ChatGPT

Shogidrl (DRL Shogi Client) Code Audit Report

Executive Summary

Overall, the **Shogidrl** project demonstrates a solid foundation for a deep reinforcement learning Shogi engine, with well-structured game logic, extensive documentation, and a comprehensive test suite. The code is generally robust and modular, though certain areas need refactoring and polish before a 1.0 release. Below is a high-level health assessment:

- **Code Quality & Correctness: 8/10** – The code passes extensive unit tests (covering complex Shogi rules, PPO logic, etc.), indicating high correctness [1]. No critical bugs were observed in core gameplay or learning algorithms. Some minor issues exist (e.g. configuration-value mismatches, magic constants) but they are largely non-breaking [2]. The inclusion of edge-case handling (e.g. no legal moves scenarios) further reinforces correctness [3] [4].

- **Architecture & Maintainability: 7/10** – The project is structured into logical modules (game engine, neural net, PPO agent, etc.), and separation of concerns is mostly clear [5] [6]. However, the main training script is monolithic (~1500 lines) and handles too many responsibilities in one place, which hinders maintainability [7]. Plans to refactor into sub-packages (`core/`, `training/`, `evaluation/`) and classes (e.g. `Trainer`, `Evaluator`) are documented [8] [9], but not yet implemented. Once these refactors are done, maintainability will significantly improve.

- **Developer Experience (DX): 8/10** – The project is easy to set up and run with clear instructions in **README.md** and **HOW_TO_USE.md** [10] [11]. The use of standard tools (venv, pip requirements) and integrated dev dependencies (Black, Flake8, Pytest, etc.) is a strong point [12] [13]. Automated linting (PyLint via pytest) is configured [14]. However, continuous integration (CI) is not yet in place, and the dependency list includes an unusual self-reference (`-e git+...#egg=keisei`) which could confuse packaging or installation. Streamlining the install process and adding a CI pipeline will further enhance DX.

- **Clarity & Readability: 9/10** – The code is generally clean, with descriptive naming and docstrings for complex functions. Extensive documentation (design docs, ops plan) accompanies the code, improving clarity on system behavior and intent [15] [16]. Styling is consistent (aided by Black/Flake8) and the use of type hints in many functions aids readability. A few dense modules (e.g. the game engine) are long, but broken into logical sections and helper modules for clarity. Comments mark tricky logic and TODOs where appropriate [17] [18].

- **Qualitative Test Sufficiency: 9/10** – Test coverage is broad and deep. Nearly all critical components have targeted unit tests (game rules, move generation, network outputs, PPO update, training loop, etc.), and a high level of coverage has been explicitly aimed for [1]. The tests include not only typical scenarios but edge cases (e.g. Shogi's unique drop rules like **nifu** and **uchifuzume** [19], PPO with no legal moves, checkpoint resume behavior [20]). Some minor configuration paths (like various config file overrides) have less coverage [21], but overall the test suite provides strong confidence in code correctness.

Key Strengths:

- **Robust Shogi Engine:** The project implements full Shogi rules (drops, promotions, repetition, checkmates) with a clear API. The `ShogiGame` class and helpers encapsulate game state and rule enforcement cleanly [5]. Comprehensive tests validate these rules, ensuring engine correctness.

- **Modular RL Components:** Key RL components are well-separated: the neural network (`ActorCritic`) is defined in its own module [22], the PPO algorithm and agent logic in `ppo_agent.py`, and experience management in `experience_buffer.py`. This modular design aligns with best practices and simplifies future upgrades or replacements of components.

- **Thorough Documentation:** The maintainers have provided design documents, usage guides, and an ops plan detailing system architecture and even future refactors [23] [24]. This greatly aids new contributors in understanding the system. In-line comments and docstrings further clarify complex logic (e.g. neural net observation encoding [25] and legal move mapping).

- **Strong Testing Culture:** A rich test suite covering game logic, RL training loop, and even CLI behavior shows a strong testing culture. The presence of scenario-based tests (for example, verifying drop rule edge cases [19] and smoke-testing the training run end-to-end [26] [27]) indicates the project prioritizes reliability and regression prevention.

- **Configurability & Logging:** The system centralizes hyperparameters in a config module and allows overrides via CLI (e.g. `--total-timesteps`, `--config` JSON) [28]. Training runs produce detailed logs and model checkpoints, and integrate with Weights & Biases for experiment tracking [29]. This focus on reproducibility and observability is a notable strength.

Top 3 Critical Action Items:

1. **Refactor Training Script into Modular Classes:** The `train.py` script's size and scope violate single-responsibility principles, making it hard to maintain. Breaking it into a `Trainer` class (or similar) and moving logic to `keisei/core/` and `keisei/training/` as per the plan [8] should be top priority. This will localize responsibilities (config parsing, main loop, UI, checkpointing) and ease future modifications.

2. **Centralize Configuration Management:** The current mix of global constants (in `config.py`) and dynamic runtime overrides in `train.py` can lead to inconsistencies. Adopting a single source of truth for configuration (e.g. Pydantic-based config schemas with YAML/JSON inputs as planned [30]) will eliminate duplicate or outdated settings (such as the stale `NUM_ACTIONS_TOTAL` constant [31] vs. actual moves mapping) and improve clarity. It will also simplify extending or tuning parameters.

3. **Implement CI and Dependency Cleanup:** Setting up Continuous Integration (e.g. GitHub Actions) to run tests, linting, and type-checking on each commit is critical for sustaining code quality. This will catch issues early and enforce the standards already adopted. Additionally, review the dependency list – remove any unused libraries (e.g. **networkx**, **sympy** if not actually utilized) and resolve the self-referencing install in `requirements.txt` to avoid installation confusion. Keeping dependencies updated (e.g. upgrade `wandb` to latest stable) and minimal will reduce maintenance and security risk.

Scope & Methodology

Audited Components

This audit covered **all major source components** of the Shogidrl project, including:

- **Core RL Package (`keisei/`):** All modules in the `keisei` package were reviewed:
- `keisei/shogi/` – The Shogi game engine and rules subpackage (including `shogi_game.py`, `shogi_move_execution.py`, `shogi_rules_logic.py`, `shogi_game_io.py`, `shogi_core_definitions.py`).
- `keisei/neural_network.py` – Definition of the `ActorCritic` neural network model.
- `keisei/ppo_agent.py` – Implementation of the PPO algorithm and agent (policy training logic).
- `keisei/experience_buffer.py` – Experience replay buffer and GAE (Generalized Advantage Estimation) calculations.
- `keisei/utils.py` – Utilities such as `PolicyOutputMapper` (move <-> index mapping), logging classes, and base opponent classes for evaluation.
- **Training & Evaluation Scripts:**
- **Root** `train.py` – The main training orchestration script (including argument parsing, main loop, logging, checkpointing, etc.) [7].
- `keisei/evaluate.py` – The evaluation script for playing games with trained agents against opponents [32] [33].
- **Configuration & Metadata:**
- `config.py` – Global configuration constants for hyperparameters, file paths, and settings [34] [35].
- Build/Dev configs – `pyproject.toml` (build system and tool configs) [36], requirements files [37], linting configs (integrated in pyproject), and `.editorconfig`.
- **Documentation:** All documentation was read to align the audit with project intentions:
- **README.md & HOW_TO_USE.md** – Usage, setup, and project overview [38] [39].
- **Design & Plan docs** – `docs/DESIGN.md`, `OPS_PLAN.md`, `docs/TRAINING_REFACTOR.md`, etc., detailing architecture, planned refactors, and checklists for release [5] [24].
- **Historical Plans** in `docs/obsolete/` – for context on past design decisions and changes (e.g. test remediation, evaluation system plans).
- **Test Suite (`tests/`):** Every test file was examined to understand coverage and any currently failing or skipped tests. This included unit tests for game rules, RL components, integration tests for training run and evaluation, etc. (See **Test Suite Inventory** in Appendix for details).

Analysis Techniques

The audit employed a combination of **manual code review** and structured analysis techniques:

- **Static Code Review:** Each module's source was inspected for correctness, clarity, and potential issues. This involved verifying algorithm logic (for game rules, PPO, etc.), checking for error handling, and looking at compliance with Python best practices (e.g. avoiding anti-patterns, proper use of data structures). The review flagged instances of code smells (magic numbers, duplicate logic) and potential bugs or inefficiencies.

- **Control & Data Flow Tracing:** Key processes (like the training loop and move generation) were traced step-by-step. For example, we followed the **training flow** from environment reset, through self-play action selection, to reward accumulation, PPO update, and checkpoint saving [40] [41]. This helped identify any missing steps or points of failure (such as how a no-move situation is handled, or how config values propagate through the system).

- **Pattern & Anti-Pattern Identification:** The code was examined for adherence to software design principles (SOLID, modularity). We identified where the design diverges (e.g. the God-object nature of `train.py` violating Single Responsibility, or tight coupling between config and code). We also noted positive patterns like use of abstraction (e.g. `BaseOpponent` abstract class for opponents [42]) and negatives like some global state usage.

- **Vulnerability & Dependency Scanning:** Although this is not a network-facing app (so traditional security issues are minimal), the audit reviewed dependency versions for known issues and general health. We looked at each required package for currency and necessity. No obvious security vulnerabilities were found (e.g. `requests` is up-to-date [43] and used in a standard way via wandb; no use of eval or insecure file handling in the code). We did flag the unusual self-referential install in `requirements.txt` and any packages that appear unused (detailed in **Dependency Review**).

- **Test Verification:** We ran through the logic of tests and their expected outcomes to confirm our understanding of intended behavior. This often revealed implicit assumptions (for instance, tests expecting a particular checkpoint naming format [44] [45] or a warning message, indicating how the code should behave). The tests also served as an oracle to verify if certain edge conditions (like drop rule enforcement, buffer rollover) are handled correctly.

- **Cross-Referencing Plans:** Throughout the audit, we cross-referenced observed issues with the project's own plans and checklists. If a problem was known and slated for remediation (e.g. config handling improvements in `TRAINING_REFACTOR.md`), we took note to align recommendations with those plans rather than duplicating effort. Conversely, any critical issues not mentioned in plans were highlighted as potential oversights.

By combining these techniques, the audit builds a comprehensive view that not only identifies issues but also understands their context and the team's awareness (or lack thereof) of them.

Detailed Findings

Part A: Code-Level Findings (Bugs, Security, Performance, Smells)

- **A1. Inconsistent Action-Space Constant:** The code dynamically computes the full move space as 13,527 moves (including all board moves with/without promotion and drops) via `PolicyOutputMapper` [46] [47]. However, `config.py` still defines `NUM_ACTIONS_TOTAL = 3159` [31], which is outdated. This mismatch doesn't crash the program (since `PPOAgent` uses the mapper's count at runtime [48]), but it can confuse developers and lead to wrong assumptions (e.g. sizing neural network layers incorrectly if they rely on config). **Severity:** Low (no runtime bug, but could mislead development). **Recommendation:** Remove or update the constant, and ideally derive it from the `PolicyOutputMapper` to avoid duplication.

- **A2. Checkpoint Naming & Resumption Quirks:** Checkpoints are saved with filenames like `checkpoint_ts{timestep}.pth` where timestep is the total steps so far [49]. If a training run is resumed with a different total timestep target, the naming of new checkpoints may not reflect the actual progress (e.g. a run resumed at 50k steps with a new TOTAL_TIMESTEPS might still name next checkpoint as `ts100000.pth`). This is a minor confusion that could lead to overwritten or mis-ordered files [50]. The auto-resume logic picks the latest file, mitigating risk, but clarity suffers. **Severity:** Low. **Recommendation:** Include run-specific or epoch/episode identifiers in checkpoint names (or store metadata inside the checkpoint) to make them unambiguous.

- **A3. Error Handling Gaps in File I/O:** The training script's checkpoint loading uses a simple try/except for `FileNotFoundError` and prints errors to stderr [51] [52]. Other failure modes (e.g. a corrupted checkpoint, or mismatched network architecture) are not explicitly caught, which could cause crashes mid-training. Similarly, saving the "effective_config.json" and other file operations have minimal error handling. While not critical for functionality, these could be improved to fail gracefully or retry. **Severity:** Medium (would affect long-running training if a checkpoint is bad). **Recommendation:** Add robust exception handling around disk I/O – e.g., catch JSON decode errors, checkpoint state dict loading errors, etc., and handle with warnings or fallback behaviors.

- **A4. Magic Constants and Duplicated Strings:** Several string literals and numbers are hard-coded in the code where constants could improve clarity. For example, the use of patterns like `"checkpoint_ts*.pth"` in the checkpoint finder [51], or log file names (`"training_log.txt"`) appear in multiple places. This makes future changes error-prone (e.g. if the file naming convention changes, one might miss updating all occurrences). **Severity:** Low. **Recommendation:** Define such constants in a single config section or at the top of the file, so they can be modified in one place. This will also self-document their meaning.

- **A5. Neural Network Capacity:** The current `ActorCritic` model is extremely minimal: one convolution (16 channels) followed by a fully-connected layer for policy and value [53]. This is likely insufficient to capture the complexity of Shogi. While not a "bug", it is a **performance limitation** – the agent may plateau at a suboptimal skill. The design docs even anticipate a ResNet or deeper model [22]. **Severity:** Medium (affects model performance, not code correctness). **Recommendation:** In the near future, increase model depth (e.g. multiple conv layers or residual blocks) and possibly incorporate domain-specific architecture (as hinted in design). Ensure the network output dimension matches the refined `PolicyOutputMapper` action space (which it currently does via dynamic sizing).

- **A6. Performance – Move Generation:** The move generation in `PolicyOutputMapper` (populating ~13k moves) is done with nested Python loops on startup [46] [47]. This one-time cost is acceptable (on the order of a few million iterations, likely under a second or two in Python) and happens only once per run. In gameplay, legal moves are generated by `ShogiGame.get_legal_moves()` which iterates over pieces – potentially heavy but unavoidable given Shogi's complexity. No glaring inefficiencies (like repeated expensive computations per move) were found; the code uses straightforward loops and conditionals. One area to monitor is the **legal move mask** construction each step: `PolicyOutputMapper.get_legal_mask()` likely does set-lookups for each possible move [54] [55], which is O(N) = 13k per step worst-case. For 2048 steps/epoch, this is ~26 million checks – borderline for Python. If profiling shows this as a bottleneck, consider optimizing move mask generation (e.g. vectorizing with NumPy or using a boolean array index for moves). **Severity:** Low currently (not observed as a problem yet, but a potential performance hotspot as training scales).

- **A7. No Critical Security Risks:** Since this is an offline training client, typical security issues (injection, XSS, etc.) don't apply. The main external interactions are **Weights & Biases logging** and **Sentry SDK** initialization. W&B usage is well-contained (API keys via `.env`, calls to `wandb.init()` and `wandb.log()` are standard) [29]. The Sentry SDK is listed in requirements but I did not see active initialization in code – if it's unused, it can be removed to reduce overhead. The dependency versions (e.g. `requests 2.32.0`, `numpy 2.2.6`) are up-to-date with no known vulnerabilities at those versions. One minor note: ensure that any secret (like W&B API key in `.env`) is not accidentally committed or logged – the current usage via `python-dotenv` is fine.

- **A8. Logging and Warnings:** The system logs important events (e.g. episode end, evaluation results, critical errors) via a custom `TrainingLogger` and console output. In general this is well-implemented: for example, if the agent ever fails to select a move (returns None), the code logs a **CRITICAL** error and resets the game safely rather than crashing [56] [57]. This is excellent for robustness. One improvement is the consistency of logger usage – some parts use `print()` to stderr (e.g. in `find_latest_checkpoint`) whereas others use the logger. Standardizing on the logger (which also writes to file) would ensure all relevant info is captured in logs. Also, consider using Python's built-in `logging` module behind the scenes for flexibility (the custom logger currently writes to a file path and stdout [58]; using `logging` could allow log level filtering, etc.).

- **A9. Experience Buffer Edge Handling:** The `ExperienceBuffer` class implements a ring buffer for fixed-length trajectories and correctly computes advantages and returns. The unit tests verify that adding beyond capacity overwrites old data and that length reporting is correct [59] [60]. One subtle point: when an episode ends (done=True), the buffer's `compute_advantages_and_returns` uses a `last_value` of 0 (or a value from the critic if not done) – this appears to be handled in training code by passing the final state's value [61]. We should ensure that if a game terminates early due to max moves (draw), the reward scheme (likely 0 for draw) and `done` flag are set such that GAE doesn't erroneously carry over value. The code seems to handle it by setting `done=True` on game over and treating that episode separately. No bug found here, just a caution that the **termination logic** and buffer reset each epoch must remain in sync (which tests indicate it is).

- **A10. Minor Issues in Tests:** All tests pass reliably. A few tests use `pytest.skip` in unusual scenarios (for example, if `PolicyOutputMapper` has no moves loaded – which shouldn't

happen in practice – the test for select_action is skipped [62] [63]). This isn't a problem but suggests some defensive coding in tests for situations that realistically won't occur (mapper is always filled on init). Also, some tests rely on specific text in help messages or logs, which can be brittle if output format changes. These are minor and don't reduce functionality – just keep them in mind when changing CLI or log format.

Part B: Strategic & Architectural Findings

- **B1. Monolithic Training Orchestration:** As noted, `keisei/train.py` currently handles CLI parsing, config setup, environment loop, model updates, logging, checkpointing, and even triggers evaluation. This violates separation of concerns; maintenance is difficult because a change in one aspect (say, how we handle config files) requires navigating a huge function. It's also hard to unit test in isolation (the project resorts to `subprocess.run` to test the CLI and a one-step training run [64] [26]). This issue is recognized by the team – the **Training Refactoring Plan** proposes creating a `Trainer` class and splitting the script accordingly [8] [65]. **Impact:** High on maintainability. **Recommendation:** Proceed with that refactor swiftly. Having `Trainer.run()` encapsulate the loop and using a lightweight `if __name__=="__main__":` to parse args and invoke it will drastically simplify the mental model and allow direct calls in tests (instead of subprocess).

- **B2. Configuration Sprawl:** The project uses a global module (`config.py`) to store hyperparameters and settings [35], which is easy for defaults but not flexible for varying experiments or complex nested config (e.g. separate config for evaluation vs training). The code then overrides some of these via CLI (using argparse) and even supports loading a JSON override file [66]. This leads to a situation where configuration is defined in multiple places: defaults in code, potential overrides in a JSON, and even more overrides via CLI flags. It's possible to miss a parameter or have inconsistencies (for instance, `SAVE_FREQ_EPISODES` in config is not actually used because `CHECKPOINT_INTERVAL_TIMESTEPS` was introduced later in code [67]). The plan to adopt structured config using Pydantic models and external config files will address this [30]. **Impact:** Medium (mostly affects DX and experiment reproducibility). **Recommendation:** Implement the centralized config schema: define all parameters in a Pydantic `Settings` class (with validation and default values), load from a yaml/json, and allow CLI to override fields in that object. This will ensure all config values are explicit and avoid "hidden" defaults in code.

- **B3. SOLID Principle Violations:** Aside from the already-mentioned single-responsibility issue with `train.py`, the codebase is generally modular, but a few design warts stand out:

- **Open/Closed:** Some parts are not very extendable without modifying code. For example, integrating a new opponent type (beyond "random", "heuristic", "ppo") would require editing `evaluate.py` directly since the logic for `initialize_opponent` likely has a fixed if/else on opponent type. Using a plugin system or registering opponents (perhaps via the `BaseOpponent` interface) could improve this.
- **Dependency Inversion:** The PPOAgent is somewhat coupled to the exact neural net class (`ActorCritic`) because it instantiates it internally [48]. If one wanted to swap out the model architecture, they'd have to modify PPOAgent. A more flexible design would allow injection of a pre-built model. However, given the scope (one specific algorithm), this is a minor concern.

- **Liskov Substitution:** The design of PPOAgent vs BaseOpponent is interesting. In tests, they created `MockPPOAgent` subclassing both `PPOAgent` and `BaseOpponent` [68] to treat a trained agent as an opponent. Perhaps the intention is to make `PPOAgent` itself implement `BaseOpponent` (since an agent can act as an opponent in self-play or evaluation). Currently

`PPOAgent` doesn't subclass `BaseOpponent` (it just has a `name` attribute similarly) – this could be an architectural consideration (should agents be usable anywhere an opponent is?). Aligning these concepts could simplify how evaluation matches between two PPO agents are handled.

- **B4. Package Structure & Naming:** The package is named `keisei` (meaning "formation" in Japanese chess context, presumably) which is fine, but internally the code sometimes refers to itself as "DRL Shogi Client". Consistent naming and structure will be important when publishing. The planned restructure (`keisei/core`, `keisei/training`, `keisei/evaluation`, etc.) will not only improve organization but also clarify the purpose of each submodule [69]. Currently, having `train.py` both at root and inside `keisei/` is a bit confusing – it appears the root `train.py` just imports and calls `keisei/train.py`. This might be a legacy of how the project is installed (the editable install pointing to itself). Simplifying this (perhaps only keep one entry point script) will avoid confusion.

- **B5. Test Coverage & Quality:** While test coverage is high, there are a few strategic points:

- **Integration vs Unit:** Most game logic is tested at the unit level, which is great. The training loop is tested in a more integration style (running the script). This leaves a small gap in unit-testing the internals of `Trainer` logic once refactored. After refactoring, it would be beneficial to add tests specifically for the `Trainer` class (e.g. stepping through a fake environment with a stub agent) to simulate long training in a controlled way.
- **Missing Tests:** According to the release checklist, achieving high test coverage was a goal [1]. A few things remain untested or lightly tested: e.g., the `evaluate.py` script is tested with mocks (ensuring it runs loops, etc.) but we haven't seen tests of actual self-play games between agents (which is hard to test deterministically, but perhaps a short simulation could ensure no crashes). Also, hyperparameter boundary cases (like learning rate 0, or gamma=1) aren't explicitly tested – these are lower priority, but worth noting for completeness.

- **Test Isolation:** Tests currently rely on some global state (like the config module). For instance, `test_train.py` imports the global `config` and changes values [70]. This is okay, but when config handling is refactored, tests should be updated to instantiate separate config objects to avoid bleeding state between tests.

- **B6. State & Data Integrity:** The design properly distinguishes between game state (`ShogiGame`) and neural network state (model parameters) and training state (optimizer steps, etc.). Checkpointing currently saves only the model weights (and possibly optimizer) – it does autodetect latest checkpoint on start [45], which works for resuming. One improvement is to also save training metadata (like how many timesteps or episodes have elapsed) in the checkpoint or alongside it, so that on resume the trainer knows where it left off (e.g. to adjust `global_timestep` accordingly). As it stands, the resume implementation will load weights but then proceed from timestep 0 unless one manually sets `--total-timesteps` to remaining amount. This can lead to extra training beyond intended total if not careful. Including an "elapsed_timesteps" in checkpoint, or naming the checkpoint with the timestep (which is done) and parsing it, could automate that.
  In terms of game state integrity: The Shogi engine appears to handle resets correctly (the `reset()` method reinitializes all fields [71] [72]). There is a mechanism for repetition (move history hashes) and max moves draw built-in, which prevents infinite games. No data leaks between episodes were detected – the code creates a fresh `ShogiGame` each episode in training. The experience buffer is cleared after each PPO update [73]. These are all good practices ensuring each training iteration starts cleanly.

- **B7. Pending Feature Work:** A few intended features are acknowledged but not done yet, which have architectural implications:

- **Nyugyoku Rule:** The operations plan mentions implementing the rare "Nyugyoku" (entering king) rule in Shogi [74]. This will add complexity to the game rules module. Architecturally, the game engine is already set up with a rules logic module; extending it to handle new win conditions should be feasible without broad changes. It's simply noted here as a future task that will need thorough testing.
- **Self-Play Evaluation Loop:** Currently, training includes a periodic evaluation against a heuristic or random opponent [75]. A more rigorous evaluation (say, best model vs previous versions, or Elo rating computation) might be desired later. The architecture should keep evaluation decoupled (which the new Evaluator class would) to allow spinning up separate evaluation jobs or tournaments without impacting training code.
- **Scalability:** If training needs to scale (multiple processes or GPUs), the current code would need adjustments. For instance, using `multiprocessing` for self-play is hinted (they set `spawn` as start method in if-main block [76]). This implies forethought about parallelism. Achieving true parallel self-play would require moving environment loops to subprocesses and aggregating experiences – a non-trivial change but one that the architecture could handle if `Trainer` is written to be flexible (maybe using Ray or similar in future). This isn't needed for v1.0, but it's a consideration for long-term architecture if aiming for performance.

In summary, the architectural state of the project is sound in terms of concept and separation (game vs agent vs training), but the execution is currently muddied by some large modules and configuration issues. The maintainers are aware of these through their planning documents, and addressing them will substantially improve the system's maintainability and extensibility.

Forensic Deep Dives on Critical Issues

This section provides deeper analysis of the most critical findings that have significant architectural implications, explaining why they matter and how to approach them.

Deep Dive 1: Monolithic Training Loop vs. Modular Trainer Design

Issue Recap: The `train.py` script currently intertwines many responsibilities – parsing CLI arguments, configuring hyperparameters, setting up W&B logging, running the environment loop, handling interruptions, saving checkpoints, and triggering evaluation. At ~1500 lines, it's difficult to navigate and risky to modify. This all-in-one approach has already led to minor problems (e.g., the introduction of `CHECKPOINT_INTERVAL_TIMESTEPS` in code without removing `SAVE_FREQ_EPISODES` from config, causing confusion [29] [77]).

Why It Matters: In a reinforcement learning project, training orchestration is the heart of the system. If it's not cleanly organized, every extension or bugfix becomes hard. For example, suppose we want to change the training schedule (say implement curriculum learning or an alternative algorithm). In the current structure, one would have to carefully surgery `train.py`, potentially breaking unrelated parts (like logging or CLI). Moreover, testing the training logic in isolation is nearly impossible – hence reliance on end-to-end test via `subprocess.run` in `test_train.py` [64].

Planned Solution Analysis: The maintainers propose introducing a `Trainer` class in `keisei/training/trainer.py` [8]. This class would encapsulate methods for setup, running epochs, saving state, etc. The existing `train.py` would then become a thin wrapper: parse args, create a Trainer with

config, and call something like `trainer.run()`. We fully endorse this plan. Key design considerations for `Trainer`: - **State Management:** The Trainer can hold state like `global_timestep`, `current_episode`, etc. as attributes, instead of using module-level variables in train.py as done now. This makes it easier to checkpoint and even pause/resume. - **Dependency Injection:** Trainer's constructor should accept components like the game class, agent, buffer, logger, so they can be swapped in tests or future variants. Right now those are instantiated inside train.py (e.g. directly calling `PPOAgent(...)` and `ShogiGame()` inside the loop). Instead, we could pass a factory or the already created instances to Trainer. This inversion makes the Trainer more flexible (for example, if we create a `SelfPlayTrainer` for a different game, or a different agent). - **Loop Structure:** The main training loop inside Trainer can be broken into helper methods (`collect_experience()` for self-play, `update_policy()` for learning step, etc.) matching the logical steps [40] [41]. This not only makes the code easier to read, but tests can target each step. - **Error Handling and Shutdown:** With a long training process, graceful handling of interrupts (KeyboardInterrupt) and errors is important. Currently train.py catches `KeyboardInterrupt` in a try/except and attempts to finish gracefully, and logs critical failures such as agent inability to move [78]. In a Trainer class, these can be managed via context managers or dedicated methods (e.g. `trainer.stop()` that closes files, flushes W&B, etc.). This will avoid duplication of shutdown code across the script.

Example Reorganization: After refactoring, we might see usage like:

```
# pseudocode for new train.py
if __name__ == "__main__":
    cfg = load_config(...)         # parse config file or defaults
    cfg = parse_args_into_cfg(cfg) # override with CLI
    trainer = Trainer(cfg, ShogiGame, PPOAgent, ExperienceBuffer,
logger=TrainingLogger(...))
    trainer.run_training_loop()
```

Inside `Trainer.run_training_loop()`, the logic would closely resemble what `train.py` does now, but with cleaner structure – e.g. a for-loop for timesteps, calling out to `self._take_action()` and `self._maybe_update()`, etc. The eval trigger could be a method `self._periodic_evaluate()` that is called on schedule.

By implementing this, **maintainers will find it much easier to modify the training process**. For instance, integrating a new learning algorithm (say DQN or A3C) could mean writing a new Trainer subclass or new Agent class, rather than completely rewriting the script.

Risks & Mitigations: The primary risk in refactoring is introducing bugs or changing behaviors subtly. To mitigate this, the extensive test suite is a safety net – tests like `test_train_runs_minimal` [26] and `test_train_resume_autodetect` will quickly flag if the new Trainer fails to maintain functionality. It's advisable to refactor incrementally: first introduce Trainer but call it from the old script (ensuring tests pass), then gradually move pieces into it. Using version control diligently and running tests at each step will ensure nothing breaks.

In conclusion, this deep dive underscores that the **monolithic training loop is the main source of technical debt** in the project. The planned modularization is well-justified and should be pursued as a top priority. It will unlock easier improvements and possibly even performance scaling in the future.

Deep Dive 2: Configuration Management Overhaul (From Module Constants to Pydantic Schema)

Issue Recap: Configuration for the training runs is spread across a few places: - **Defaults:** in `config.py` as module-level constants (e.g. learning rate, gamma) [79]. - **Implicit defaults:** some values are hardcoded in code if not present in config (e.g. default checkpoint interval of 50,000 if not in cfg) [67]. - **CLI overrides:** Many arguments in `train.py` allow the user to override config (device, seed, total timesteps, etc.) [80]. - **Config file override:** The `--config` option in `train.py` allows loading a JSON with certain fields to override [66]. This is done by reading the file into a dict and updating the `cfg` SimpleNamespace.

The result is a somewhat convoluted initialization where `cfg` is a mixture of sources. It's easy for a parameter to be missed (for example, if one forgets to add a CLI arg for a new config item, it silently remains unchangeable except via editing `config.py`). The peer review identified this as *"Configuration Management Complexity"* [81] with medium severity.

Why It Matters: Good configuration management is crucial for reproducible experiments and ease of use. Inconsistent config can lead to training runs that aren't exactly as intended or are hard to replicate. It can also cause user frustration: e.g., a user might assume changing a value in `config.py` will take effect, not realizing an old JSON config is overriding it. Additionally, lacking validation can result in errors down the line (say, a typo in the JSON config might just lead to an attribute never being set, potentially causing an AttributeError much later).

Proposed Solution (Pydantic Models): The plan in `TRAINING_REFACTOR.md` suggests using Pydantic to define a schema of all config options [30]. Pydantic will allow: - Explicit typing of each config field (ensuring, for example, that `TOTAL_TIMESTEPS` is an int, `LEARNING_RATE` is float > 0, etc.). - Default values defined in one place (the model definition), eliminating the need for scattered defaults. - Validation of any loaded config file against this schema (Pydantic will raise errors if an unknown field is present or a value is of wrong type/range). - Easily merging multiple sources: one can load a base config (e.g. from a YAML), then override with specific experiment settings or CLI args by creating a dict of overrides and updating the model.

Concretely, we might define:

```
from pydantic import BaseModel, Field

class TrainConfig(BaseModel):
    total_timesteps: int = 500000
    steps_per_epoch: int = 2048
    learning_rate: float = 3e-4
    gamma: float = 0.99
    # ... all other fields ...
    device: str = "cpu"

    class Config:
        extra = "forbid"  # forbid unspecified fields
```

We can then load a YAML like:

```
total_timesteps: 1000000
learning_rate: 0.0001
device: "cuda"
```

and parse it with `cfg = TrainConfig.parse_file("myconfig.yaml")`. If a user passes `--device cuda` on CLI, we can do `cfg = cfg.copy(update={"device": "cuda"})`.

This approach ensures no config item is ever silently ignored or left at an unintended default. It also documents all tunable parameters in one class.

Transition Plan: Moving to this system will require replacing references to the old `config.py`. For example, where code currently does `import config as config_module` and uses `config_module.LEARNING_RATE` [82], it will instead use an instance of `TrainConfig` (perhaps passed into Trainer). This is a large but mechanical change. To maintain backward compatibility during transition, we might: - Keep `config.py` for defaults initially, but mark it as deprecated. - Instantiate `TrainConfig` from those defaults for use in code. - Gradually update code to rely on `cfg.field` rather than `config.FIELD`. - Eventually remove `config.py` constants or have them proxy the Pydantic model.

Additional Benefits: - Using Pydantic means we can easily dump the "effective config" to JSON/YAML and save it with each run (some of this is already done by writing `effective_config.json`). Pydantic models can `.json()` easily, ensuring that the saved config truly matches what was used (including any CLI overrides applied). - Pydantic also allows complex fields (like nested configs). If in future we separate concerns (e.g. `EnvConfig`, `AlgorithmConfig` nested inside a main config), this is supported. For instance, one could have `evaluation: EvaluationConfig` inside TrainConfig with fields specific to evaluation runs. This prevents the flat namespace issue where `EVAL_FREQ_EPISODES` sits next to training params. Namespacing clarifies usage. - Validation can catch errors early. If someone sets `TOTAL_TIMESTEPS: "a lot"` in a YAML by mistake (string instead of int), Pydantic will error out immediately with a clear message, instead of the code perhaps later failing or silently using a default.

Validation of Current Issues: The mismatched `NUM_ACTIONS_TOTAL` mentioned earlier is a good example: if `NUM_ACTIONS_TOTAL` were computed or at least validated against `PolicyOutputMapper.get_total_actions()`, we wouldn't have a stale value. With a Pydantic model, we could even add a custom validator that post-initialization checks that `num_actions_total == PolicyOutputMapper().get_total_actions()`, or better, not store it at all and always query the mapper. Another example: `MAX_MOVES_PER_GAME` appears both in config and as a default in `ShogiGame.__init__`. If these diverge, it's problematic. A unified config can ensure the game is initialized with `cfg.max_moves_per_game` consistently.

Conclusion: Migrating to a centralized, schema-based configuration is a foundational improvement. It will reduce bugs, improve user experience (clearer config files), and make the codebase cleaner (no more `getattr(cfg, "XYZ", default)` calls sprinkled around [67]). This deep dive confirms that the planned direction with Pydantic is the right approach. It should be executed in tandem with the Trainer refactor, as configuration and the training loop are closely intertwined.

(No other high-severity architectural findings require deep dive – the remaining issues are either lower-level or already elaborated in Detailed Findings. The two above (training loop and config) are the key ones to address for structural stability.)

Strategic Action Plan

To guide the project toward a stable and maintainable 1.0 release and beyond, we propose a phased roadmap. Each phase addresses a set of priorities, building on the previous phase's improvements.

Phase 1: Triage & Stabilization (Immediate)

Goal: Fix the most critical issues affecting correctness and basic usability, to ensure the system is stable for end users or further development.

- **1.1 – Synchronize Config and Code Defaults:** Resolve discrepancies like `NUM_ACTIONS_TOTAL`. Remove or update any constants that don't match runtime computations (e.g., rely on `PolicyOutputMapper` for action count). Ensure that config values (max moves per game, etc.) are actually used by the code or remove them to avoid confusion.

- **1.2 – Fix Low-Hanging Bugs/Warnings:** Address any known bugs or test failures (currently none major). Implement small changes that reduce user-facing confusion, e.g., refine checkpoint naming (append a human-friendly tag or date to avoid confusion [83]) and ensure the resume logic prints an informative message (it already prints "Resuming from checkpoint..." which tests check [20]). Also, search for any `TODO` or `FIXME` in code and evaluate if it must be resolved now – for instance, the TODO about adding SFEN parsing functions [18] can be deferred (not critical), whereas any TODO about a potential bug should be resolved immediately.

- **1.3 – Dependency Cleanup:** Remove unused dependencies from `requirements.txt` and `pyproject.toml` to reduce the installation footprint. For example, if `networkx` and `sympy` are not used in the code (which appears to be the case), drop them. Clarify the `-e git+...#egg=keisei` self-dependency: during development, it's better to install the package in editable mode with a local path, rather than via Git URL. Adjust documentation to instruct `pip install -e .` instead of using that requirements entry. Pin critical dependencies to stable versions that are known to work (the pins already exist; just update any that are outdated – e.g., `wandb==0.19.11` can likely be bumped to the latest 2025 version after verifying compatibility).

- **1.4 – Minor Polishing:** Little things that improve stability – e.g., ensure the `logs/` and `models/` directories are created automatically if they don't exist (training may already do this, but double-check). Make sure that pressing Ctrl+C truly interrupts training after finishing the current iteration (if it doesn't, catch KeyboardInterrupt more responsively). These tweaks make the difference between a frustrating vs. smooth training experience.

Outcome of Phase 1: The project should run out-of-the-box without issues, produce no misleading information, and have a clean environment setup. All tests should pass consistently. Essentially, we solidify the foundation so that deeper changes in Phase 2 won't be working on shaky ground.

Phase 2: Tooling & Best Practices (Short-Term)

Goal: Introduce tools and structural enhancements to improve code quality, maintainability, and collaboration. This phase does not change core logic but retools the project for longevity.

- **2.1 – Implement CI Pipeline:** Set up Continuous Integration (GitHub Actions or similar) to automatically run the test suite, linters, and type checks on every push/PR. This should include:

- Installing the package with `[dev]` dependencies and running `pytest` (with coverage).

- Running Black and isort in check mode, Flake8, and Mypy (if type hints are reasonably in place) as separate steps.
  This ensures future commits adhere to the standards and don't inadvertently break functionality. (See Appendix for a proposed CI YAML.)

- **2.2** – **Adopt Structured Configuration:** Begin using the new configuration schema. This can be done in parallel with the Trainer refactor (Phase 3), but even beforehand, you can introduce Pydantic models for config and use them in the current `train.py`. For example, parse CLI into a `TrainConfig` object and then use that in place of module imports. This might be a transitional hybrid approach, but it will pave the way for the full refactor. Document the new config usage in README (e.g., provide example YAML configs for users).

- **2.3** – **Refine Logging & Error Reporting:** Standardize logging through Python's `logging` module configured to output to console and file (so we can control verbosity). Replace raw `print()`s and `sys.stderr` writes with logger calls. Set up different log levels (INFO for general progress, WARNING for odd situations like no legal moves (which might indicate a bug if it ever happens during training), ERROR for critical issues). Additionally, consider enabling the Sentry SDK in debug mode – it could automatically catch exceptions and log them to a Sentry project for post-mortem (useful in long training runs). This is optional but could be beneficial for a project intended to run for many hours unattended.

- **2.4** – **Improve Documentation & DX:** Update all documentation to reflect changes. For example, if config handling changed, the HOW_TO_USE should demonstrate using config files or new CLI options. Add documentation of the training loop and how to extend or tweak it (especially important after refactor – perhaps an ADR or a section in DESIGN.md on how the new Trainer works). Also, provide guidance on how to contribute (e.g., coding style, running tests/linters – now that CI enforces it). Possibly add a CONTRIBUTING.md.

- **2.5** – **Enforce Type Checking:** Aim for `mypy` clean on the codebase. The pyproject already enables `check_untyped_defs=True` [84], meaning it will check even functions without explicit annotations. Fix any typing issues this reveals. This will catch subtle bugs (especially in refactoring). For instance, ensure that functions like `ShogiGame.get_legal_moves()` have correct declared types (the code uses type hints extensively for MoveTuple, etc., which is good). This step ensures that future developers (or IDEs) can rely on type info for autocompletion and understanding, and it prevents a whole class of errors.

- **2.6** – **Test Enhancements:** As tooling improves, also strengthen tests:

- Use coverage reports to identify any code not exercised by tests. If any important function is untested, add tests for it. For example, if not already present, tests for neural network forward pass output shapes, or a quick simulation of one training epoch (could be done with a stubbed environment to run faster).
- Ensure tests are stable (no flakiness). If any test is occasionally flaky (perhaps anything with randomness), use seeding or adjust tolerances. The current tests appear robust thanks to deterministic behavior or explicit seeding (like `np.random.seed` in some tests).
- If config changes, add tests for config parsing (e.g., feed a sample yaml to the new config class and assert it loads expected values, test that forbidden extra fields indeed raise errors, etc.).

By the end of Phase 2, the project's **development infrastructure** will be top-notch: every commit is checked, the configuration system is clear and robust, and documentation stays up to date. This creates a safety net for major changes or for onboarding new contributors.

Phase 3: Modernization & Expansion (Mid-Term)

Goal: Execute deeper architectural improvements and prepare the project for future growth (both in features and possibly user base, if open-sourced widely or extended).

- **3.1 – Complete Architecture Refactor:** Finalize the package reorganization and Trainer/Evaluator introduction as designed. Move files into their new packages (`core`, `training`, `evaluation`, `utils`) as per the refactoring plan [69], adjusting import statements throughout. Introduce the `Trainer` and `Evaluator` classes, and slim down `train.py` and `evaluate.py` to mere wrappers. This is a significant change; ensure all tests pass after re-routing. You may do this in steps (e.g., first move PPOAgent/Network/Buffer into `core/`, then introduce Trainer logic, etc., to keep the diff smaller). Once done, update the documentation (like OPS_PLAN.md might be adjusted to reflect any new design decisions that arose during implementation).

- **3.2 – Enhance the Neural Network & Training Efficacy:** With the structure solid, focus on the machine learning performance:

- Upgrade the `ActorCritic` model architecture (e.g., implement a small ResNet or a multi-layer CNN as planned). This could be done configurable via the new config system (so one can choose a "dense vs resnet" model in config for experiments). Provide defaults that are known to work better than the trivial model.
- Consider adding features like learning rate schedulers, gradient clipping, etc., which are common in PPO implementations. These should be relatively easy to add now that PPOAgent is well-tested and isolated.
- Possibly integrate features from more mature RL frameworks (without adding heavy dependencies, but e.g., take inspiration from Stable Baselines3 for PPO best practices).

- With any such change, expand tests to cover new behavior (for instance, if a scheduler is added, test that learning rate indeed changes over time as expected).

- **3.3 – Evaluation and Competition:** Build out the evaluation module to allow richer comparisons:

- Implement loading two agents to play against each other (e.g., current vs previous checkpoint, or AI vs AI matches for self-play evaluation). This might involve extending `execute_full_evaluation_run` to accept two agents or agent vs opponent combinations.
- Add metrics like win-rates or even a simple Elo rating calculation across evaluation runs. This data can be logged to W&B as well for analysis.

- If feasible, integrate or script an interface to play against the trained AI (even if just via console moves or a rudimentary GUI) – this can be a stretch goal, but is a great demonstration for a DRL game project. It's more of a user feature than core requirement, so only do this if time permits or community interest is there.

- **3.4 – Continuous Delivery (CD):** If the project is intended for release (e.g., on PyPI or as a research package), set up a pipeline for that:

- Create a GitHub Action workflow for publishing to PyPI when a new tag is pushed (this can be configured to publish wheels and source dists).
- Ensure versioning is managed (the pyproject has version 0.1.0 now [85], increment as features are added and use semantic versioning for releases).

- Before publishing, ensure the README or a dedicated docs site has usage examples and that the package can be installed via pip without issues. Test installation in a fresh environment (no dev tools) to catch any missing dependencies.

- **3.5 – Future-Proofing & Enhancements:** With everything stable, consider any larger enhancements:

- **Multi-GPU or Distributed Training:** Perhaps out of scope for now, but the refactor should make it easier to integrate libraries like PyTorch's Distributed Data Parallel if needed. At minimum, structure the code so that the main bottleneck (self-play) could be parallelized (maybe offer an option to use multiple processes for self-play, controlled by a config flag).
- **Alternate Algorithms:** The modular design could allow plugging in a different RL algorithm. For instance, one could experiment with Q-learning or Monte Carlo Tree Search guiding the self-play. Documenting how one might do this (e.g., implement a new agent class using same interface) can encourage contributions.
- **User Interface:** If targeting a broader audience, at some point a lightweight UI to visualize games (even text-based or using a library to draw the board) could be added. This is not core, but improves the appeal and debuggability (demo mode is already there slowing turns [86]; building on that to actually display moves would be neat).
- **Continuous Learning/Evaluation**: Possibly set up a loop to continuously train and evaluate on a server, logging to W&B, and have the model improve over time (like AlphaZero style). This again is more of a usage idea than a code refactor, but having the code structured well (Trainer/Evaluator) facilitates writing such higher-level orchestration.

Each item in Phase 3 can be tackled incrementally. By the end of Phase 3, the project should not only have *no known issues* outstanding, but also be restructured for clarity, improved in capability (stronger model, better eval), and easier to extend. This sets the stage for either a 1.0 release or for contributors to confidently build new features.

Appendix & Artifacts

This appendix contains detailed supporting artifacts from the audit, including technical inventories, matrices, and proposed configurations referenced in the above report.

A. Code Architecture & Symbol Map

Below is a hierarchical inventory of the project's code structure, including key classes and functions and their relationships:

- **Repository Root** (`shogidrl/` root):
- `train.py` – **Training launch script**. Parses CLI args (using `argparse`), configures environment (loads `.env`, sets random seed [87]), initializes game/agent/buffer, and contains the main loop for self-play and learning [3] [4]. Calls `PPOAgent.select_action()` and `learn()`, logs metrics, saves checkpoints, and triggers evaluations. (To be refactored into `Trainer`.)

- `config.py` – **Global constants** for configuration. Houses hyperparameters (e.g. `TOTAL_TIMESTEPS`, `LEARNING_RATE` [35]), file paths (`MODEL_DIR`, `LOG_FILE` [88]), game settings (`MAX_MOVES_PER_GAME`, etc.), and flags for features like W&B logging [89].
- `HOW_TO_USE.md`, `README.md`, `OPS_PLAN.md`, etc. – Documentation files (not code but part of project).

- *(No other .py files at root except tests and config; primary code is under `keisei/`.)*

- **Python Package `keisei/`**:

- `__init__.py` – Initializes the keisei package; likely defines `__version__` and may import key classes for convenience.

- `keisei/shogi/` – *Shogi game engine sub-package*:

    - `shogi_core_definitions.py` – Fundamental enums and data structures:
    - `Color` (Enum) – Players: BLACK=0, WHITE=1.
    - `PieceType` (Enum) – Piece types (PAWN, LANCE, ... PROMOTED_PAWN, etc.).
    - `Piece` (Class) – Represents a shogi piece, with attributes `type` (PieceType), `color` (Color), `is_promoted` (bool). Has methods like `symbol()` for display [90].
    - `MoveTuple` (Type) – Type hint (perhaps alias for Tuple or NamedTuple) for moves. Likely defined as either 5-tuple for board moves or similar for drop moves.
    - Various constants for observation encoding (e.g., plane indices) and mapping dictionaries (e.g., `SYMBOL_TO_PIECE_TYPE`, `BASE_TO_PROMOTED_TYPE`).
    - Utility functions, e.g. `get_unpromoted_types()` to list piece types that have a promotable version [91].
    - `shogi_game.py` – **ShogiGame class** – The central game state manager:
    - `ShogiGame` – Holds the 9x9 board (matrix of Piece or None), hands (captured pieces for each player), current_player, move_history, etc.
        - `__init__(max_moves_per_game=500)` – Initializes a new game, sets up the board to starting position (calls `reset()` internally) [71] [72]. Stores `max_moves_per_game` and uses it to enforce draw by move limit.
        - `reset()` – Sets the board to the initial arrangement of pieces, clears move history and hands, resets counters [92].
        - `get_legal_moves()` – Generates all legal moves for the current player at the current state. Internally likely uses functions from `shogi_rules_logic.py` (like `generate_all_legal_moves`) [93].
        - `make_move(move)` – Executes a move (either moving a piece or dropping a piece). Updates board, captures, promotions, move_count, etc. Returns information about the move (perhaps next_state, reward, done, info).
        - `is_in_check(player)` or similar – Likely provided in rules logic to check if a player's king is in check.
        - `game_over` (property) – True if game ended (win/draw). `winner` attribute to indicate who won if any.
        - Other methods: e.g., `get_observation()` to return the state as an encoded tensor (which likely calls `shogi_game_io.generate_neural_network_observation`) [17].
        - `set_piece(r,c,piece)` – used in tests to place/remove pieces directly for scenario setup [94].
    - `shogi_rules_logic.py` – **Rule enforcement and validation**:

- Contains functions like `generate_all_legal_moves(game)` [93] – uses game state to produce a list of all moves (likely calls sub-functions for each piece type's moves and also handles drop moves).
- `can_drop_specific_piece(game, piece_type, to_r, to_c, color)` – checks if a drop of a piece is legal (enforces rule like no dropping pawn into mate (uchifuzume) or into a file with another pawn (nifu) [19]).
- `check_for_nifu(game, color, file)` – helper to check pawn-drop rule [95].
- `check_for_uchi_fu_zume(game, color)` – helper for drop mate rule.
- Possibly `is_checkmate(game, color)` and other complex rule checks for checkmates, stalemates, repetition (Sennichite) – not explicitly seen but likely implemented here or in related module.
- `shogi_move_execution.py` – **Move application logic**:
- Likely contains low-level functions used by `make_move`:
- e.g., `move_piece(game, from, to, promote)` that moves a piece on the board (handles capture into hands, promotion application).
- May also handle special moves like drops: perhaps a function `drop_piece(game, piece_type, to)` that places a piece from hand to board and decrements hand count.
- Ensures move legality (perhaps integrating with rules_logic for final validation).
- `shogi_game_io.py` – **Input/Output and observation**:
- `generate_neural_network_observation(game)` – creates the 46-channel 9x9 NumPy array representing the game state for the neural net [25] [96]. This includes separate planes for each piece type for current player and opponent, plus hand counts and meta info (current player, move count, etc.). This function is used to feed the neural network.
- `format_move(move)` or similar – possibly to convert a MoveTuple into human-readable notation (e.g., USI, KIF formats). The presence of regex and KIF headers in this module [97] suggests it can parse or produce traditional Shogi notation for moves/games.
- In tests, there's mention of SFEN (Shogi Forsyth-Edwards Notation) parsing; likely functions to convert game state to SFEN string and vice versa might exist or be planned.
- Logging or saving games: maybe functions to output move sequences in KIF format.
- Other contents: The `shogi` package might also contain `test_positions.py` or data files, but not indicated in tree. Possibly all game-related logic is in these four files + core_definitions.

- `keisei/neural_network.py` – *Neural network model definition*:

- `ActorCritic(nn.Module)` – A PyTorch model class with two heads:
- Layers: Currently 1 convolutional layer (`Conv2d(in_channels=46, out_channels=16, kernel_size=3, padding=1)`), ReLU, Flatten, then two Linear layers for policy and value [53].
- `forward(x)` – returns `(policy_logits, value)` given an input state tensor [98].
- `get_action_and_value(obs, legal_mask=None, deterministic=False)` – Convenience method that applies softmax to logits (with legal move masking if provided) and samples or argmax an action [99] [100]. It returns the chosen action index (or actual Move if mapping is done here), the log probability, and the value. Notably, it masks out illegal moves by setting their logits to -inf [101] [102] and handles the edge-case of all moves being illegal (if that were to happen, it leaves it to upstream to handle as we saw) [103].
- *No other classes* in this file. It's minimal by design. In the future, might include additional network classes or architectures.

- `keisei/ppo_agent.py` – *PPO reinforcement learning agent*:
    - `PPOAgent` class – Manages the policy and value network plus the PPO update algorithm:
    - Attributes:
        - `policy_output_mapper` (PolicyOutputMapper) passed in on init, to map between moves and network outputs [104].
        - `num_actions_total` – the size of action space, set by querying the mapper [48].
        - `model` – an `ActorCritic` network instance created on init (on given device) [48].
        - `optimizer` – Adam optimizer for the model parameters [105].
        - Hyperparams: `gamma`, `clip_epsilon`, `ppo_epochs`, `minibatch_size`, `value_loss_coeff`, `entropy_coef` are stored from init arguments [106] [107].
        - (Optionally `name` attribute, used for logging or identifying agent, default "PPOAgent" [108] [109].)
    - Methods:
        - `select_action(obs: np.ndarray, legal_shogi_moves: List[MoveTuple], legal_mask: torch.Tensor, is_training: bool) -> (MoveTuple|None, int, float, float)` – Given an observation and the list of legal moves + mask, returns:
        - `selected_move` (MoveTuple or None if no move available),
        - `policy_index` (int index of move in flattened action space),
        - `log_prob` of that action,
        - `value` of the state. Internally: it sets the model to train/eval mode based on `is_training` [110], converts obs to torch tensor, and calls `self.model.get_action_and_value` (with legal_mask). If `legal_mask` is all False (no moves), it prints a warning [111]. If the selected move comes back None (which would only happen in an error case), upstream code resets the episode as seen in train.py [78].
        - `learn(experience_buffer: ExperienceBuffer) -> dict` – Performs the PPO update:
        - Retrieves all data from the buffer (states, actions, advantages, returns, log_probs, etc.), then for a number of epochs (`ppo_epochs`) iterates over shuffled minibatches.
        - For each minibatch, computes new log_probs and values from current policy, calculates policy loss (with clipping by `clip_epsilon`), value loss, entropy bonus.
        - Takes an optimizer step. Tracks KL divergence (to decide if early stopping needed, though not sure if implemented) and possibly adjusts learning rate.
        - The method likely returns a dictionary of metrics (policy_loss, value_loss, entropy, KL, learning_rate) for logging [112].
        - Tests assert that this metrics dict is returned and contains floats [113].
        - `save_model(path: str)` – (Not explicitly shown above, but implied) Saves the model weights (and maybe optimizer state) to given file. Indeed, `test_train_resume_autodetect` calls `agent.save_model(path)` to create a fake checkpoint [114]. The implementation probably uses `torch.save()` on a state dict that includes at least the model state. (It might not save optimizer to keep it simple, meaning resume will load network but reset optimizer – acceptable trade-off).

        - `load_model(path: str)` – Loads model (and possibly optimizer) from file. We saw in tests a `MockPPOAgent.load_model()` returns empty dict as placeholder [115], suggesting real `PPOAgent.load_model` likely returns a dict of loaded stuff (or nothing). Possibly it loads model weights via `torch.load()` into `self.model` and returns some info.
        - `get_value(obs_np: np.ndarray) -> float` – likely returns the critic's value estimate for a given state (used at episode end to compute advantage for last state). The mock agent defines it to return 0.0 [116]. Real one probably does a forward pass with no grad and returns the value.
    - Overall, `PPOAgent` encapsulates both the policy and learning. It does not subclass any framework class and is custom. It implements the PPO update loop internally, meaning it is somewhat self-contained.

- `keisei/experience_buffer.py` – *Experience trajectory storage*:

    - `ExperienceBuffer` class – Holds experiences for PPO:
    - Attributes: configured with `buffer_size` (max timesteps per epoch, e.g. 2048), `gamma`, `lambda_gae` (for GAE), device.
    - Internally stores lists (or tensors) for each element: observations, actions, rewards, dones, log_probs, values, and perhaps legal_masks (the code indeed stores `legal_mask` for each step [117]).
    - `add(obs, action, reward, log_prob, value, done, legal_mask)`: appends experience to buffers, or if full, discards/overwrites oldest. In tests, adding beyond capacity keeps length at capacity and discards oldest entry [60].
    - `__len__`: returns how many entries currently (<= buffer_size).
    - `compute_advantages_and_returns(last_value: float)`: after an episode or epoch finishes, computes GAE advantage for each timestep and the discounted returns. Likely produces tensors for advantages and returns that will be used in PPOAgent.learn. This involves iterating from end of trajectory backwards computing $\delta = r + \gamma * V(next) - V(curr)$ and advantage = discounted sum of deltas (with λ).
    - `get_batches(minibatch_size)` or similar: possibly yields minibatches of the collected experiences for PPO update. Alternatively, `PPOAgent.learn` may directly index into the buffer's arrays if they're tensors.
    - `clear()`: resets the buffer (called after an epoch's PPO update) [73].

- `keisei/utils.py` – *Miscellaneous utilities and base classes*:

    - `PolicyOutputMapper` class – **Move indexing utility**:
    - In `__init__`, it **generates all possible moves** and builds two mappings:
        - `idx_to_move: List[MoveTuple]` – index -> move.
        - `move_to_idx: Dict[MoveTuple, int]` – move -> index.
    - It generates:
        - All board moves (from every square to every other square, with and without promotion flag) [46] [118], skipping moves that start and end on the same square. (This results in 9*9 * 9*9 - 81 (null moves) = 6561 positions * 2 (promo flag) = 13,122 possible board moves.)
        - All drop moves (for each piece type that can drop, each board square) [119] [120]. If 7 piece types can drop and 81 squares, that's 567 drop moves. Total comes to 13,689 by that count; however, evaluate.py suggests 13,527 [121], so perhaps some moves are invalid and not included (like dropping pawns on last rank may not be allowed?

But the mapper likely includes even those because legality is handled later. The discrepancy might be due to not including King drops as King is never in hand).
        - `get_total_actions()` – returns the length of `idx_to_move` (size of action space) [122].
    - `get_legal_mask(legal_moves: List[MoveTuple], device)` – returns a boolean tensor of length = total actions, where indices corresponding to `legal_moves` are True, others False [55]. This is used to mask out illegal moves for the neural net.
    - `shogi_move_to_policy_index(move: MoveTuple) -> int` – looks up a move's index. If move not found exactly (due to maybe PieceType identity differences for drops), contains a heuristic to match drop moves by value rather than enum identity [123] [124]. If still not found, raises error. This is used likely when converting the move chosen by the agent back into a MoveTuple for the environment.
    - (Potentially a `policy_index_to_shogi_move(idx)` could be implemented via `idx_to_move[idx]`).
    - This class is central to connecting the NN outputs to game moves.
    - `BaseOpponent` class (abstract) – An interface for opponent behavior:
    - Method `select_move(game: ShogiGame) -> MoveTuple`: to be overridden by concrete opponents [42].
    - The PPOAgent itself does not subclass this, but **SimpleRandomOpponent** and **SimpleHeuristicOpponent** in evaluate.py do (they subclass BaseOpponent) [32] [33].
    - This allows treating different opponent strategies uniformly (especially in evaluation).
    - `TrainingLogger` class – **File and console logger for training**:
    - Constructor takes a file path and maybe verbosity flags. It likely opens the file for writing (append).
    - Method `log(iteration, timestep, info_dict)` – writes a formatted line to file and optionally stdout. The format might be JSON or tab-separated. It's used to record episode results and loss metrics periodically.
    - The test mentions an "effective_config.json" and "training_log.txt". Possibly TrainingLogger also handles writing the config to a file at start.
    - There might be a corresponding `EvaluationLogger` with similar structure for evaluation runs (writing match outcomes).
    - Logging classes use context managers or flush on each write to ensure no data loss if crash.
    - Other utilities: possibly some small functions, e.g., for seeding or Rich console setup. Given `train.py` directly imports Rich classes, those might not be here.

- `keisei/evaluate.py` – *Evaluation script*:

    - Responsible for running evaluation games (outside of training loop). Key components inside:
    - Loading a trained agent from checkpoint for evaluation (`load_evaluation_agent`) – creates a PPOAgent, then loads weights [125] [126].
    - Initializing an opponent (`initialize_opponent(type)`) – returns an instance of BaseOpponent subclass (random or heuristic, or loads another PPOAgent if type=="ppo" and path given).
    - `run_evaluation_loop(agent, opponent, num_games)` – runs a series of games between the agent and opponent and collects results:
        - Likely loops for num_games, each time resetting a ShogiGame and playing until game_over. On each turn, it asks one player (agent or opponent) for a move (probably agent as Black and opponent as White, then alternates who is to move).
        - Maintains win/draw/loss counters.

- `execute_full_evaluation_run()` – Higher-level function that ties everything: reads config (how many games, opponent type), calls `initialize_opponent`, loads the agent, runs the loop, logs results (possibly to console and W&B if enabled), and saves any evaluation artifacts.
- Opponent classes (SimpleRandomOpponent, SimpleHeuristicOpponent) are defined here (though logically they could live in utils):
  - `SimpleRandomOpponent` – selects a random legal move from `game.get_legal_moves()` [127].
  - `SimpleHeuristicOpponent` – uses basic heuristics (e.g., prefer capturing moves, avoid certain drops) [128] [129]. For instance, it iterates legal moves and groups them into capturing moves, pawn moves that are non-promoting, and others, then selects from a category (likely captures first if available) [130] [129]. It's simplistic but a bit stronger than random.
- This script also loads `.env` for W&B (similar to train) [131] [132], and presumably initializes W&B logging for the evaluation run if enabled (with a separate project name as per config [133]).

• **Tests** (`tests/` directory):

• Organized by functionality:
- **Game logic tests:** `test_shogi_engine.py`, `test_shogi_core_definitions.py`, `test_shogi_rules_and_validation.py` (covering move generation and rule edge cases like pawn drops [19]), `test_shogi_game_io.py` (likely tests observation generation and notation conversion), `test_shogi_game_rewards.py` (checks reward assignment for win/loss/draw).
- **Mapping tests:** `test_legal_mask_generation.py` (ensures the PolicyOutputMapper correctly flags legal moves and size matches action space).
- **Agent/Buffer tests:** `test_ppo_agent.py` (PPOAgent initialization, select_action output types [134], and learn update correctness with dummy data [112]), `test_experience_buffer.py` (buffer add/clear behavior and length management [59]).
- **Utils tests:** `test_utils.py` (possibly tests PolicyOutputMapper mapping consistency, and logger formatting).
- **Training & Eval tests:** `test_train.py` (CLI interface, resume logic, minimal run with checkpoint creation [26]), `test_evaluate.py` (simulate an eval run using MockPPOAgent to verify tournament loop works and results are logged).
- **Logging test:** `test_logger.py` (verifies TrainingLogger and EvaluationLogger output format and that files are written to).

(The above map provides a structural overview; for brevity not every small function is listed, but major components and their interplay are described.)

B. Test Suite Inventory

The following table summarizes the test files in the project and the aspects they cover, along with an indication of coverage breadth:

Test File	Targeted Module/Feature	Coverage Highlights
`test_shogi_core_definitions.py`	**Core Data Structures** (Color, PieceType, Piece, etc.)	Verifies enum values, Piece initialization and `symbol()` outputs, promotion mappings, etc. Ensures fundamental constants (board size, piece lists) are as expected.
`test_shogi_engine.py`	**ShogiGame Engine (Integration)**	Creates full game scenarios to test turn-turn mechanics. Likely includes checkma detection, repetition draw, and max mo draw enforcement. Ensures `ShogiGame.reset()` correctly sets up initial position and that a sequence of moves leads to expected outcome (like a known mate sequence resulting in game_over and winner).
`test_shogi_rules_and_validation.py`	**Move Legality Rules** (`shogi_rules_logic.py`)	Focuses on specific rule functions: e.g., dropping pawn rules (nifu) [135], drop-checkmate (uchifuzume), general legal move generation. Uses an `empty_game` fixture [136] to simulate board configurations and asserts that `can_drop_specific_piece` returns True/False appropriately [137]. Also tests `generate_all_legal_moves` produc moves consistent with game state (e.g., illegal moves included). Thoroughly cov pawn-drop edge cases and promotion rules.
`test_shogi_game_io.py`	**Observation & Notation I/O** (`shogi_game_io.py`)	Likely tests conversion functions: e.g., creating an observation tensor from a known board position and checking the array has correct bits set (for pieces, ha current player indicator). May also test serialization like SFEN: for a given game state, `to_sfen` and `from_sfen` proc an equivalent state (if those functions exist). This ensures the neural network inputs are correct and that any game record outputs are formatted properly.

Test File	Targeted Module/Feature	Coverage Highlights
`test_shogi_game_rewards.py`	**Rewards and Game End States**	Ensures that the game logic returns appropriate rewards for outcomes. For example, if Black checkmates White, Bla reward = +1, White's = -1 (or similar sche and game_over is True with winner=Bla If draw by repetition or max moves, rew might be 0 for both and winner=None. Confirms that these values align with expectations so the PPO agent is learnir from correct signals.
`test_legal_mask_generation.py`	**PolicyOutputMapper Masking** (`utils.py`)	Validates that given a set of legal moves from a position, the `PolicyOutputMapper.get_legal_ma` correctly marks those indices True and a others False. Likely constructs a known scenario (or uses a small subset of move to test mask length equals total actions that at least those moves are allowed. A might test that if no moves, mask is all False (and that scenario is handled upstream).
`test_utils.py`	**Utility Functions & Mappers**	Possibly tests aspects of `PolicyOutputMapper` mapping. For instance, take a known MoveTuple and ensure `move_to_idx` and `idx_to_move` are consistent inverses. might also test the move heuristic in `shogi_move_to_policy_index` for c moves (ensuring that moves with PieceT from different enum instances still map properly). If `TrainingLogger` or othe utils have complex logic, it could test th (though there is also a separate logger test).
`test_logger.py`	**Training/Evaluation Logger** (`TrainingLogger` , `EvaluationLogger`)	Ensures that log files are created and lir are written correctly. It might feed a dummy info_dict to `TrainingLogger.log()` and then re the file to confirm format (e.g., JSON lin CSV line contains the keys/values). Also tests that `EvaluationLogger.log()` handles input similarly (noting that a previous finding indicated slight structu differences [138]). This confirms that logg doesn't crash and outputs are as expect

Test File	Targeted Module/Feature	Coverage Highlights
`test_ppo_agent.py`	**PPOAgent Behavior** (`ppo_agent.py`)	Tests PPOAgent end-to-end on small sca
- *Initialization & Action:* Create a PPOAgent with a `PolicyOutputMapper` and random input, call `select_action()` and assert the outputs are of correct types and within valid ranges [134] [55] (e.g., index is int within num_actions, log_prob is float, etc.). It simulates a scenario with at least one legal move; if none, it handles skip as seen.		
- *Learning Update:* Constructs a tiny `ExperienceBuffer` with a few steps of dummy data (random observations and some made-up rewards/values) and calls `agent.learn(buffer)` [61] . Asserts that a metrics dict is returned and contains keys like "ppo/policy_loss" with float values [113] . This ensures the backpropagation runs without runtime errors and produces numeric results. It may also verify that after learning, the buffer is not implicitly cleared (since clearing is handled outside in train loop).		
`test_experience_buffer.py`	**ExperienceBuffer Mechanics** (`experience_buffer.py`)	Validates adding experiences and buffe length management: fills the buffer ove capacity and ensures it caps at buffer_si (and likely evicts oldest) [59] . Checks tha after adds, the internal lists (actions, rewards, etc.) have expected contents (e oldest dropped). May also test `compute_advantages_and_returns` perhaps by constructing a simple scena where rewards are known and last_valu and verifying the computed returns ma manual calculation (for instance, if rewa = [1,1,1], gamma=1, advantages should equal returns = [3,2,1]). Since GAE is involved, they might test a scenario whe lambda=1 vs lambda=0 to ensure extrem produce expected results (though this might be beyond what the current tests do).
`test_train.py`	**Training Script Integration** (`train.py`)	Uses subprocess calls to simulate runni the training script:

Test File	Targeted Module/Feature	Coverage Highlights
- `train.py --help` to ensure the argparse help executes without error and contains expected options [139] .		
- Tests resume logic: creates a fake checkpoint file, then runs `train.py --savedir tmp --run_name run --total-timesteps 1` and checks that output indicates resuming from checkpoint [140] [45] . This confirms that if a checkpoint exists in the provided save directory, the script detects it and prints the resume message.		
- Tests a minimal training run: sets `CHECKPOINT_INTERVAL_TIMESTEPS=1` via a JSON config override, runs training for 1 timestep [141] , and then asserts that a checkpoint file was indeed created and a log file exists [27] . This effectively tests the training loop can start and finish in a trivial case and that file outputs occur (without needing to simulate a full long training).		
These are more like **smoke tests** ensuring the script runs as expected with certain flags, and that certain side effects (printing usage, creating files) happen.		
`test_evaluate.py`	**Evaluation Pipeline** (`evaluate.py`)	Tests the evaluation script functions in isolation with mocks:
- It defines a `MockPPOAgent` that inherits PPOAgent and BaseOpponent to simulate an agent that can select moves but uses a dummy model [68] . This allows testing evaluation without needing a trained model.		
- Tests `initialize_opponent` for various types (random, heuristic, PPO) returns correct classes.		
- Tests `run_evaluation_loop` : likely by patching a `ShogiGame` to a small deterministic scenario or by limiting moves. It might simulate a single game and force certain outcomes by controlling the `MockPPOAgent` behavior (the mock selects random moves, so outcome may be random – the test could set a fixed random seed or intercept certain calls to force a particular sequence).		

Test File	Targeted Module/Feature	Coverage Highlights
- Ensures that after running `execute_full_evaluation_run` with, say, 1 game, the reported results (wins/draws/losses) match expectations (perhaps expecting 1 win for one side if the mock agent just makes a move and then game is artificially ended).		
- Checks that evaluation does not crash and that it logs to W&B if enabled (maybe by patching `wandb.init` to a dummy and verifying it's called).		
Overall, it verifies the evaluation pipeline logic, including that it can handle the `BaseOpponent` interface uniformly.		

Coverage Summary: The test suite is very comprehensive, covering virtually all modules. Game logic tests ensure the complex Shogi rules are correctly implemented. RL component tests ensure the training loop and algorithms work in principle. The few gaps are non-critical (e.g., no direct test of `neural_network.py` forward pass, but that's indirectly covered via PPOAgent tests; no explicit test of Sentry integration which is not active anyway). With ~95% of the code exercised by tests (estimated qualitatively), the test suite gives strong confidence in code quality and facilitates safe refactoring.

C. Feature-to-Module Traceability Matrix

This matrix maps the major features or functionalities of the Shogidrl system to the modules that implement them, ensuring clear ownership of each concern:

Feature / Functionality	Implemented in Module(s)	Notes
Full Shogi Rules Enforcement	`keisei/shogi/shogi_game.py`, `shogi_rules_logic.py`, `shogi_move_execution.py`	All game mechanics are handled here: legal move genera [142], move application, check/checkmate detection, repe draw tracking, pawn drop rules [135], etc. The rules logic n contains the specific rule checks, while `ShogiGame` prov the state container and high-level operations (reset, make_move).
Game State Representation	`keisei/shogi/shogi_game.py`, `shogi_game_io.py`	Board and pieces represented via classes in `shogi_core_definitions.py`. `ShogiGame.get_observation()` uses `shogi_game_io.generate_neural_network_observ` to produce a 46x9x9 tensor for the NN [25]. Also, `shogi_game_io` handles conversion to standard notati (SFEN/KIF) for saving or debugging game states.

Feature / Functionality	Implemented in Module(s)	Notes
Neural Network (Policy & Value)	`keisei/neural_network.py` (ActorCritic class)	Defines the architecture and forward pass of the policy/v network [53] . Currently minimal (one conv layer), slated fo expansion. All network weight operations happen here. T of the system treats it as a black box providing policy logi state value.
Policy-Action Mapping	`keisei/utils.py` (PolicyOutputMapper class)	Bridges the gap between the neural network's policy outp and Shogi game moves. Generates the static mapping of to MoveTuples on initialization [46] [47] . Used by PPOAgen filter illegal moves (via masks) and to convert selected ind back to actual moves. This isolates all action space indexi logic in one place.
PPO Algorithm & Agent Behavior	`keisei/ppo_agent.py` (PPOAgent class)	Implements the RL algorithm (Proximal Policy Optimizati action selection with exploration (using the model's probabilities) [110] and network training (`learn()` perfo gradient descent on policy/value loss). It encapsulates all hyperparameters and logic, providing a clean interface (`select_action` , `learn` , `save_model` , etc.) to the t loop.
Experience Collection & GAE	`keisei/experience_buffer.py` (ExperienceBuffer)	Manages storage of trajectory data and computation of **Generalized Advantage Estimates (GAE)** for PPO [61] . Th training loop simply feeds observations, actions, rewards the buffer, and triggers `compute_advantages_and_re` when appropriate; the buffer handles the math of discou and advantage calc. Ensures PPOAgent receives pre-com advantages for learning.
Training Loop Orchestration	`train.py` (to be refactored into `keisei/training/`)	Controls the overall reinforcement learning process. Responsibilities:
- Initializing environment (ShogiGame), agent (PPOAgent), etc.		
- Running the self-play loop: for each timestep, get state, get action from agent, apply to game, store experience [3] [4] .		
- Detecting episode end (win or draw) and resetting game.		

Feature / Functionality	Implemented in Module(s)	Notes
- Every `STEPS_PER_EPOCH` steps, calling PPOAgent.learn on collected data [41] .		
- Logging training progress (to console, file, W&B).		
- Saving model checkpoints periodically.		
It is effectively the “main function” tying together all pieces. Planned to be moved into a `Trainer` class for better structure.		
Configuration Management	`config.py` , `train.py` (argparse), planned `configs/`	Currently via `config.py` constants [35] plus CLI overrid The training script loads these into a `cfg` (SimpleName and updates based on args or JSON config file [66] . A futu feature is to use structured config in `keisei/configs/` Pydantic for clarity [30] . This will centralize all config in on place.
Telemetry & Logging	`keisei/utils.py` (TrainingLogger), `train.py` , `evaluate.py`	Logging of metrics is done through `TrainingLogger.l` calls inside the training loop (e.g., logging episode reward losses every iteration). `train.py` also uses `wandb` for experiment tracking (initialized if WANDB_LOG_TRAIN is T [29]). The `.env` file is used to supply W&B API keys secu Similarly, `EvaluationLogger` is used in evaluate runs t results. Console output is managed via Rich (progress ba formatting of live stats). The plan includes possibly impro logging consistency.
Model Checkpointing	`train.py` (checkpoint logic), `ppo_agent.py` (save/load)	The training script handles when to save (every N timeste episodes) [49] . It uses `PPOAgent.save_model()` to per network (and maybe optimizer). The naming convention “checkpoint_ts{timestep}.pth”. Resume logic on startup fi latest checkpoint in a given directory [51] [143] and loads it `PPOAgent.load_model()` , allowing training to continu the last weights. This feature ensures long runs can be pa or recovered.

Feature / Functionality	Implemented in Module(s)	Notes
Evaluation Matches	`keisei/evaluate.py`, `keisei/utils.py` (BaseOpponent)	Evaluating the agent's strength by playing games outside training loop. The `evaluate.py` script uses the saved m to instantiate a PPOAgent (in eval mode), and pits it again chosen opponent: random, heuristic, or another trained a 144 32 . It runs multiple games and reports statistics (wi draws/losses). This provides a measure of performance a be done periodically during training (the training config h EVAL_DURING_TRAINING flag and will call out to this scrip functions accordingly 89).
Test & Dev Support	`pyproject.toml` (dev dependencies, lint config), `pytest.ini`	The project supports developer productivity with Black, F Isort, PyLint, Mypy configured (mostly via pyproject 12 1 `pytest` is configured to run PyLint on tests as well 14 . ensures code quality. Additionally, numerous tests (as list above) act as a safety net and documentation of expected behavior.

This traceability matrix confirms that for each major capability of the system, there is a clear corresponding implementation area in the code, and vice versa every module serves a distinct purpose in the overall feature set.

D. Architectural Decision Record (ADR) Backlog

(A list of pending or upcoming architectural decisions that the team may need to make, based on observations and plans.)

1. **Config System (Pydantic vs. Hydra/OmegaConf):** The team has leaned towards Pydantic for config management 30 , moving away from plain argparse + constants (and possibly scrapping OmegaConf if it was previously considered 81). The decision essentially made: *Use Pydantic BaseModel for config.* The remaining ADR is designing the schema (flat vs. nested) and how to integrate with CLI. **Decision:** Adopt Pydantic (as planned) with one model for training (possibly containing a nested model for evaluation settings). Implement strict validation (e.g., extra=forbid). This ADR seems resolved in principle, pending implementation.

2. **Module Restructuring & Naming:** The proposal to create `core/`, `training/`, `evaluation/`, `utils/` subpackages 69 is an important architecture change. **Decision:** Proceed with restructuring to improve modularity. Also, consider renaming some files for clarity (e.g., `shogi_game.py` might remain as is, but `evaluate.py` might become just an `evaluation/evaluate.py` script or integrated into an Evaluator class). Ensure imports are updated and backwards compatibility (if any external code uses keisei package) is either maintained or communicated.

3. **Agent-Opponent Design:** There's an implicit decision on whether `PPOAgent` should implement the `BaseOpponent` interface so it can seamlessly be used in evaluation matches. The test implemented a combined subclass for mocking 68 , indicating the architecture wasn't finalized. **Decision Option A:** Make `PPOAgent` subclass `BaseOpponent` (since it already has `select_move` via `select_action` albeit with different signature). **Option B:** Keep them separate but provide an adapter in evaluation (which basically just calls

`agent.select_action` internally). Given that `PPOAgent` already contains all needed to act as an opponent, having it implement `select_move(game)` might be cleanest – it would retrieve the observation and call its own policy to return a MoveTuple. This ADR is about design elegance; likely outcome: implement `select_move` in PPOAgent (or a thin subclass) to conform to BaseOpponent, simplifying evaluation logic.

4. **Neural Network Architecture Choice:** Currently a very simple ConvNet is used. There's an open decision on how to evolve this:

5. Stick with a simple model for now (faster iteration) vs. implement a deeper ResNet as per original vision.
6. If deeper, design of architecture: number of residual blocks, channels, etc., possibly informed by AlphaZero-like networks.

7. Also, whether to incorporate more advanced features (self-attention layers? given 2025 timeline, could consider). This decision will affect training speed and final strength. Possibly postpone until base system is stable. **Likely Decision:** Implement a modest ResNet (maybe 4-6 residual blocks) once basic training is confirmed to work, and make it configurable (so experiments can toggle network size).

8. **Parallel Self-Play vs. Single-thread:** As performance needs grow, a decision will loom on scaling. The code hints at multiprocessing considerations (setting spawn) but doesn't yet implement multi-process self-play. **Decision:** For v1.0, stick to single-process self-play (simpler, enough for baseline). Consider multi-process in future if needed (which might involve significant changes like a manager for game instances and asynchronous experience collection). This ADR is essentially deciding not to complicate things now – and revisiting after core refactor if training is too slow.

9. **Evaluation Frequency & Strategy:** How to integrate evaluation during training is partly decided (they tie it to checkpoint saving frequency [89]). But going forward, one might decide to run a small evaluation match every N timesteps to monitor progress, or use a separate script. **Decision:** The current approach (periodic eval in training process, logging to W&B) seems fine for now. In future, if doing lengthy training, might offload evaluation to a separate process to not stall training – but that's a complexity trade-off. For v1.0, keep it simple (in-process eval is okay given it's relatively quick or infrequent).

10. **Reward Structure and Termination:** If new rules like Nyugyoku are added, need to decide how to handle their reward (likely treat as win for one side). Also if adjusting reward shaping (currently likely +1/-1/0 for win/loss/draw). **Decision:** Keep reward sparse (win=1, loss=-1, draw=0) to adhere to "learn from scratch" philosophy – no change needed. Just ensure any new termination conditions map to appropriate rewards.

11. **Package Release vs. Research Code:** Decide if this is primarily a research project or intended as a reusable package. If releasing on PyPI, might add things like command-line entry points (via console_scripts in pyproject) for `train-keisei` or similar. If it's more internal, that's not needed. **Decision:** Likely prepare it as a package that can be pip-installed and used via CLI or imported (since a pyproject and version exists). So, ensure `__main__.py` or console script for easy launching.

These ADR items outline upcoming choices. Most align with the maintainers' current plan trajectory, showing that the team is on the right path to address them. Documenting these decisions (perhaps in a `docs/ADR.md`) as they are resolved would be beneficial for future contributors.

E. Full Issue Inventory

(A consolidated list of issues identified by this audit or documented in project plans, serving as a to-do list for the team. Each issue is briefly described with any relevant reference.)

1. **Monolithic `train.py` (Refactor Needed):** Break up responsibilities into `Trainer` class and supporting modules. *Status:* Planned (TRAINING_REFACTOR.md) [8]. *Severity:* High – will improve maintainability greatly.

2. **Scattered Configuration Management:** Unify config handling using a structured approach. Remove redundant config fields (e.g., `NUM_ACTIONS_TOTAL`) and use one source of truth. *Status:* Planned (refactor doc) [30]. *Severity:* Medium.

3. **Checkpoint Naming Confusion:** Filenames use `total_timesteps` which can mislead if run is resumed with a new target [50]. Consider including actual steps in filename or a separate ID. *Status:* Identified in internal review [83]. *Severity:* Low.

4. **Checkpoint Loading Robustness:** Improve error handling for corrupted or incompatible checkpoints (currently only FileNotFound is caught) [146]. Perhaps verify model architecture or include version in checkpoint. *Status:* Identified internally. *Severity:* Medium (affects long-run reliability).

5. **Magic Strings and Constants:** Use centralized constants for file patterns and log file names instead of scattered literals [2]. E.g., define `CHECKPOINT_PATTERN` in config or at top of train.py. *Status:* Identified internally. *Severity:* Low.

6. **Logging Consistency:** Align TrainingLogger and EvaluationLogger interfaces (there was an inconsistency noted where EvaluationLogger wasn't updated to match TrainingLogger's changes) [138]. Ensure both loggers can handle similar usage patterns. *Status:* Identified (Test Remediation Plan). *Severity:* Low.

7. **Logger Flexibility:** TrainingLogger's current format is somewhat rigid [147] (expects certain fields). Might consider making it more general or using Python logging module. *Status:* Minor suggestion. *Severity:* Low.

8. **Test Coverage for Config Variations:** Need more tests for using `--config` files (malformed JSON, missing fields) to ensure the program fails gracefully or applies overrides correctly [21]. *Status:* Identified internally. *Severity:* Medium.

9. **Resume Edge Cases:** Test and handle scenarios like resuming from a checkpoint when `TOTAL_TIMESTEPS` in config is **less** than the checkpoint's step count (should the run exit immediately? or continue indefinitely?). Also resuming with an explicit checkpoint path that doesn't exist – currently, if `--checkpoint` param was allowed, how is it handled? Ensure clear messaging. *Status:* Identified internally [148]. *Severity:* Medium.

10. **Test Setup Clarity:** In tests, overriding config by directly setting attributes (like `cfg.CHECKPOINT_INTERVAL_TIMESTEPS`) works, but is a bit implicit [149]. It's fine, but note in contributing docs that if config refactor changes how config is accessed, tests need updating. *Status:* Not a user-facing issue, just a note. *Severity:* Low.

11. **Neural Net Capacity Insufficiency:** Current network likely underfits. Plan to increase complexity (ResNet). No GitHub issue filed, but recognized as improvement point. *Status:* To-do for performance. *Severity:* Medium (for agent strength).

12. **PolicyOutputMapper Size Mismatch (3159 vs 13527):** Outdated constant in config [31]. Fix to avoid confusion. *Status:* Discovered in audit. *Severity:* Low (no runtime impact but potential confusion).

13. **Self-Dependency in Requirements:** `-e git+https://...#egg=keisei` is an odd entry that could cause confusion or installation of wrong version. Should be resolved (maybe remove it, since the package can be installed by path). *Status:* Discovered in audit. *Severity:* Low (dev experience issue).

14. **Sentry SDK Utilization:** Sentry is listed but not used. Decide to integrate properly (e.g., call `sentry_sdk.init()` with DSN if available) or drop it. An unused dependency just adds bloat. *Status:* Discovered in audit. *Severity:* Low.

15. **Nyugyoku Rule Not Implemented:** Mentioned in docs but not in code yet [74]. Add this rule to game logic to complete Shogi rules. *Status:* Known future feature. *Severity:* Low (rare scenario, only matters in very specific endgames).

16. **Evaluation Enhancements:** Possibly allow agent vs agent evaluation (currently one agent vs built-in opponents). If wanting to measure true strength, playing two instances of the trained agent against each other or against older checkpoints would be useful. Not implemented yet (the infrastructure is partly there via BaseOpponent and ability to load a PPO as opponent, but might need polish). *Status:* Future improvement. *Severity:* Low (nice-to-have for research).

17. **Parallelization (if needed):** If training is slow, consider multi-process for self-play. Not urgent for functional correctness, but flagged for future if scaling up (no issue now as 2048 steps PPO is okay on one CPU). *Status:* Future consideration. *Severity:* N/A now.

Each of these items can be tracked (in GitHub issues or project board) as tasks. The top half of the list (1–10) comes from either this audit's findings or the project's own review documents, and addressing them will largely finalize the 1.0 readiness. The latter items (11–17) are more enhancements or forward-looking improvements to increase the agent's performance and the system's completeness.

F. Manual Dependency Review

The project's dependencies (from `requirements.txt` [37] and dev requirements in `pyproject.toml` [12]) are reviewed below for version appropriateness, usage, and any concerns:

- **Python Version:** Requires Python 3.12+ [150]. This is a very modern requirement (Python 3.12 released Oct 2023). It's fine, but note that some users may still be on 3.10/3.11. If there's no 3.12-specific feature in use, consider allowing 3.10+ to broaden the user base. However, if using 3.12 (perhaps for performance improvements or just to ensure latest), document it clearly.

- **Core Libraries:**

- **torch==2.7.0** – PyTorch, used for neural network and tensor operations. Version 2.7.0 is futuristic (PyTorch 2.0 came in 2023; 2.7 by mid-2025 seems plausible). No issues; PyTorch is well-maintained. Just ensure compatibility of model save files with this version if users use slightly older Torch.
- **numpy==2.2.6** – Numpy major version 2 indicates a new release series (Numpy 1.x was the standard for decades; assume 2.x is available by 2025). Using latest is good. Ensure the code doesn't rely on deprecated 1.x behavior (looks fine).
- **networkx==3.4.2** – NetworkX is for graph algorithms. **Not obviously used anywhere in code.** Possibly left from an idea to use graphs for something (maybe board connectivity?). If unused, remove to avoid unnecessary install time.
- **sympy==1.14.0** – Symbolic math library. Also not used in code as far as audit can tell. Maybe considered for something like solving something analytically or generating formulas, but likely vestigial. Recommend removing unless a specific use exists.
- **requests==2.32.3** – HTTP library. Not directly used by project code, but wandb and others use it internally. Pinning it explicitly is okay, but wandb already brings its own pinned version. Might remove explicit pin to avoid conflict (if wandb expects a different version) – however 2.32.3 is current so it's fine. No security concerns, since it's up-to-date.
- **python-dotenv==1.1.0** – For loading .env files (used to load W&B API key). Up-to-date and simple. Good.
- **pydantic==2.11.4** – Data validation library. Not actually utilized in code yet (no Pydantic models in use). Listed likely in anticipation of config refactor. Version 2.11 is very new, presumably fine. If not using it yet, including it doesn't harm, but keep only if the plan is to use it soon (which it is).
- **wandb==0.19.11** – Weights & Biases for experiment tracking. 0.19.11 is a 2023 version. By 2025, wandb might be >1.x (they were in 0.xx for a long time, but hypothetically might hit 1.0). Check if there's any known issues: W&B 0.12–0.13 had some API changes, but 0.19 is fairly stable. Possibly update to 0.20+ to get latest fixes (like improved resilience offline, etc.). Also, ensure wandb is an optional dependency – if a user doesn't have a W&B account, does training run? The code tries to init wandb if WANDB_LOG_TRAIN=True [29]. Maybe mention in docs that one can set that False to avoid needing wandb login. Alternatively, catch wandb import errors or failure gracefully.
- **GitPython==3.1.44, sentry-sdk==2.28.0, MarkupSafe==3.0.2, Jinja2==3.1.6** – These are indirect dependencies likely via wandb or pydantic:
    - GitPython: wandb uses it to log the git commit of repo if available.
    - Sentry-SDK: included maybe for error tracking if desired.
    - Jinja2 & MarkupSafe: possibly pulled in by pydantic (if it uses Jinja for schema generation?) or by wandb's reporting (though not sure).
    - They seem fine version-wise. Sentry 2.x is current.
    - No action needed unless one wants to remove Sentry entirely.

- **tqdm** – (It's listed in requirements.txt end implicitly since a blank line, maybe they forgot version pin for tqdm which wandb uses? Actually the snippet shows just "tqdm" with no version [151]. Should pin it for consistency, e.g., tqdm>=4.40. But minor.)

- **Dev/Test Libraries:**

- **pytest, pytest-cov, pytest-pylint** – for running tests and coverage. Up-to-date versions pinned in pyproject [12] [13].
- **pylint, flake8, black, isort, mypy** – all present with specific versions pinned [152] [153]. Good to have consistent formatting and linting.

- **mypy-related pins** (mypy_extensions, typing_inspection, etc.) – ensures type checker works smoothly.
- **deptry** – a tool to find unused dependencies. It's added (likely it would flag networkx/sympy as unused indeed). This shows maintainers are aware of cleaning deps; perhaps they left networkx/sympy intentionally until confirming unused.

- All dev dependencies are fine and modern.

- **Potential Outdated or Problematic Packages:**

- *None of significance.* All packages are either up-to-date or naturally do not pose issues. Wandb's 0.19 could be updated; not critical but new versions have nice features (e.g., better integration with Jupyter, etc.).
- Torch 2.7 is presumably stable; if using GPU, ensure CUDA toolkit matching version is installed, but that's user's responsibility typically.
- Pydantic 2.x changed API significantly from 1.x. If any code was written with Pydantic 1 (not likely here since not used yet), it would need updating. Starting with 2.x fresh is good.

Licensing: All listed packages are permissively licensed (PyTorch - BSD, Numpy - BSD, etc., wandb - MIT). No conflicts.

Security: No known CVEs for these specific versions at time of writing. Always good to run `pip audit` as part of CI to catch any new vulnerabilities in dependencies.

Conclusion: After removing unused entries and updating a couple of version pins, the dependency list will be lean and secure. It's advisable to periodically update the pins (especially for wandb and dev tools) to incorporate bug fixes, but pins ensure reproducibility which is great for a research codebase.

G. Complexity Hotspots

The audit identified a few areas in the code that are particularly complex (either due to length, logic density, or potential performance cost). These "hotspots" deserve special attention:

- **ShogiGame & Rules (Complex Domain Logic):** The Shogi game implementation is inherently complex – handling all movement rules (including underpromotion possibilities, unique moves like dropping pieces, etc.) means the code in `shogi_game.py` + `shogi_rules_logic.py` is long and intricate. At ~1000+ lines [154], `shogi_game.py` is a hotspot. However, this complexity is essential and largely handled via logical decomposition (using helper modules). The risk here is subtle bugs in rare rules (like repetition or stalemate conditions). The tests mitigate a lot of this, but future modifications (like adding a rule or optimizing move gen) should be done very carefully. One might consider splitting some logic into even more modules or classes (for example, having a `RulesEnforcer` class) but that could also add indirection. As it stands, it's manageable but remains a hotspot to monitor with tests on any change.

- **Training Loop (Lengthy Method):** `train.py` (specifically the main while loop in `if __name__=="__main__"`) is a hotspot due to sheer length and the fact it does many things. It's both complex and critical (any mistake here impacts the entire training). This report already recommends refactoring it (Deep Dive 1). Breaking it down will remove it from "hotspot" status by distributing complexity into multiple smaller methods. Until then, any edits to this file must be done with caution and thorough testing.

- **PolicyOutputMapper (Large loops, Memory):** The `PolicyOutputMapper.__init__` is potentially heavy as it generates ~13k move entries with nested loops [46] [47]. Complexity-wise, it's O(9^4) which is 6561 iterations for no-promo moves times 2 for promo (13122) plus drop moves (567) – roughly 13.7k iterations, which is fine. Memory-wise, storing 13k MoveTuples and a dict of same size is not an issue (a few hundred KB). So this is a hotspot only in concept, not in actual cost – the approach is acceptable given the action space needs. The complexity is more about correctness: ensuring that the mapper includes all possible moves and no duplicates. If any mapping bug exists, it could seriously degrade the agent (e.g., missing a move means the agent can never choose it). Tests indirectly check this via legal mask usage. This class is also complicated by the need to reconcile enum identity for drop moves [123]. So while not a performance bottleneck, it's a logical complexity hotspot where a subtle bug could hide. It's well-contained though; any changes (like if Shogi extended to different board size or rules) would need adjusting this mapping logic accordingly.

- **Rich TUI and Logging Integration:** The training script's integration of the Rich library for live progress display is another minor complexity. Handling the Live console, Progress bars, updating them inside the loop, etc., adds complexity that is tangential to core training logic. If something were to go wrong with the Rich update (like a mis-rendered table or an exception in the console), it could break training UI. This is not likely if left as is, but if someone modifies the UI code (perhaps to display additional stats), they must understand the Rich library. It's a "hotspot" in the sense that it's an auxiliary complexity (UI in a non-UI program). If troubleshooting performance or crashes, consider that the Rich loop runs at `refresh_per_second` (configurable, default maybe 10) – negligible overhead, but just to note. This complexity is acceptable for the sake of user experience.

- **Evaluation Loop (Multiple Agents Interaction):** The evaluation routine is slightly complex because it juggles two players (agent and opponent) possibly of different types (AI vs AI, or AI vs random). Ensuring the turn-taking logic and game resets are correct requires careful handling. If an evaluation game ended and the code doesn't reset state properly, it could skew results. The complexity is not high, but it's a place where a logical error (like forgetting to switch current_player or a bug in opponent move selection leading to exception) could stop the evaluation. Tests with MockPPOAgent have helped shake out obvious bugs. As more opponent types or conditions are added, keep evaluation logic straightforward (the Evaluator class may encapsulate this better, reducing complexity in the script).

- **Global State & Cross-Module Interactions:** A different kind of complexity is tracking global state like `wandb` context or config usage across modules. E.g., `train.py` sets `cfg.DEVICE` based on args and uses it throughout [80]; if any module reads the config at import time (none do currently, they use it at runtime), that could be problematic. Fortunately, there's no evidence of misused global state beyond config. After config refactor, passing around a config object explicitly will actually reduce this implicit coupling, which is good.

In summary, the hotspots are under control through testing and planning, but they highlight where future maintainers should focus code reviews and additional tests when changes are made. Documenting these in comments (e.g., "# complex logic – carefully verify if modifying" at top of train.py or shogi_game.py) can also be helpful.

H. Execution Flow Diagrams

To better illustrate how the system executes, below are step-by-step flow descriptions of two core processes: **Training Loop Execution** and **Evaluation Match Execution**.

Training Loop Flow (Self-Play with PPO):

1. **Initialization (Main Script Start):**
2. The user launches `train.py` (optionally with CLI args like `--total-timesteps`).
3. Config is loaded: default constants from `config.py` are put into a `cfg` object, then overridden by any CLI arguments (e.g., device "cuda", seed, etc.) [80]. Random seeds are set for NumPy, Torch, and Python RNG for reproducibility [87].
4. Logging is set up: a `TrainingLogger` opens `logs/training_log.txt` and W&B is initialized if enabled (this reads `.env` for API key and creates a new run on the W&B dashboard) [29].
5. Environment and agent are created: `game = ShogiGame()` (initial board setup) and `agent = PPOAgent(input_channels=46, mapper=PolicyOutputMapper, other hyperparams from cfg)`. The PPOAgent internally creates its `ActorCritic` neural net and an Adam optimizer [48], and sets up for training on the specified device.
6. An `experience_buffer` is created (size = STEPS_PER_EPOCH, gamma, lambda per cfg) to collect experiences.

7. If resuming training: the code searches for latest checkpoint in `models/` directory [51]. If found, it loads the model weights into the agent and possibly adjusts internal counters. A log message notes resumption [20].

8. **Episode Loop (Self-Play):**

9. `global_timestep` is the master counter of environment steps, initialized to 0 (or to the checkpoint's last step if resuming).

10. **While** `global_timestep < TOTAL_TIMESTEPS`:

    - The environment might need resetting if a new episode should start. In code, they likely manage an `episode_step` or check `game.game_over`. Typically, before entering the main loop or within it, if `game.game_over` or `episode_length` reached `MAX_MOVES_PER_GAME`, they call `game.reset()`, increment an episode counter, and maybe log episode end stats. (From OPS_PLAN: they shifted to timestep-based loop rather than episode loop [155], which implies episodes run continuously but with checks.)
    - **Get State:** The current board state is fetched as `obs_np = game.get_observation()` (a NumPy array shape 46x9x9).
    - **Legal Moves:** `legal_shogi_moves = game.get_legal_moves()` returns a list of all moves the current player can make [3]. If empty (should only happen if game is over), they log a warning [156] and create `legal_mask_tensor` as all False (no legal moves) [157]. Otherwise, they compute a legal mask via `policy_output_mapper.get_legal_mask(legal_shogi_moves)` (a boolean tensor marking allowed moves) [157].
    - **Agent Action:** The agent selects an action: `selected_move, policy_index, log_prob, value = agent.select_action(obs_np, legal_shogi_moves, legal_mask_tensor, is_training=True)` [4]. Under the hood, the `ActorCritic` produces policy logits and value, the legal_mask filters illegal ones, and then either a random sample (training mode) or argmax is taken. This yields a move choice and associated log probability and value estimate.
    - If for some reason no move is selected (which could happen if legal_mask was all False and the model couldn't handle it), the code catches `selected_move is None` [78]. In

such a case, they log a critical error, reset the game (to avoid stalling), and continue to next loop iteration (effectively skipping this timestep) [57].

- **Apply Action:** Assuming a valid move:
- `move_result = game.make_move(selected_move)` is called [158]. The game updates its state accordingly (moving piece, capturing if applicable, toggling current_player, etc.). It returns something like `(next_obs, reward, done, info)`:
  - `reward` is the immediate reward for the action (likely 0 for most moves, non-zero only on terminal states win/loss).
  - `done` is True if the game ended as a result of the move (checkmate or draw).
  - `info` might contain metadata (like termination reason).
- The code unpacks `move_result` into e.g. `next_obs_np, reward, done, info` [158] and might do some extra logging for demo mode (slowing down or printing the piece moved using `piece_info_for_demo`) [159] [160].
- **Store Experience:** The transition is added to the buffer: `experience_buffer.add(obs_tensor, policy_index, reward, log_prob, value, done, legal_mask)` (note: they store the state's *tensor* or numpy? Possibly tensor to preserve grad or numpy to save memory; tests show they pass tensor [161]). This records everything needed for learning later, including the legal_mask (which is used for advantage calc if needed).
- **Increment Counters:** `global_timestep += 1`; also increment an episode step counter if tracking.
- **Check End of Episode:** If `done` is True (game over):
- Determine winner from `game.winner` and maybe set reward for both players accordingly (though in self-play, they might be just training one perspective; often in symmetric games, you train from one player's perspective per episode).
- Log episode outcome (win/draw/loss and episode length) to console/file via TrainingLogger.
- Reset environment: `game.reset()`, possibly switch who plays as which side for next episode (not sure if they always start as Black or alternate to balance learning – not mentioned, but alternating might be wise).
- Reset episode-specific counters (episode_reward, episode_length).
- Continue loop (the new state after reset will be processed in next iteration).
- **Check Update Time:** If `global_timestep % STEPS_PER_EPOCH == 0` (i.e., collected one epoch of experience):
- Compute `last_value = 0` if episode done at end of epoch, or if not done, get the value estimate for the current state: `last_value = agent.get_value(current_obs_np)` to estimate value of the not-yet-finished episode state [61].
- Call `experience_buffer.compute_advantages_and_returns(last_value)` to process all collected steps. This populates advantage and return fields in the buffer for use in learning.
- Call `metrics = agent.learn(experience_buffer)`. This runs PPO epochs and updates network parameters [112]. It returns average losses/metrics.
- Log the PPO metrics (policy loss, value loss, etc.) via TrainingLogger and optionally to W&B.
- Clear the experience buffer for the next epoch of collection [73].
- (Possibly update any learning rate schedulers or other training parameters if used – not currently, but maybe planned.)
- **Check Checkpointing:** If `global_timestep` hit a checkpoint interval (say every 50k steps):

- Save model: construct filename `checkpoint_ts{global_timestep}.pth` and call `agent.save_model(path)` [49].
- Also save an "effective_config.json" with current config (so the exact parameters of this run are recorded) and maybe other run metadata.
- If `EVAL_DURING_TRAINING` is True, trigger an evaluation: e.g., call a function `execute_full_evaluation_run` (likely imported from evaluate.py) which will run a set of games with the current model against the specified opponent (random/heuristic/another model) [29] [77]. That evaluation might log its own results (like win rates) to console and W&B. Training loop typically waits for it to finish (since it's probably not threaded).
- The loop then continues collecting the next batch of experiences.

11. **Termination:** Once `global_timestep` reaches `TOTAL_TIMESTEPS`, exit loop. The script will then:

12. Save a final model checkpoint ("checkpoint_ts{TOTAL_TIMESTEPS}.pth").
13. Perform a final evaluation (some configs might do an eval at end).
14. Ensure W&B run is closed properly: `wandb.finish()` to upload final logs [29].
15. Print a completion message (and maybe summary of training).
16. Exit cleanly. Any open file handles (log file) are closed (the logger might close on `__del__` or context manager).

Throughout the training flow, Rich's live progress might be updating a progress bar or table at each step or episode, providing visual feedback in the console. Key metrics like current episode reward, moving average of reward, steps/sec, etc., might be shown (depending on how they set up the Live display).

This flow ensures a cycle of **self-play -> accumulate experiences -> PPO update** repeats until the budget of interactions is exhausted, at which point we have a trained model saved.

Evaluation Loop Flow (One Agent vs Opponent):

1. **Initialization:** The evaluation is typically triggered either by running `python keisei/evaluate.py --checkpoint path --opponent heuristic --games 10` or by the training process calling an evaluation function.
2. The evaluate script loads config (it might reuse `config.py` or accept separate args for eval settings like number of games, opponent type).
3. It calls `load_dotenv()` to get any env variables (like W&B credentials if logging eval).
4. If using W&B for eval, it calls `wandb.init()` with a separate project name (e.g. "shogi-drl-periodic-evaluation") [133], so that eval metrics are logged distinctly.
5. **Agent Setup:** It creates a PPOAgent with the same architecture as training (ensuring to use same input_channels and policy_output_mapper configuration) and then loads the specified model checkpoint weights into it [162]. The agent is set to evaluation mode (could set `is_training=False` flag when selecting moves to disable exploration).

6. **Opponent Setup:** Based on the chosen `--opponent`:

   - "random" -> instantiate `SimpleRandomOpponent` [32].
   - "heuristic" -> instantiate `SimpleHeuristicOpponent` [33].
   - "ppo" -> here likely you'd provide another checkpoint or use the same agent as opponent for self-play evaluation. The code possibly allows specifying `--opp_checkpoint` to load

a second agent. If not, maybe "ppo" means self-play (agent vs itself), but the BaseOpponent interface might not handle that directly (the agent could play both sides by alternating, which effectively is self-play).
    - The result is an object `opponent` that has a `.select_move(game)` method.

7. **Game Loop (for each game out of N):**

8. For game_index in 1...N_games:

    - Create a new `ShogiGame` instance for this match and call `reset()` to ensure starting position.
    - Initialize any game-specific trackers (move count, etc.). Possibly randomize who starts as Black/White if evaluating symmetric agents.
    - Set `game.current_player = Color.BLACK` to start with Black always (by Shogi convention, Black moves first).
    - While not `game.game_over`:
    - If `game.current_player` is the trained agent (say we consider agent as Black always):
        - Obtain state observation (9x9x46) via `game.get_observation()`.
        - Agent needs legal moves: call `legal_moves = game.get_legal_moves()`.
        - Compute mask = mapper.get_legal_mask(legal_moves).
        - `agent.select_action(obs, legal_moves, mask, is_training=False)` to get a move (here we use deterministic= True possibly to pick highest probability if we want consistent eval, or False if stochastic evaluation is desired). This returns a MoveTuple.
    - If `game.current_player` is the opponent:
        - Call `move = opponent.select_move(game)`. For Random it picks random from `game.get_legal_moves()`, for Heuristic it uses its logic to choose.
        - If opponent is actually another PPO agent (in case of agent vs agent), one could either have two agent instances or just use the same `agent.select_action` but this time treating it as the other side (if symmetric). The test uses a MockPPOAgent for opponent which inherits BaseOpponent and PPOAgent, meaning they treated the agent as BaseOpponent as well.
    - Once a `move` is obtained from whichever side:
        - `game.make_move(move)` to apply it. This updates `game.current_player` to the other player, updates board, etc.
        - If `game.game_over` becomes True after this move (someone won or draw):
        - Break the loop.
        - Determine result: e.g., if `game.winner == Color.BLACK`, then agent (Black) won; if White, agent lost (assuming agent was Black); if None/Draw (maybe represented as `winner=None` or `2` for draw as design doc suggested), count as draw.
    - If game not over, loop continues with next turn (which will now be the other side).
    - After exiting the while (game ended):
    - Record outcome: increment agent_wins, agent_losses, or draws counters accordingly.
    - If using an EvaluationLogger or printing results per game, output something like "Game i: Black win" or final move count, etc., for insight.
    - Possibly store some moves or states if needed for debugging (but likely not, to keep eval fast).
    - Move to next game iteration.

9. **After All Games:**

10. Calculate statistics: win rate = wins/N, etc.
11. Print or log summary: e.g., "Agent vs Heuristic: 7 wins, 2 draws, 1 loss".
12. If W&B logging is on, log these stats as an evaluation metric (so it can show up on training dashboard).
13. `wandb.finish()` the eval run (if it was separate).
14. Return or save results as needed (the training loop, if it called this function, might use the returned stats to log in training log as well).
15. **Special Cases:** If the evaluation is agent vs itself (both sides use the same policy), the code could either:
16. Alternate the agent for both players: basically a self-play match with a fixed policy (no learning). This is symmetric and expected outcome should be ~50% if identical agents. It's more to test if agent overfits to playing as one color.
17. Or evaluate new agent vs an older checkpoint (to see improvement over time). The current code allows `EVAL_OPPONENT_TYPE="ppo"` with `EVAL_OPPONENT_CHECKPOINT_PATH` [163], which implies you can specify another model file as the opponent. The evaluate script would then load a second PPOAgent for the opponent from that path. In such case, the loop logic would treat each turn: if current_player is "agent", use primary agent; if "opponent", use opponent_agent's select_action. That would effectively simulate a duel between two different models.

The evaluation flow is simpler than training (no learning or advantage calculation, just playing out games), but it must correctly handle all game end conditions and switching between two AI decision-makers.

These flows demonstrate how the components interact over time. The training flow ensures continuous learning, and the evaluation flow provides measurement of that learning. Both are well-supported by the current architecture, though the training flow will become cleaner after refactor, the conceptual steps remain as described.

I. Proposed CI Configuration (GitHub Actions YAML)

Below is a proposed GitHub Actions workflow for Continuous Integration, which runs tests and linters on each push/PR. It uses an Ubuntu runner with Python 3.12 (the project's minimum) and installs both regular and dev requirements. This will help automate quality checks:

```
name: CI

on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]

jobs:
  build-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [ "3.12" ]  # Use Python 3.12 (project requires
```

```
>=3.12)
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          # Install project in editable mode with dev dependencies
          pip install .[dev]
      - name: Lint with Flake8
        run: flake8 .
      - name: Lint with Pylint
        run: |
          # Only lint the source package (and optionally tests)
          pylint keisei tests

      - name: Check code formatting (Black and isort)
        run: |
          black --check .
          isort --check-only .
      - name: Type-check with mypy
        run: mypy keisei

      - name: Run test suite with PyTest
        env:
          PYTHONPATH: .  # ensure local package is discoverable
        run: |
          pytest --cov=keisei --cov-report=xml

      - name: Upload coverage report
        if: always()  # run even if tests fail, to record coverage
        uses: actions/upload-artifact@v3
        with:
          name: coverage-xml
          path: coverage.xml

  # (Optional) Add a job to publish to PyPI on tags, or separate jobs for
different OS if needed.
```

Notes on this CI configuration: - It triggers on pushes and pull requests to main (adjust branch names as appropriate). - It uses the project's dev dependencies (the `[dev]` extra in pyproject) to install everything needed for tests and linters in one go [12]. - It runs Flake8 and Pylint for complementary lint coverage (PyLint is also run in tests via pytest-pylint plugin, but running separately gives immediate feedback and can cover all files including non-test code). - Black and isort ensure code style consistency. These will fail the build if formatting is off – developers can run these locally to fix. - Mypy to enforce

type correctness (ensuring the types we expect hold up, important after refactors). - Pytest runs with coverage; it generates a coverage XML which is uploaded as an artifact. (For now just stored, but can integrate with Coveralls or Codecov if desired to track coverage over time.) - The PYTHONPATH is set to `.` so that the local `keisei` package can be imported if `pip install .` didn't occur (though we did pip install, so this might not be needed). - This pipeline ensures that any code changes pass all tests and style checks before merging – catching the issues noted in our audit (like unused imports or mismatched types) automatically.

By adopting this CI, the project will maintain its high code quality and avoid regressions, especially during the major refactoring and beyond.

---

1 5 6 15 22 29 40 41 74 77 155 OPS_PLAN.md
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/OPS_PLAN.md

2 21 50 81 83 138 146 147 148 149 TRAINING_REMEDIATION_PLAN.md
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/docs/TRAINING_REMEDIATION_PLAN.md

3 4 7 49 51 52 56 57 67 73 76 78 80 82 87 143 156 157 158 159 160 train.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/train.py

8 9 23 24 30 65 69 TRAINING_REFACTOR.md
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/docs/TRAINING_REFACTOR.md

10 38 README.md
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/README.md

11 39 HOW_TO_USE.md
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/HOW_TO_USE.md

12 13 14 36 84 85 145 150 152 153 pyproject.toml
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/pyproject.toml

16 90 92 DESIGN.md
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/docs/DESIGN.md

17 18 25 96 97 shogi_game_io.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/shogi/shogi_game_io.py

19 93 94 95 135 136 137 142 test_shogi_rules_and_validation.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/tests/test_shogi_rules_and_validation.py

20 26 27 28 44 45 64 66 70 114 139 140 141 test_train.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/tests/test_train.py

31 34 35 75 79 86 88 89 133 163 config.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/config.py

32 33 121 127 128 129 130 131 132 144 evaluate.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/evaluate.py

37 43 151 requirements.txt
https://github.com/tachyon-beep/shogidrl/blob/e9996e2c95b0a71cf93a75facb0b3bf463439ce5/requirements.txt

42 46 47 58 118 119 120 122 123 124 utils.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/utils.py

48 104 105 106 107 108 109 110 111 ppo_agent.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/ppo_agent.py

53 98 99 100 101 102 103 neural_network.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/neural_network.py

54 55 61 62 63 112 113 117 134 test_ppo_agent.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/tests/test_ppo_agent.py

59 60 161 test_experience_buffer.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/tests/test_experience_buffer.py

68 115 116 125 126 162 test_evaluate.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/tests/test_evaluate.py

71 72 91 154 shogi_game.py
https://github.com/tachyon-beep/shogidrl/blob/60aa1cdb0e9e2980f39f13f93c59518dde9a1ca5/keisei/shogi/shogi_game.py