler – it no longer needs to handle file paths or opponent selection itself. It can be something like:

```
class EvaluationCallback(Callback):
    def __init__(self, interval: int):
        self.interval = interval

    def on_step_end(self, trainer: "Trainer"):
        if (trainer.global_timestep + 1) % self.interval != 0:
            return
        if not trainer.agent:
            trainer.log_both("[ERROR] Cannot evaluate - agent not
initialized.", also_to_wandb=True)
            return
        # Trigger evaluation via the manager
        trainer.log_both(f"Triggering evaluation at timestep
{trainer.global_timestep+1}...", also_to_wandb=True)
        trainer.evaluation_manager.evaluate_current_agent()
```

This callback simply delegates to the EvaluationManager. It doesn't need to manage model state (the manager handles switching to eval mode and back), doesn't need to pick or load opponents itself, and doesn't need to construct the long parameter list for `execute_full_evaluation_run` (which is now gone). All of that complexity is encapsulated.

3.3 Trainer and Workflow Integration Changes

With the above components in place, the overall training workflow will change as follows:

- **Trainer Initialization:**
- Remove the legacy `self.execute_full_evaluation_run` and instead instantiate `self.evaluation_manager` and possibly `self.opponent_pool` if not inside the manager.
- The Trainer can pass itself to the EvaluationManager so that the manager can access the agent and logging methods (as we did in the skeleton by storing `trainer` reference). This does create a circular reference (Trainer -> EvaluationManager -> Trainer), but it's manageable as long as we're careful with usage; alternatively, the EvaluationManager could accept specific needed components (like the agent or a logger function) instead of full trainer reference to decouple slightly.

- Ensure the Trainer passes config values like `previous_model_pool_size` to OpponentPool.

- **Checkpoint Callback:** After saving a checkpoint, it should inform the OpponentPool. For example, in `CheckpointCallback.on_step_end`, where we currently see `trainer.previous_model_selector.add_checkpoint(ckpt_path)` [30], we will replace it with:

```
if hasattr(trainer, "evaluation_manager"):
    trainer.evaluation_manager.opponent_pool.add_agent(trainer.agent,
agent_id=ckpt_name, checkpoint_path=ckpt_save_path)
```

  This way, every new checkpoint is recorded in the pool. (Alternatively, we might only add to pool when an evaluation is about to use it, but adding all is simpler and matches current behavior of keeping a rolling pool.)

- **Removal of Redundant Config Fields:** In the current code, the evaluation callback passes a lot of parameters to `execute_full_evaluation_run` [23] which are sourced from config (like `eval_cfg.num_games`, `eval_cfg.wandb_project_eval`, etc.). With the new design, many of these can be consolidated. The EvaluationManager is configured once with these values. We no longer need to pass them around repeatedly. For instance, `wandb_project_eval` and similar fields might become obsolete if we log to the same project as training. The config can be simplified to just parameters needed by EvaluationManager (num_games, maybe whether to do background tournaments, etc.).

- **MetricsManager Integration:** The `MetricsManager` in training possibly tracks Elo and other metrics for display. We should ensure it stays updated or possibly use it to log evaluation metrics. For example, MetricsManager might have a method to record the latest Elo of current model. The Trainer could retrieve `trainer.evaluation_manager.opponent_pool.get_rating(current_id)` and feed it to metrics for plotting. This is an optional integration for richer UI. At minimum, using WandB logging as shown suffices.

- **Thread/Process Safety:** Running parallel processes for evaluation while the main training is also possibly using multiple processes for self-play (if parallel training is enabled) introduces complexity. We need to ensure that the evaluation workers do not interfere with training

workers. Ideally, they should be separate pools. The current training parallelism (if enabled) uses its own `ParallelManager` [31] for self-play. Our EvaluationManager could either use that same infrastructure (e.g., reuse self-play workers for evaluation tasks during idle times) or maintain a separate pool. For clarity, we designed a separate pool. This means if both are active, CPU usage will increase. A future optimization could coordinate them (for example, pause collecting new experiences while evaluation runs, or run evaluations on separate machines). For now, we assume either training is not heavily parallel or we schedule eval at times that don't conflict too much.

- **Backward Compatibility Considerations:** We should ensure that if the new system fails or is turned off, the trainer can still function. For instance, we might keep a fallback option to use the old `execute_full_evaluation_run` for a transitional period (perhaps behind a config flag). But ultimately, the goal is to replace it entirely. We will remove the legacy evaluation script usage. The CLI entry point `keisei.evaluation.evaluate.main_cli` can remain for manually evaluating a model outside of training (that can still instantiate an Evaluator), but internally, training won't call that. Our redesign focuses on the training-time evaluation.

Before/After Code Diff Example

Below are illustrative code diffs showing how the training code changes with this refactor:

Trainer Initialization (before vs after):

```
 class Trainer(CompatibilityMixin):
     def __init__(self, config: AppConfig, args: Any):
         ...
-        self.execute_full_evaluation_run: Optional[Callable] = None
         ...
         # Initialize managers
         self.display_manager = DisplayManager(config, self.log_file_path)
         ...
         self.model_manager = ModelManager(config, args, self.device,
self.logger.log)
         ...
         self.previous_model_selector =
PreviousModelSelector(pool_size=config.evaluation.previous_model_pool_size)
         self.evaluation_elo_snapshot: Optional[Dict[str, Any]] = None
         self.callback_manager = CallbackManager(config, self.model_dir)
         ...
         # Setup display and callbacks
         self.display = self.display_manager.setup_display(self)
         self.callbacks = self.callback_manager.setup_default_callbacks()
         ...
-        # Initialize TrainingLoopManager
+        # Initialize TrainingLoopManager and Evaluation Manager
         self.training_loop_manager = TrainingLoopManager(trainer=self)
+        self.evaluation_manager = EvaluationManager(trainer=self,
eval_config=config.evaluation)
```

Diff 1: Trainer no longer assigns `execute_full_evaluation_run` *function, and instead creates an EvaluationManager.* (The `PreviousModelSelector` could be removed in favor of OpponentPool inside the EvaluationManager.)

Evaluation Callback (before vs after):

```
 class EvaluationCallback(Callback):
     def __init__(self, eval_cfg, interval: int):
-        self.eval_cfg = eval_cfg
         self.interval = interval
```

```
     def on_step_end(self, trainer: "Trainer"):
-        if not getattr(self.eval_cfg, "enable_periodic_evaluation", False):
-            return
-        if (trainer.global_timestep + 1) % self.interval == 0:
-            if not trainer.agent:
-                ...
-                return
-            ...
-            eval_ckpt_path = os.path.join(trainer.model_dir,
f"eval_checkpoint_ts{trainer.global_timestep+1}.pth")
-            opponent_ckpt = None
-            if hasattr(trainer, "previous_model_selector"):
-                opponent_ckpt =
trainer.previous_model_selector.get_random_checkpoint()
-            if opponent_ckpt is None:
-                trainer.log_both("[INFO] No previous checkpoint available
for Elo evaluation.", also_to_wandb=False)
-                return
-            trainer.agent.save_model(eval_ckpt_path, ...)
-            trainer.log_both(f"Starting periodic evaluation at timestep
{trainer.global_timestep + 1}...", also_to_wandb=True)
-            trainer.agent.model.eval()
-            if trainer.execute_full_evaluation_run is not None:
-                eval_results = trainer.execute_full_evaluation_run(
-                    agent_checkpoint_path=eval_ckpt_path,
-                    opponent_type="ppo",
-                    opponent_checkpoint_path=str(opponent_ckpt),
-                    num_games=getattr(self.eval_cfg, "num_games", 20),
-                    max_moves_per_game=...,
-                    device_str=trainer.config.env.device,
-                    log_file_path_eval=getattr(self.eval_cfg,
"log_file_path_eval", ""),
-                    policy_mapper=trainer.policy_output_mapper,
-                    seed=trainer.config.env.seed,
-                    wandb_log_eval=...,
-                    wandb_project_eval=...,
-                    wandb_entity_eval=...,
-                    wandb_run_name_eval=f"periodic_eval_{trainer.run_name}
```

```
_ts{trainer.global_timestep+1}",
-                     wandb_group=trainer.run_name,
-                     wandb_reinit=True,
-                     logger_also_stdout=False,
-                     elo_registry_path=getattr(self.eval_cfg,
"elo_registry_path", None),
-                     agent_id=os.path.basename(eval_ckpt_path),
-                     opponent_id=os.path.basename(str(opponent_ckpt)),
-                 )
-                 trainer.agent.model.train()
-                 trainer.log_both(f"Periodic evaluation finished. Results:
{eval_results}", also_to_wandb=True,
-                                  wandb_data=(dict(eval_results) if
isinstance(eval_results, dict) else {"eval_summary": str(eval_results)}))
-                 if getattr(self.eval_cfg, "elo_registry_path", None):
-                     ...
-                     trainer.evaluation_elo_snapshot = snapshot
-             else:
-                 trainer.agent.model.train()
+         if (trainer.global_timestep + 1) % self.interval != 0:
+             return
+         if not trainer.agent:
+             trainer.log_both("[ERROR] EvaluationCallback: No agent to
evaluate.", also_to_wandb=True)
+             return
+         trainer.log_both(f"Starting evaluation at step
{trainer.global_timestep+1}...", also_to_wandb=True)
+         eval_results = trainer.evaluation_manager.evaluate_current_agent()
+         # (The EvaluationManager handles logging and setting model train/
eval modes internally)
```

Diff 2: The new EvaluationCallback is dramatically simpler. It no longer needs to manage file paths or WandB details. The `EvaluationManager.evaluate_current_agent()` call encapsulates what used to be ~30 lines of logic. This improves maintainability and clarity.

After evaluation results handling: In the old code, after eval, they updated an `evaluation_elo_snapshot` in Trainer with some Elo info [32] [33]. In the new design, the Elo handling is within OpponentPool. If needed, the Trainer can query the OpponentPool for an updated ranking. For example, we could remove `trainer.evaluation_elo_snapshot` entirely, or update it like:

```
trainer.evaluation_elo_snapshot = {
    "current_id": current_model_id,
    "current_rating":
trainer.evaluation_manager.opponent_pool.get_rating(current_model_id),
    "opponent_id": opp_id,
    "opponent_rating":
trainer.evaluation_manager.opponent_pool.get_rating(opp_id),
    "last_outcome": ...,
    "top_ratings":
```

```
trainer.evaluation_manager.opponent_pool.get_top_ratings(n=3)
}
```

This way, any UI component that was showing `evaluation_elo_snapshot` remains supported.

Overall, the **after** state has a cleaner separation of concerns: the callback triggers eval, the manager does the heavy lifting, and the pool maintains the history and stats.

4. Implementation Plan (Phased Roll-out)

Refactoring a core piece like this needs to be done in stages to ensure reliability. We propose the following phases:

- **Phase 0: Preliminary Design and Validation** – Discuss and approve this design (via an ADR, see below). Write comprehensive tests for the current evaluation behavior (if not already existing) to capture baseline functionality (win-rate calculation, Elo updates, etc.), which will be used to verify the refactor does not break intended outcomes.

- **Phase 1: Introduce `EvaluationManager` and integrate with Trainer (without parallelism or OpponentPool initially).** In this phase, we implement the `EvaluationManager` as a wrapper around the existing sequential evaluation logic. It will still save the model to file and call `run_evaluation_loop` internally, but through a cleaner interface. The goal is to get rid of `execute_full_evaluation_run` and move that logic into `EvaluationManager.evaluate_current_agent()`. The OpponentPool can initially be a thin wrapper around PreviousModelSelector (or even just always use the last checkpoint) to keep behavior identical. We ensure that all tests pass with this new pathway, validating that metrics and logs appear as before. Essentially, Phase 1 is a refactor without changing functionality yet, but establishing the new class structure.

- **Phase 2: Integrate OpponentPool for opponent management.** Develop the OpponentPool class to manage a list of recent checkpoints and Elo ratings. At first, we can use it in a simple way: always select a random opponent (to mimic current behavior) or the latest previous checkpoint. Then gradually enhance selection strategy to champion or top-k. Also, remove `PreviousModelSelector` usage in Trainer/Callbacks, replacing with OpponentPool. Write unit tests for OpponentPool (adding and evicting agents, Elo update math – compare EloRegistry outcomes to ensure consistency [34] [35]). In this phase, we'll start updating Elo via OpponentPool.update_ratings and ensure the Elo JSON file is being updated and saved [36] [37]. Verify that after an eval, the Elo values match what EloRegistry would have produced earlier.

- **Phase 3: Replace disk file dependency with in-memory model access.** Modify `EvaluationManager.evaluate_current_agent()` to avoid saving the agent to disk for evaluation. We can directly use `trainer.agent`'s model. For the opponent, if OpponentPool has a file path, we may still load it (this is fine). But we can eliminate the redundant save of the current agent. This optimizes performance and reduces I/O. Test that evaluation still returns correct results. Also ensure that skipping the save doesn't break any checkpoint-related assumptions (it shouldn't, since we still call the CheckpointCallback separately for persistence). Essentially, we decouple evaluation from checkpoint saving – it evaluates the current model state directly.

- **Phase 4: Implement parallel game execution.** Introduce multiprocessing in the EvaluationManager. We can start with a simple approach: spin up a few processes to each play ~1/N of the games. Use Python's `multiprocessing` or `concurrent.futures`. Take care with PyTorch: if using CUDA, ensure each worker loads the model on the GPU or use CPU for evaluation to avoid GPU contention (possibly make eval use device=cpu if that's acceptable, or if multi-GPU, assign one for eval). Start with a fixed small number of workers and test with a small number of games in a dev environment to ensure it speeds up linearly. Write stress tests (simulate 50 games with dummy fast agent vs agent) to measure speedup and validate consistency of aggregated results vs single-thread version. Also handle worker failures gracefully (timeouts or exceptions).

- **Phase 5: Asynchronous/background evaluation (optional initial implementation).** This is more advanced – enabling the EvaluationManager to run in a separate thread such that training doesn't pause. We might skip this if Phase 4 parallelism already makes eval fast enough (e.g., completing in minutes). However, for full completeness, we can prototype running `evaluate_current_agent()` in a background thread. One approach: when callback triggers eval, instead of directly blocking, call `threading.Thread(target=self.evaluate_current_agent).start()`. The thread will execute and log independently. The Trainer can carry on training in parallel. We must be careful with shared state (the agent's model shouldn't be updated during evaluation games to maintain fairness – one solution is to snapshot weights at the start of eval; since we already clone or have them saved, that's fine). If using threads, GIL might limit parallelism especially if the game loop is Python-heavy – but our heavy tasks (neural net inference) release GIL via numpy/torch, and we'll also rely on multi-processing for multiple games, so thread is mostly overseeing that. We have to ensure that if training is updating the model at the same time, the eval uses the old snapshot – which we've handled by copying weights or using a saved checkpoint. This feature will need careful testing to ensure no race conditions or crashes (especially with WandB logging from two threads). Possibly introduce a lock around WandB log or use thread-safe queues for logging.

- **Phase 6: Background round-robin tournament feature.** Extend the EvaluationManager with a method to periodically run broader evaluations. For instance, every N timesteps or whenever a new model is added, spawn tasks for that model to play some games against top pool members (this could be done immediately in the evaluation callback or on a separate schedule). Another idea: run a tournament at the end of an training epoch or end of training. This phase can be adjusted based on project needs – it might be sufficient to implement the support and not use it by default initially. Implementation steps: get all pairs from OpponentPool (or a subset), distribute matches to workers similar to before, compute a mini table of results, update Elo for all pairs. This is essentially extending what we did for one-vs-one to many-vs-many. We should ensure this can run asynchronously to not stall training (since it could be many games). Perhaps spawn a separate process solely dedicated to tournaments which can run concurrently. Testing this thoroughly is important: we can simulate a pool of agents with known relative strength and see if Elo rankings converge correctly after a tournament.

- **Phase 7: Cleanup and Deprecation Removal.** Remove the now-unused `keisei.evaluation.evaluate` script or mark it deprecated (it might still be useful for a standalone eval CLI, but the training no longer uses it). Remove `PreviousModelSelector` if fully replaced. Update documentation (the docs in `docs/component_audit/` would need revisions to reflect new architecture). Also update configuration schema if some fields (like separate WandB project for eval) are deprecated – or repurpose them (e.g., `enable_periodic_evaluation` can remain to toggle the whole system, etc.). Update any user-facing HOW_TO_USE guides to describe how evaluation is now integrated.

Throughout these phases, we will maintain **test coverage**: - Unit tests for `EvaluationManager` with a fake game (we might create a dummy environment or stub `ShogiGame` that terminates immediately and returns a fixed winner, to test the parallel logic). - Unit tests for `OpponentPool` ensuring adding and removing works and Elo is updated as expected (including edge cases like draws). - Integration test where we run a few training steps with a Trainer configured to use the new system and verify that after an eval trigger, the metrics (win_rate, Elo) are sane and that the OpponentPool contains expected entries. - Performance tests to confirm improvement (this can be as simple as measuring time to evaluate 50 games before vs after parallelization). - Failure injection tests: e.g., make a worker process raise an exception to see if EvaluationManager handles it without crashing the Trainer.

By rolling out in phases, we can deploy Phase 1–3 which already simplify the design without changing external behavior much (ensuring we haven't broken anything), then enable Phase 4 parallelism (which should be internal and not affect results, only speed), and then optionally Phase 5–6 for advanced functionality.

Careful communication in release notes (if this is a published codebase) would accompany Phase 7, as it changes how evaluation is configured and reported.

5. Architecture Decision Record (ADR)

ADR Title: Refactor Evaluation Pipeline to Integrated EvaluationManager & OpponentPool

Context & Problem Statement: The Keisei DRL project's evaluation mechanism is currently slow, brittle, and not well-integrated with the training loop. Evaluations run sequentially in the main process, causing long training pauses [16], and require saving/loading model files for communication [2]. The system for choosing opponents and tracking performance (Elo ratings) is rudimentary, limiting the insight into training progress. The evaluation code is spread across callbacks and utility functions, making maintenance and extension difficult. As we plan to scale up experiments and perhaps use more advanced evaluation (multi-opponent testing, faster feedback), the current design poses a bottleneck.

Decision: We will implement a new **EvaluationManager** class within the Trainer and an **OpponentPool** to manage past checkpoints and opponents. The Trainer will use these to conduct in-process evaluations in a modular way. Key aspects of the decision include using Python multiprocessing for parallel game execution to utilize multiple CPU cores and reduce evaluation time, and maintaining an internal pool of opponents (with Elo ratings) to allow continuous, systematic benchmarking of the agent. We will phase out the old file-based, sequential evaluation in favor of this new architecture. This decision aligns with project goals of faster iteration (as noted by planned 10-20x eval speedups) and improved research rigor (by enabling tournaments and more reliable performance metrics).

Consequences:

- *Pros:*
- **Dramatically Faster Evaluations:** By running games concurrently, we expect to cut evaluation time from hours to minutes for a given number of games [38]. This means training runs will spend far less time idling for evals, and researchers get performance feedback much sooner.
- **Continuous Performance Tracking:** The OpponentPool with Elo ratings allows us to continuously track the agent's strength relative to past versions. We can detect if a new policy is truly better or if training has regressed (e.g., if Elo drops). This addresses the single-opponent limitation by providing a more holistic evaluation.

- **Better Integration & Reliability:** The evaluation becomes part of the training process, using the same logging and error-handling facilities. We remove ad-hoc subprocess behavior – no more separate WandB runs or manual file juggling (except where necessary for worker processes). This reduces points of failure and makes the code easier to understand. If something does go wrong in evaluation, the EvaluationManager can catch it and report it without crashing the entire training.
- **Extensibility:** With a dedicated EvaluationManager, adding new features like evaluating on different tasks, using different metrics, or implementing new selection strategies becomes localized changes. For instance, if we wanted to sometimes evaluate against a human or external engine, we could extend OpponentPool to include that. Or we could implement different evaluation frequencies or criteria easily (e.g., only evaluate when a certain reward threshold is met) by adjusting the callback/manager logic, all without hacking the training loop. The modular design opens the door to distributed evaluation in the future (e.g., launching evaluation on another machine or GPU) by swapping out the backend of EvaluationManager, if needed.
- **Testability:** We can write focused tests for the EvaluationManager and OpponentPool. They can be instantiated in isolation and fed dummy agents to verify behavior, which was not straightforward with the previous tightly coupled approach. This means we can more confidently iterate on the evaluation code.

- *Cons:*

- **Increased Complexity:** The introduction of concurrency (multi-processing and possibly multi-threading for background tasks) adds complexity to the system. Care must be taken to avoid race conditions (e.g., training updating a model while a worker is using it) and to manage resources (ensuring processes terminate, avoiding memory leaks from copying models, etc.). Debugging issues in parallel code can be harder than in sequential code.
- **Higher Memory/CPU Usage:** Running multiple game simulations in parallel will consume more CPU cores. If the system runs on a machine with limited cores, this could potentially slow other parts (though we expect net gain since training spends a lot of time waiting on environment steps, which can overlap with eval). Also, storing multiple opponents in memory or having several copies of the model loaded by workers will increase RAM usage. We need to monitor resource usage and possibly provide config knobs (like limiting num_workers or offloading older models from memory).
- **Migration Effort:** This refactor touches many parts of the code (Trainer, callbacks, evaluation modules). There's a risk of introducing bugs during transition. Existing training runs or scripts might assume the presence of certain fields (like `trainer.execute_full_evaluation_run` or the specific logging format of eval results). We must ensure the new system is well-tested and update any dependent code or documentation. There might be a learning curve for contributors to get familiar with the new classes.
- **Behavior Changes:** While we aim for the new system to be a superset of old functionality, some behavior will change slightly. For example, logging will now intermix eval logs with training logs (which is usually fine, even desirable). Elo ratings will now update more continuously rather than only using single comparisons; this is an improvement but results (like which model is considered "best") might differ from the old approach. We need to validate that such differences are acceptable or better align with project goals (e.g., using Elo rather than raw winrate against a recent model to decide promotion of new models).

Alternatives Considered:

1. *Status Quo with Incremental Optimizations:* We considered simply increasing the evaluation frequency or number of opponents in the current setup without architectural changes. This was deemed insufficient because running more games sequentially would only further slow training, and adding more opponent variety would require even more complex callback logic (diminishing returns given the current design). The fundamental issues of single-threaded execution and code brittleness would remain.

2. *External Evaluation Service:* Another option was to run a completely separate process (or microservice) for evaluation. For example, the training process could send the latest model to a separate evaluator program (via sockets or filesystem), which would then run games and return results asynchronously. While this could isolate evaluation load and failures from the trainer, it introduces a lot of complexity in inter-process communication and coordination (essentially we'd be building a small distributed system). It also still requires model serialization and potentially redundant GPU resources. Given that our evaluation is closely tied to training progress, keeping it in-process (with controlled parallelism) is more straightforward and efficient for our case.

3. *Post-Training Evaluation Only:* We also discussed foregoing periodic evaluation altogether and simply training the model to the end, then doing a large evaluation or tournament. This would simplify training but at the cost of losing interim feedback – which is critical for monitoring training stability (e.g., catching mode collapse early) and for selecting models (if we only evaluate at the end, we might miss that an earlier checkpoint was actually better). The team decided continuous evaluation is too valuable to drop; we want to know during training how the agent is improving.

4. *Use existing Parallel Self-Play Workers for Evaluation:* Since the training system already has a `ParallelManager` for self-play, one could try to reuse those worker processes to also perform evaluation games (maybe by sending a special message to them to play a game without learning). This might save on launching separate processes. We opted against entangling training and evaluation roles for now, to keep concerns separate. Self-play workers have a different lifecycle and requirements (they continuously generate experience and sync gradients), which we didn't want to interrupt or complicate with evaluation duties. However, this remains a potential optimization after the refactor: in the future, idle self-play workers could indeed be used to play evaluation matches on the side. The chosen design does not preclude this – it could be an extension where EvaluationManager interfaces with ParallelManager.

Mitigations for Drawbacks: To address the added complexity, we will invest in robust testing (unit tests and integration tests as outlined) and debugging tools (perhaps enabling more verbose logging for evaluation threads/workers). We will also implement the change in phases (as above) to catch issues early. Resource usage will be configurable – e.g., we can allow setting `eval_num_workers` in config to limit CPU use, and the OpponentPool size is configurable to cap memory. If needed, we can default to a conservative number of workers (like 2 or 4) which the user can increase if they have more cores. We'll document how the new system works so users understand the changes (especially in how results are reported).

Decision Outcome: Approved – We will proceed with refactoring the evaluation pipeline as described. This decision is expected to greatly improve the system's evaluation efficiency and maintainability, unlocking faster experimentation and more insightful training monitoring. We will monitor the

implementation closely and be prepared to adjust (for example, tuning the parallelism level or improving thread safety) as we gather empirical data with the new system.

---

1 2 3 4 5 6 11 23 25 30 32 33 callbacks.py
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/keisei/training/callbacks.py

7 8 13 14 18 19 20 21 24 evaluate.py
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/keisei/evaluation/evaluate.py

9 10 15 28 loop.py
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/keisei/evaluation/loop.py

12 train_legacy.old
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/deprecated/train_legacy.old

16 17 26 27 38 PARALLEL_EVALUATION_BUSINESS_CASE.md
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/deprecated/obsolete_docs/PARALLEL_EVALUATION_BUSINESS_CASE.md

22 29 trainer.py
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/keisei/training/trainer.py

31 training_loop_manager.py
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/keisei/training/training_loop_manager.py

34 35 36 37 evaluation_elo_registry.md
https://github.com/tachyon-beep/shogidrl/blob/df65de109d73b687ed127dd3a54a286588f4c85b/docs/component_audit/evaluation_elo_registry.md