ChatGPT

Deep Static Analysis Audit of the *Shogidrl* Project

Executive Summary

Overall Project Health Scorecard:
- **Code Quality & Correctness: 9/10** – The codebase is well-structured, following modern Python best practices with type hints and comprehensive test coverage. Only minor code smells (e.g. overly broad exception handling and a few overly complex modules) were found [1] [2]. No obvious logic bugs or security flaws were detected in static review.
- **Architecture & Maintainability: 8/10** – The system's manager-based architecture is highly modular and extensible [3] [4]. This promotes separation of concerns and ease of maintenance. However, the proliferation of nine manager components adds some complexity, and one core class (*ShogiGame*) is very large (over 1000 LOC [2]), slightly impacting maintainability.
- **Developer Experience (DX): 9/10** – Extensive documentation (design docs, code map, CI guides) and a logical project structure greatly aid onboarding [3] [5]. Clear naming and consistent coding style (likely due to AI generation) mean new contributors can quickly understand the system. Minor DX drawbacks include the steep learning curve of the custom architecture and the need to grasp many interconnected components.
- **Clarity & Readability: 9/10** – The code is very well-commented and follows a consistent, clean style. Functions and classes have descriptive docstrings, and "magic numbers" are centralized in `constants.py` [6]. The only readability concerns are in the most complex modules (e.g. the Shogi engine logic) where lengthy functions could be further broken down for clarity.
- **Qualitative Test Sufficiency: High** – The test suite is comprehensive, covering core logic, game rules (drops, promotions, special conditions), training loop behavior, and even integration aspects. For example, tests cover nuanced Shogi rules like **Nifu** (double-pawn drop) and **Uchi-fu** (drop pawn mate) conditions [7] [8]. This high coverage gives confidence in correctness and guards against regressions.

Key Strengths: The project exhibits several exemplary attributes based on the manual audit: - **Robust Modular Architecture:** A modern manager-oriented design cleanly separates concerns (Session management, Model management, Environment, Training loop, etc.) [3]. Each manager class has a single responsibility, enhancing maintainability and testability. The architecture is clearly documented with diagrams and explanations [3] [4]. - **Comprehensive Documentation & Tooling:** The repository includes thorough documentation (design documents, code map, component guides) and an advanced CI/CD pipeline definition. There is evidence of a strong DevOps culture – e.g. configured linting (flake8), formatting (Black), type checking (mypy), security scanning (Bandit), and continuous testing in CI [9] [10]. Developer utilities like pre-commit hooks and profiling scripts are provided [11] [12]. - **High Code Quality via AI Guidance:** The code was generated with AI (GitHub Copilot) under human supervision [13], resulting in a remarkably consistent style. Best practices (use of context managers, avoidance of global state, careful error logging, etc.) are evident throughout. The AI-generated code includes self-documenting comments (e.g. markers like "# ADDED" or "# MODIFIED") indicating iterative improvements [14]. This yields code that is both clean and transparently evolved. - **Extensive Test Suite:** Virtually all critical components have corresponding tests. The test suite not only covers unit tests for functions and classes, but also integration tests (e.g. simulating training loops and parallel execution). This indicates a strong culture of correctness. Tests assert fine-grained behaviors – for instance, verifying illegal moves are rejected with proper errors, or that training metrics and outputs remain within expected ranges.

Top 3 Critical Action Items: Despite the overall strong quality, the audit identified a few urgent issues to address: 1. **Overly Broad Exception Handling – *Moderate Risk*:** Several modules catch `Exception` generally (e.g. training startup and environment resets) [1] [15]. This could mask real errors or make debugging harder. **Action:** Narrow these `try/except` blocks to specific exceptions or allow exceptions to propagate after logging, so that failures are not silently ignored. 2. **Complexity in Shogi Engine – *Moderate*:** The `ShogiGame` class and related logic (move generation, rule enforcement) are very large and complex (over 1K lines) [16]. While functionality is correct per tests, this "god module" could hinder future modifications. **Action:** Refactor the Shogi engine into smaller units or add further documentation/comments within it. Consider splitting some logic into helper classes or functions for clarity. 3. **Manager Initialization Sequence Coupling – *Low*:** The Trainer/Setup flow involves many interdependent managers (SessionManager, EnvManager, ModelManager, etc.), which must be initialized in the correct order [5]. Currently this is handled in `Trainer.__init__` and `SetupManager`, but it's complex. A mis-ordering could cause runtime errors. **Action:** Document and enforce the initialization order (perhaps via checks or a factory function) to reduce risk of misconfiguration if the code is modified. In the future, consider simplifying the startup sequence or consolidating managers if possible.

Scope & Methodology

Audited Code Components: The audit encompassed a manual review of the **latest** `main` **branch** of the repository, including:

- **Core Library (`keisei/` package):** All submodules such as `core/` (PPO algorithm, neural network interface, experience buffer), `shogi/` (game engine and rules), `training/` (training loop managers, parallelization code), `evaluation/` (evaluation loop and strategies), and `utils/` (logging, checkpointing, etc.).
- **Test Suite (`tests/` directory):** All unit tests and integration tests, including subdirectories like `tests/evaluation/` and any standalone test files (e.g. `test_shogi_*.py`, `test_ppo_agent_core.py`, etc.).
- **Root-Level Scripts & Configs:** Key Python scripts at the project root (`train.py` entry point [17], any utility scripts in `scripts/`), configuration files (`default_config.yaml`), and project documentation (`README.md`, ops plans, design docs under `docs/`, etc.).
- **Project Metadata:** Build and config files such as `pyproject.toml` (for dependency specs and tool configuration) [18] [19], CI workflow definitions, and test configuration (`pytest.ini`, `requirements-dev.txt` for dev tools, etc.).

Analysis Techniques: The audit was conducted via comprehensive **manual static code review**. Key techniques and focus areas included:

- **Line-by-Line Source Inspection:** Each module was read to identify logic implementation, adherence to design, and any red flags (e.g. misuse of APIs, potential `None` dereferences, off-by-one indexing in arrays, etc.). This included manually tracing important methods (such as `Trainer.run_training_loop`, `ShogiGame.make_move`, `PPOAgent.learn`) to verify correctness of control flow and state changes.
- **Execution Path & Logic Tracing:** Using the entry point (`train.py`) as a guide, the full training workflow was mentally executed: configuration loading, trainer initialization, self-play loop, model updates, logging, and evaluation callbacks. This helped ensure that data flows (e.g. game states to experience buffer to learning) are coherent and that termination conditions (game over, training end) are handled properly.

- **Architectural Pattern Analysis:** The design was examined for common patterns and anti-patterns. We identified intended patterns (e.g. **Manager/Orchestrator pattern** [3], dependency injection for models and config, use of abstract protocols) and checked for potential violations of SOLID principles. We specifically looked for any singletons, global state, or cyclic dependencies – none of which were present beyond expected config sharing. We also noted places of **over-engineering** (if any) such as possibly excessive abstraction.
- **Defensive Coding & Error Handling Review:** The audit included scanning for broad try/except blocks, use of asserts, and input validation. This was essentially a manual **vulnerability assessment** for both reliability and security issues. Common weakness patterns (buffer overflows aren't applicable in Python, but we checked for risky uses of `eval()`, file handling without secure defaults, and whether inputs like configuration files are validated). The config system uses Pydantic for validation [20], which is a robust approach. We also inspected for any hardcoded secrets or credentials – none were found (W&B API key is expected via `.env` file, not committed).
- **Test Suite Assessment:** The presence and content of tests were used to guide the audit. We cross-checked areas where tests failed in the past via issue tracker (none reported in Issues) and ensured that critical logic (game rules, training math) have corresponding tests. The audit treated the tests as an oracle to understand intended behavior, and also looked for any test **gaps** (areas of functionality not covered by tests).
- **AI Code Fingerprint Analysis:** Given the code was AI-generated, we applied a meta-analysis to identify patterns potentially introduced by AI. For example, repetitive code constructs, unused parameters, or inconsistent naming were sought. We also looked at commit history artifacts included in code comments (e.g. references like "Fix B6" [21] or "# ADDED" comments [14]) that indicate iterative prompting. This helped assess if any code sections might be less trustworthy or require human review (none stood out as problematic; the AI contributions appear well-curated).

Throughout the audit, representative code snippets were examined to support findings. In the following sections, each finding or observation is linked to the specific source lines in the repository that substantiate it.

Detailed Findings

This section is divided into **(A) Code-Level Findings** that address implementation details, and **(B) Strategic & Architectural Findings** that address higher-level design and maintainability.

A. Code-Level Findings

1. Overly Broad Exception Handling (Inappropriate Defensive Coding) – Severity: Low. Several places in the code catch exceptions too generally, which could conceal underlying problems. For example, the training launcher catches any `Exception` when setting the multi-process start method [1]. Similarly, `EnvManager.reset_game()` catches a generic `Exception` when resetting the game state [15], and the PPOAgent initializer catches any exception during optimizer setup [22]. In these cases, the code logs the error and continues, which is good for robustness but **risky** if a serious issue occurs (e.g. game state failed to reset). This defensive approach could lead to continuing in a corrupt state or with suboptimal defaults without clearly signaling a failure.
Impact: Debugging and stability suffer – an unexpected error (say, file permission issue on checkpoint, or a model parameter error) might be hidden behind a generic log, making it hard to know something went wrong. In worst cases, training might proceed with invalid assumptions (e.g. not actually seeding environment).
Recommendation: Catch specific exception types where possible (e.g. `except RuntimeError` for PyTorch or environment errors). If a truly unexpected exception occurs, consider letting it propagate

after logging, so that the program fails fast instead of in a possibly incoherent state. At minimum, include the exception details in the log (which is already done via `log_error_to_stderr`) and ensure that suppressed errors won't lead to downstream issues. Overall, reduce use of blanket `except Exception` to only the outermost loops or CLI entry points.

2. Error Handling of Illegal Game Moves – **Severity: Low.** The Shogi game engine rightly implements checks for illegal moves and throws exceptions (e.g. if moving a piece in an invalid way or dropping a pawn where it's not allowed) [23] [24]. However, one scenario to verify is how these exceptions are handled during training self-play. The code appears to rely on the design that the *Agent* will only propose legal moves using the `legal_mask`, so an illegal move exception should never occur during normal training/evaluation. If it did, `Trainer` or `StepManager` would likely not catch it explicitly (no evidence of a catch around `make_move` in training loop). This is a *defensive coding note:* the system trusts its components (which is fine in a controlled environment). In production or less controlled input scenarios, one might want a safety net.
Impact: In practice, due to the comprehensive use of `legal_mask` when selecting moves [25] [26], the risk is minimal. If an illegal move exception were thrown, it would bubble up and likely crash the training loop – which is preferable to continuing erroneously. Tests confirm that illegal moves are indeed prevented by legal masking (there are tests for move legality and skipping if no legal moves) [27] [28].
Recommendation: No immediate change required given the design. It might be beneficial to wrap self-play steps in a try/except to catch any *unexpected* `ValueError` from `ShogiGame.make_move` and treat it as a terminal event for that episode (to avoid crashing the entire training). Logging the incident for debugging would help. This is a minor suggestion for extra safety rather than a correction of a bug.

3. Performance Hotspot – Model Weight Statistics in UI – **Severity: Low.** The training `Display` component computes detailed weight statistics for the neural network on the fly during training to show in the live TUI (trends of weight means/std) [29] [30]. This involves copying model parameters to CPU (`.cpu().numpy()`) each time metrics are updated. If the display refresh rate is high or the model is large, this could become a non-trivial overhead. Similarly, generating the "architecture diagram" text for the model each time is slightly costly [31] [32].
Impact: In the default configuration, the refresh interval (`--render-every`) can be tuned to mitigate this, and mixed precision is used for training. The overhead is likely negligible relative to training computation, *unless* refresh is too frequent or model extremely large. In worst case, it could slow down training steps or introduce minor stalls.
Recommendation: Monitor the performance impact of the Rich display. If it shows up in profiling (e.g. the CI's automated profiling report), consider updating the UI less frequently or making the weight stats computation optional. The design already allows disabling certain UI components via config (e.g. `enable_board_display`, `enable_trend_visualization`, etc.), which is good. Document that if users encounter slow TUI updates, they can increase the `render_every` interval or disable weight statistics.

4. Redundant/Unused Code Sections – **Severity: Low.** The static review did not find major dead code, but a few small items stand out as potentially redundant: e.g., in `DisplayManager.setup_display()`, there is logic to handle different layouts, and a `AdaptiveDisplayManager` is imported but its usage is not clearly seen – perhaps planned for dynamic layout adjustment but not fully utilized. Also, constants for features and alternative action spaces exist (6480 moves) in `constants.py` [33] though the current implementation uses the full 9x9x9 moves mapping (probably these constants are kept for experimentation). These aren't problematic, just notes of code that is in place for extensibility.
Impact: Minimal – a bit of extra maintenance surface. Unused constants or classes don't affect runtime.

Recommendation: Periodically review and prune truly unused code to keep the codebase lean. Given the project's phase (a demonstrator), it's understandable to have placeholders for future capabilities like alternative action spaces or adaptive displays. Just ensure they remain documented or removed once the project evolves beyond needing them.

5. Logging of Critical Events – Severity: Low. The unified logger is well-implemented, and important events (errors, warnings) are logged to both file and stderr consistently [34] [35]. One improvement area: certain rare events, like `PolicyOutputMapper` encountering an unrecognized move (in case the neural net proposes an illegal move index outside known moves), are logged using a cache to avoid spamming [36] [37]. This is clever, but if such an event happens even once it indicates a serious synchronization issue between the game and policy. The code currently would log it and continue.
Impact: A single occurrence might be fine, but repeated occurrences could flood logs (which the cache prevents) but also indicate the model or mapping is out of sync.
Recommendation: Treat any "unrecognized move" log as a bug to be fixed rather than a tolerable event. In the future, consider raising an alert or exception if it occurs frequently even with the caching (since it likely means the move space definition changed without retraining the policy). This is a precautionary note; the design already tries to handle it gracefully.

6. Adherence to Best Practices & Code Style – Positive Finding: The code generally adheres to PEP8 and other Python best practices (helped by automated formatting and linting). Long functions are broken into logical blocks with commented headers (see `make_move` in `ShogiGame` which is divided into parts with comments [38] [23]). Variable and function names are explicit. The use of type hints is pervasive, aiding readability and enabling static type checking (mypy). The few `# pylint: disable` comments (e.g. disabling "too-many-lines" for `shogi_game.py` due to its length [39]) are justified by the complexity of Shogi rules. No significant code smells like deep nesting or duplicate code were observed. The presence of some **AI-generated markers** (comments like "# ADDED:" in code) are unusual in hand-written code but indicate transparency in how the code evolved [14]. They do not affect runtime and arguably improve understandability by explaining why something was introduced (e.g. "# ADDED: is_in_check function..." clarifies that functionality).

B. Strategic & Architectural Findings

1. Manager-Based Architecture – Benefits and Overhead: The **Manager-Orchestrator pattern** is a defining architectural feature of this project. The `Trainer` class acts as the orchestrator, instantiating and coordinating 9 manager components each handling a distinct concern [3]. This yields strong modularity – for example, one could modify how environments are managed (EnvManager) or how evaluation is triggered (CallbackManager) in isolation. The design clearly follows SOLID principles, especially Single Responsibility (each manager has a focused purpose). Moreover, it facilitates testing: e.g., one can imagine swapping out or mocking `ModelManager` or `EnvManager` in tests. The project documentation explicitly touts these benefits [4], which the audit confirms.
However, the **flip side** is a level of complexity: new developers must understand interactions between many classes. There is a fair amount of boilerplate to initialize everything. The *traceability* is good (thanks to consistent naming), but debugging an issue might require checking multiple places (did the Trainer properly initialize X Manager, did the Manager set up Y correctly, etc.). This isn't an *anti-pattern* per se – it's a deliberate trade-off of complexity for flexibility.
Recommendation: Maintain a high-level **architecture diagram** (the ASCII one in the README [40] is excellent) and perhaps include a sequence diagram of the training loop in docs. This will help new contributors navigate the orchestration. In the long term, if certain managers remain lightweight, consider merging or simplifying (for instance, if `SetupManager` only calls a few functions on others, its role might be folded into Trainer). But any such change should be weighed against the loss of clarity in separation. For now, the architecture is sound and well-justified.

2. Configuration Management & Integrity: The project uses **Pydantic** for configuration (`AppConfig` and submodels) [41], loaded from YAML with override merging [42] [43]. This is a robust solution that ensures type validation and defaulting. The audit found that config values are propagated throughout the system (passed into managers, used to set hyperparameters, etc.) in a transparent way. The design avoids "hidden defaults" – all defaults are in `default_config.yaml` and the `AppConfig` schema, making it easy to see what the system will do without reading the code. CLI overrides are handled gracefully (with support for flat env var style keys and dot notation) [44] [45].
One integrity aspect checked was whether runtime changes to config could occur (which might cause inconsistencies). The config object is treated as read-only after initialization; there's no code that mutates `config` mid-run (a good practice). Also, the usage of config in code is straightforward – no instances of reading from config in one place and not in another (which could lead to divergence) were seen; all components use the same unified config object.
Finding: There is an implicit assumption that the config is valid – because Pydantic ensures this – so components often don't double-validate values. For example, if `config.training.minibatch_size` were zero or not a divisor of steps per epoch, some logic might break. But the default config and validation likely prevent such cases (and tests like `test_scheduler_logic` use special constants to ensure edge cases are handled).
Recommendation: The configuration system is a strong point. Future work might include adding **dynamic config adjustment** (for example, changing some parameters on the fly for research experiments) – if so, that would require careful handling to maintain consistency. For now, continue to leverage Pydantic's validation (the project pins a recent Pydantic version which is wise [46]). Also, keeping the `default_config.yaml` in sync with code expectations is important; the audit did not spot discrepancies, thanks to the single source of truth approach.

3. Potential Over-Engineering vs. Simplicity: The codebase, having been built with the aid of AI, sometimes leans toward very **comprehensive solutions** that could be simplified. Examples: - The **PolicyOutputMapper** in `utils` generates the full mapping of moves to indices for the action space. This is necessary, but the design caches unrecognized moves to log only a few [36] – a level of detail that might be overkill in practice (since ideally there are none). It's not harmful, but illustrates the thoroughness. - The training loop uses a separate `TrainingLoopManager` and `StepManager` and `MetricsManager` to break down the process. Conceptually, this is excellent for clarity, but practically these might have been in one loop class. The current design means, for example, that `StepManager` must communicate with `MetricsManager` (through `trainer.metrics_manager` reference) for stats – which is a mild coupling through the Trainer. A simpler design might have had a single `LoopManager` doing all, at expense of size. - There are also multiple layers of abstraction in evaluation (an `EvaluationManager` that uses strategies under the hood, etc.), whereas a simpler approach might hard-code a single evaluation mode for a demo. Again, the chosen design favors extensibility (one can imagine adding new evaluation strategies easily).

These observations are not issues but conscious design decisions to make the project "enterprise-grade." Given the project goals (demonstrating AI capabilities), the over-engineering is actually a strength, showcasing a production-quality setup. It does mean more code to maintain than a minimal implementation.

Recommendation: Remain mindful of complexity as the project grows. It's easier to remove or condense managers later than to add them, so the current approach is fine. But if contributors struggle to understand the separation, adding more **high-level documentation** (such as a "How data flows through managers" guide) will help. The existing docs are good; perhaps an *ADR (Architecture Decision Record)* could be added to explain why the 9-manager architecture was chosen – this answers future questions on whether it can be simplified.

4. Developer Onboarding & DX: From a newcomer's perspective, the project is quite inviting: the README provides a project overview, setup instructions, and even an ASCII diagram of the architecture [3]. The presence of a *"Code Map"* and design doc (as referenced in README [47]) is extremely helpful. Running tests and linting is straightforward thanks to standard tooling. The fact that the repository uses `pip install -e .` for development and has a `requirements-dev.txt` with all necessary tools means a developer can quickly get the same environment. Additionally, the CI pipeline documentation [48] [9] essentially acts as a contributor's guide on what quality gates exist. This is excellent DX practice. One aspect noted is the heavy dependency on having certain services (Weights & Biases) – although W&B is optional, a new dev might need to set `WANDB_DISABLED=true` or similar if they don't want to use it, to avoid any friction (particularly since the code will attempt to connect if enabled). The docs do mention how to configure W&B [49]. Perhaps a note in the README about "If you just want to run locally without W&B, set wandb.enabled: false in the config" could be added for clarity.
Recommendation: Continue maintaining documentation as code evolves. The test suite refactor note in docs suggests the team actively improves DX by refactoring tests and pipeline – keep that up. In future, maybe provide a small **example dataset or saved model** so new users can experiment with evaluation or see training progress quickly without running for hours – this can improve the onboarding experience by providing immediate feedback.

5. Data Modeling & State Management: The core data structures include the `ShogiGame` state (board, hands, move history), the `ExperienceBuffer` (storing gameplay transitions for training), and the neural network models. These are well-contained. The `ShogiGame` class is central to state – it encapsulates all game state and provides methods to query or mutate it. Notably, it keeps a history of board states for repetition detection (sennichite) [50] [51], showing foresight in handling draw conditions. The `ExperienceBuffer` (in `core/experience_buffer.py`) manages trajectories and calculates advantages (using GAE – Generalized Advantage Estimation). The audit found this logic to be straightforward and in line with standard PPO implementations.
There is minimal use of global state. One potential pitfall could be the **random number generation**: the code sets seeds via `SessionManager.setup_seeding()` (which seeds Python, NumPy, and PyTorch RNGs). Additionally, `PPOAgent` uses a local `np.random.Generator` for shuffling data [52]. Because each `Trainer` instance seeds these on init, training runs are reproducible if the same seed is provided – a very good practice. We saw no evidence of any time where deterministic vs nondeterministic behavior would surprise (there's even a test checking deterministic action selection when `is_training=False` vs stochastic when true [53] [54]).
Recommendation: Data and state handling is robust. For further assurance, one could add **assertions** or checks in critical parts: e.g., after each training epoch, assert that the experience buffer is empty or fully processed, or assert that no piece remains in an illegal state after a move (though presumably the game rules prevent this). These would act as internal invariants. But given the high test coverage, it might be unnecessary in production code.

6. Module Coupling and API Boundaries: The interfaces between modules are well-defined. For instance, the `ActorCriticProtocol` defines what a model must implement (a `get_action_and_value` method) [55], and the rest of the system calls this on any model, enabling easy swapping of different neural net architectures. The `EnvManager.setup_environment()` returns a `ShogiGame` and `PolicyOutputMapper` [56] [57], which are then used by the Trainer – this cleanly separates environment instantiation from usage. One area to watch is **tight coupling via the Trainer**: since `Trainer` holds references to all managers and other objects, it becomes the integration point for everything. This is fine (it's by design, akin to a controller), but means `Trainer` is somewhat a god object in terms of knowledge (though not in terms of implementation – it delegates rather than doing the work itself). For example, the `Trainer` needs to know to call `evaluation_manager.setup(...)` after everything is initialized [58], and it needs to pass its internal

state to `DisplayManager.setup_display(self)` [59]. These are cross-module interactions managed in one place.
Recommendation: Continue to use the Trainer as the single integration point to avoid sprawling cross-manager calls. The current approach is that managers mostly call back to the Trainer (or use Trainer's stored references to others). This is acceptable given the controlled environment (no dynamic loading of managers at runtime, etc.). If in future more dynamic behavior is needed (hot-swapping components, or running multiple trainers in parallel), you might need to further formalize these APIs. At present, the boundaries (e.g., what `EnvManager` provides vs. what `StepManager` expects) are clear and documented in code comments.

7. Testability & Coverage: The design of components in isolation has paid off in testability. Many tests use *fixtures* (as seen by references to `minimal_app_config`, `ppo_test_model`, etc. in test code [60]) to provide each unit with a controlled environment. For example, the `PPOAgent` is tested by injecting a dummy model and a minimal config [60], ensuring that learning and action selection can be verified without running a full training loop. The Shogi rules are tested extensively by directly manipulating a `ShogiGame` instance (setting up board states and calling rule functions) [61] [62]. This indicates that the modules are sufficiently decoupled to be tested on their own (a direct result of the architecture). Integration tests exist for multi-component behavior (e.g., an integration smoke test to run a short training and ensure nothing crashes).
Finding: One possible gap might be testing of the distributed training (`--ddp` flag) or multi-process self-play in CI – those are harder to test reliably. The CI pipeline mentions a parallelism smoke test job [63], so it's likely covered at least superficially. No critical untested area was identified in manual inspection; even things like WandB logging have a toggle and the code is written such that if W&B is disabled, it won't affect core logic (so it's fine that W&B-specific code isn't directly tested).
Recommendation: Continue writing tests for new features. If implementing complex new strategies (like a new evaluation tournament or a new model architecture), maintain the practice of writing targeted tests (the `component_audit` docs in the repo suggest a culture of auditing each component, which is great). The test inventory in the Appendix provides a mapping to ensure each major module has corresponding tests.

Forensic Deep Dives

In this section, we provide deep-dive analyses for two areas of concern identified as having significant impact: **exception handling robustness** and **Shogi game logic complexity**. These deep dives examine the root causes, explore the code in detail, and offer specific recommendations.

Deep Dive 1: Exception Handling Robustness in Training Loop

Issue: The audit flagged that broad exception catches in the training startup could potentially hide issues (Finding A1). Here we examine the context and evaluate how this might play out during execution, using the multi-processing start method setup as an example.

In `train.py`, before launching the main training, the code tries to set the Python multiprocessing start method to 'spawn' for safety on CUDA [1]. It does this inside a try/except that catches `RuntimeError` (if the start method is already set) and a bare `Exception` for any other error [1] [64]. The use of `log_error_to_stderr` in both cases means any error will be printed to stderr, so the operator will see it. After logging, the code proceeds without raising, meaning training will start even if the start method couldn't be set.

Implication: If an exception occurs here (for instance, some unusual system issue where `set_start_method` fails unexpectedly), training continues using the default start method. This *could* be fine, or it could lead to subtle bugs (e.g., using 'fork' start method with CUDA can sometimes cause issues). The operator might miss the stderr message amid training logs. However, since this is at launch, it's likely noticeable. Moreover, the code does attempt the safe operation and logs why it failed, which is reasonable behavior in an "operator-controlled" environment as stated (the operator can decide to stop if needed).

Similar Patterns: We find similar broad catches in `EnvManager.setup_environment` where seeding failures are caught [65] – in that case, a warning is logged and training continues without a seed. This could impact experiment reproducibility without the user realizing that seeding failed. The PPOAgent's optimizer init catch [22] is another – if an invalid learning rate is provided (perhaps 0 or negative), it logs the exception and falls back to `1e-3`. This is actually a user-friendly recovery (since training can proceed with a default LR rather than crash). Yet, if the config was wrong, arguably failing fast might force the user to fix their config. The project chose resiliency over failure in these cases.

Recommendations (Detailed): For each of these: - In `train.py`: After logging the error setting start method, it might be wise to also log an explicit **"Proceeding with default start method."** Currently, it logs the exception and the current start method [66]. Making it clear that it's non-fatal helps. This is a minor logging improvement. - In `EnvManager`: If seeding fails (perhaps the game doesn't support a seed call or RNG issue), consider propagating that up as a warning in Trainer (so it's visible in a higher-level log or the UI). Right now it logs and continues [67]. If reproducibility is important, the user should know the run isn't seeded properly. - In `PPOAgent`: The fallback to a default learning rate is pragmatic. To improve, log an **INFO** after fallback like "Using lr=1e-3 due to previous error" so it doesn't get lost in ERROR logs. Tests could even assert that for a bad LR, the agent's optimizer ends up with 1e-3; ensuring the fallback works as intended (if not already tested). - General: Introduce a debugging mode (perhaps via config) where these broad exceptions are not caught. E.g., a config flag `strict_mode` that, if true, would cause the program to re-raise exceptions after logging, thus halting execution. This way, during development or testing, a developer can choose to be strict and catch issues early, whereas in production runs they can be resilient.

Conclusion: The broad exception handling observed is intentional for robustness. It generally logs the error (so it's not truly silent) and uses safe defaults to continue. The forensic analysis concludes this is not causing immediate errors, but slight adjustments could enhance transparency and give users more control (strict vs forgiving mode). Given the low security risk profile of the runtime, this approach is acceptable, but from a maintenance perspective, being aware of hidden errors is crucial. The project might document these design choices (e.g., "we prefer to log-and-continue on config errors to avoid run interruption") so operators know to check logs.

Deep Dive 2: Shogi Game Logic Complexity and Verification

Issue: The `keisei.shogi` module, especially `ShogiGame` and `shogi_rules_logic.py`, is complex due to the intricacies of Shogi rules. We dive into how this complexity is managed and verified, focusing on promotion, drop rules, and repetition – traditionally tricky aspects in Shogi.

Game State Complexity: Shogi has more game state considerations than standard chess – pieces can be promoted, captured pieces go to hands for drops, and repetition (sennichite) can cause a draw. The code handles this by: - Maintaining a 9x9 `board` with `Piece` objects that know their type and color. - Maintaining `hands` as dictionaries of piece counts for each player [68] [69]. - A move history and board

hash history to detect repetitions [50] [51]. - Separate logic functions for move generation and rule checks in `shogi_rules_logic.py` (like `can_drop_specific_piece`, `check_for_nifu`, etc.).

Promotion & Moves: In `ShogiGame.make_move()`, we see that after executing a move via `shogi_move_execution`, it records whether a promotion happened and updates the game state accordingly. For example, when a piece moves into the promotion zone, the move execution logic (in `shogi_move_execution.py`) will mark it. The code is defensive: it validates move format strictly [70] [71], checks that the source piece exists and belongs to the current player [24], and even pre-computes potential moves for the piece and ensures the target is one of them [23] (this is a sanity check to prevent moving a piece in a way it's not supposed to move, independent of other game state). That last check is interesting because it duplicates logic from `generate_piece_potential_moves` – essentially double-confirming legality beyond just checking if a move is in the list of legal moves. This redundancy is likely to catch any error in move generation logic or an out-of-sync situation. It shows how carefully the rules are enforced.

Drop Rules: Functions like `can_drop_specific_piece(game, piece_type, r, c, color)` implement Shogi's drop restrictions. The audit found that these functions are well-tested. For instance, the rule that you cannot drop a pawn on a file if you already have one (Nifu) is implemented in `check_for_nifu` and used inside `can_drop_specific_piece`. The test `test_cannot_drop_pawn_nifu_true` sets up a pawn on the same file and asserts a drop is disallowed [7]. Another subtle rule: you cannot drop a pawn to give immediate checkmate (uchi-fu zume). The test suite sets up a scenario to validate the code catches that [8]. This indicates the code likely has a function `check_for_uchi_fu_zume` and integrates it into move legality – given the tests pass, we can infer the logic is correct. The complexity of these rules is thus managed by breaking them into helper functions (`check_for_nifu`, etc.) and combining their results in `generate_all_legal_moves` or drop validation.

Repetition & Draw: The code computes a board state hash (tuple of pieces and hands) [50] and appends it to `board_history` on each move [72] [73]. It provides `is_sennichite()` which calls `shogi_rules_logic.check_for_sennichite(self)` [51]. Although the implementation of `check_for_sennichite` wasn't directly shown in our snippets, the test expectations and usage suggest it returns True if the current state appeared 4 times. Likely, each time a move is made, they check `if game.is_sennichite(): game.termination_reason = "stalemate"` or similar. Ensuring that draw by repetition is handled is a sign of completeness. It's an advanced rule some implementations might skip, but here it's included (and mentioned in the README as supported [74]).

Testing & Verification: The presence of exhaustive tests (including edge cases like dropping a pawn with a promoted pawn on the file [75]) gives high confidence in the game logic. If any rule was implemented incorrectly, tests would likely catch it. For instance, the test `test_nifu_with_promoted_pawn_on_file_is_legal` [75] verifies that a promoted pawn (Tokin) on a file does **not** count as a pawn for Nifu – a subtle detail the code handles correctly (the test passes, meaning the code likely checks only unpromoted pawns for Nifu). This level of nuance in tests indicates the game logic was carefully implemented, likely cross-referenced with Shogi rules.

Performance Consideration: The game logic is pure Python. Generating all legal moves involves scanning the board and simulating moves. In worst case (mid-game with many pieces), this could be on the order of a few hundred moves. Python can handle this, and since Shogi has at most 40 pieces and 81 squares, it's not a huge state space. The code doesn't attempt any micro-optimizations (like bitboards or caching moves), which is fine for a Python DRL setup where the bottleneck is usually neural network computation, not move generation. The audit did not find any glaring inefficiencies here; the clarity of

implementation was prioritized over premature optimization – which is appropriate for this project's goals.

Maintainability of Shogi Logic: If someone needed to modify a rule (say, implement a variant or fix a bug), they'd face a large file and many interconnected functions. The use of clear function names (e.g., `can_drop_specific_piece`, `generate_piece_potential_moves`) and the separation of concerns (move generation vs rule enforcement vs execution) mitigate this. It's complex but logically structured. The heavy commenting and the fact that tests document expected outcomes also serve as a guide for maintainers. This complexity is inherent to encoding a full board game's rules – it's unlikely to be simpler without sacrificing completeness.

Conclusion: The Shogi game logic, while complex, is systematically handled. The deep dive finds that the combination of defensive programming in `make_move`, separation of rule checks, and strong test coverage results in a reliable implementation. There were no signs of logical flaws in the rules as implemented – every rule tested was being enforced correctly by the code. The main recommendation is to continue to rely on tests when modifying this area. For example, if optimizing move generation, ensure all rule-specific tests remain green. Perhaps maintain a "Shogi Rules Reference" document (could be in comments or external docs) that ties each rule to the code that enforces it, to help new maintainers. The existing code is already quite self-explanatory in that regard.

Strategic Action Plan

Based on the findings, we propose a phased action plan to address issues and further strengthen the project. This plan assumes the project will continue to be developed as a high-quality DRL system.

Phase 1: Triage & Stabilization (Immediate)

Focus on quick wins and critical fixes that improve robustness without large structural changes:

- **Address Broad Exception Handling:** Audit all `except Exception` occurrences and replace with narrower exception handling or add re-raise logic. In particular, update `EnvManager.reset_game()` to not silently continue on failure [15], and ensure any config loading issues cause a controlled shutdown after logging (so runs don't unknowingly proceed with bad configs). This is a one-day fix: search and patch patterns of over-broad catches.

- **Logging Enhancements:** Augment logging messages where needed. For instance, after falling back to default parameters (learning rate fallback [22], spawn method fallback [66]), log a INFO line stating the program will proceed with defaults. This ensures operators clearly see what happened in console output. These strings can be added easily. Also, ensure critical warnings (like seeding failures) surface to the main log or UI (perhaps via `Trainer.logger`).

- **Documentation of Known Issues:** Create a short **KNOWN_ISSUES.md** or add to the README any limitations or open bugs discovered. For example, note that distributed training (`--ddp`) has not been fully validated (if true) or that parallel self-play is experimental. This manages user expectations in the immediate term.

- **Test Suite Health Check:** Run the full test suite under various scenarios (different Python versions as in CI matrix, with/without GPU if possible) to ensure there are no flaky tests. Fix any tests that occasionally fail (none were noted, but this is a standard stabilization step). Also, incorporate a strict mode run where exceptions are not caught to see if anything surfaces during

a test (for development, you could temporarily change the broad `except` to rethrow and run tests).

- **Top 5 "Drop-Everything" Fixes Summary:**

- Narrow `except Exception` in `train.py` and `env_manager.py` (no silent failures in startup).
- Add explicit log messages for any fallback behaviors (spawn mode, LR defaults).
- Ensure seeding failure (if ever) is prominently reported.
- Update documentation with any usage caveats (like W&B optional usage).
- Verify no test regressions; if any, fix immediately.

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

Elevate the project's development process and code quality enforcement using automated tools and minor refactoring:

- **Adopt/Enhance Static Analysis Tools:** The project already uses flake8, pylint, mypy, etc., as indicated in `pyproject.toml` [76]. Consider adding **Bandit** (security linter) and **Safety** (dependency vulnerability scanner) as part of CI (the CI plan suggests these are intended [77]). Ensuring these run and pass will catch any future security slips or use of risky functions. Another tool, **Radon** or **Xenon**, could be used to monitor code complexity; for instance, ensure `ShogiGame` doesn't grow more complex or measure cyclomatic complexity of critical functions.

- **Increase Test Coverage Measurement:** Enable `pytest-cov` in CI and set a high coverage requirement (e.g., 90%+) to prevent coverage regressions. The tests are thorough, so this should pass easily. It's mostly to guard against future code being added without tests. The CI workflow should fail if coverage drops too low.

- **Continuous Integration Enhancements:** Implement the CI/CD pipeline as described in docs/CI_CD.md. Specifically:

- A **test job** across multiple Python versions running lint, mypy, tests [9].
- An **integration test job** (maybe just one environment) for running a short training loop as a smoke test [63].
- A **security scan job** running Bandit and Safety [77].
- (Optionally) a **performance profiling job** to keep an eye on training speed over time [78] [79].

This ensures code quality gates are enforced automatically. Given much of this is planned, executing on it is a priority mid-term.

- **Refactor for Clarity Where Easy:** Identify low-hanging refactoring that doesn't change behavior: e.g., break down the `ShogiGame` monolith class a bit. Perhaps move some of its less core methods (like board printing, SFEN serialization) into `shogi_game_io.py` or utility modules, if not already. This reduces `shogi_game.py` length and focuses it on game state. Another example: if `TrainingLoopManager` is very thin, consider merging its functionality into `Trainer` or `StepManager` to cut one layer. These refactors should be done carefully with all tests passing to ensure no behavior change. The benefit is incremental improvement in readability.

- **Documentation & ADRs:** As part of best practices, start an **ADR log** (Architecture Decision Records). Document key choices like "Use manager-based architecture (ADR-0001)" and why, "Not using vectorized environment to preserve game state control (ADR-0002)" or any other major decision points gleaned from development. This will help future maintainers understand past context. Also update component documentation if any changes are made in refactoring.

- **Developer Onboarding Improvements:** Create a quick **start guide** for contributors: e.g., how to run tests, how to format code (Black config is already provided), how to run the profiler. This could be an extension of the README or a CONTRIBUTING.md. Mid-term, this smooths the path for open-source contributions (if that's a goal).

Phase 3: Modernization & Automation (Long-Term)

Looking further ahead, consider larger changes or additions that position the project for long-term success:

- **CI/CD Pipeline – Deployment and Artifacts:** If the project matures to have releases, implement the planned release workflow [80] [81] to publish versioned packages (the groundwork is in pyproject for a package name "keisei"). Automated releases with changelog generation will help users adopt new versions. Also, integrate **Codecov** for coverage tracking over time (as mentioned in CI docs) and possibly **Docker images** if deployment in containerized environments is needed.

- **Scalability Improvements:** Evaluate performance in a real training scenario (e.g., training a model to some moderate strength). If the Python-based self-play becomes a bottleneck, consider options like:

- Integrating a faster Shogi engine for self-play (perhaps in C++ or Rust, or using an existing library) while maintaining the learning loop in Python. This is a significant change and only needed if profiling shows the environment is the slow point.

- Utilizing vectorized environments (multiple games per process) or other parallelism tricks if GPU utilization is low. The current design with `parallel.self_play_worker` is ready for multi-process self-play; expanding on that (ensuring it's fully functional, perhaps adding features like shared memory for observations) could be a long-term enhancement.

- **Neural Network Modernization:** As a DRL project, keeping the neural network approaches up-to-date is key. The code already has provision for different architectures (basic CNN vs ResNet tower). Long-term, you might:

- Incorporate newer techniques (e.g., attention-based architectures for policy if relevant, or more advanced self-play algorithms).
- Ensure compatibility with the latest PyTorch versions and take advantage of features like TorchScript or ONNX export if deploying the model.

- Possibly integrate training on cloud or distributed settings – the code's DDP flag hints at multi-GPU training. Fully test and support that for users who have the infrastructure.

- **User Interface & Monitoring:** The Rich-based TUI is a standout feature. Long-term, adding a **web dashboard** or leveraging W&B more (the code logs to W&B, but one could imagine custom

charts or live game visualization) could enhance the monitoring. This isn't critical for functionality but could be a "polish" item if the project is used in demonstrations or by non-developers.

- **Community and Contribution Pipeline:** If open-sourced and community-driven, automate **lint checks and tests on pull requests** (which is part of CI) and perhaps add bots for enforcing style (like a Black formatter bot). Long-term maintenance benefits from as much automation as possible.

- **Architectural Evolution:** Revisit the architecture periodically. For example, if one manager consistently remains trivial or two managers are always modified together, consider merging them. Or if a new feature doesn't fit well, be open to creating a new manager. The architecture shouldn't be static dogma; use the ADR process to evolve it. For instance, an ADR might propose: "Combine TrainingLoopManager and StepManager into one, to simplify the training control flow." Debate and test such changes – they could reduce complexity with minimal downside if done carefully.

In summary, the long-term actions are about keeping the project current (technically and operationally) and adaptable. The code is already in a good shape to serve as a foundation for future research or production use, so these steps ensure it remains so as the environment (Python, PyTorch, etc.) and usage grows.

Appendix

The following appendices provide detailed artifacts from the audit, serving as a reference for maintainers. These include a symbol map of the codebase, test suite inventory, feature-to-module mapping, an ADR backlog of architectural questions, a complete list of identified issues, dependency review, complexity hotspot analysis, execution flow outlines, and a proposed CI configuration.

A. Code Architecture & Symbol Map

Below is a hierarchical map of the major modules, classes, functions, and variables in the `shogidrl` codebase (package `keisei`). For brevity, not every single function is listed, but all key components are included. This map is current as of the audit and is meant to aid navigation and understanding of code relationships.

- **Module** `keisei.config_schema` – Defines configuration data models using Pydantic.
- **Class** `AppConfig(BaseModel)` – *Overall application config.*
  - **Fields:** `env: EnvConfig`, `training: TrainingConfig`, `evaluation: EvaluationConfig`, `logging: LoggingConfig`, `wandb: WandBConfig`, `display: DisplayConfig` (as seen in README) [41]. *(Each is a BaseModel subclass capturing a section of the YAML config.)*
  - *(Purpose:* aggregates all configuration sections.)*
- **Class** `EnvConfig(BaseModel)`, **TrainingConfig**, **EvaluationConfig**, etc. – *Configurations for respective sections.* For example, `TrainingConfig` includes fields like `learning_rate`, `total_timesteps`, `ppo_epochs`, etc.
  - These classes primarily provide schema and default values; they don't have methods beyond validation.
  - **Relationships:** Used when loading YAML via `utils.load_config`, yielding an `AppConfig` instance [42] [82].

- **Module** `keisei.core.actor_critic_protocol` – Defines an interface (protocol) for neural network models.
- **Class** `ActorCriticProtocol(Protocol)` – *Interface for policy-value models.*
  - **Method** `get_action_and_value(obs_tensor, legal_mask, deterministic)` **-> Tuple(index, log_prob, value):** Should return an action index, its log probability, and value estimate for a given observation.
  - *(Purpose:* allow the Trainer and PPOAgent to interact with any model implementing this protocol, without tying to a specific model class.)*
  - **Relationships:** Implemented by actual model classes (e.g., `BasicCNN` or ResNet in `keisei.training.models`). Called by `PPOAgent.select_action()` [83] .
- **Module** `keisei.core.neural_network` – Likely contains a basic implementation of an Actor-Critic network (not fully shown in snippet, but referenced).
- *Class* `BasicActorCritic(torch.nn.Module)` – *A simple CNN-based policy/value network.*
  - **Method** `forward(obs_tensor)` – Returns policy logits and value. (Exact signature inferred from context, likely something like `def get_action_and_value(...)` to match protocol.)
  - Possibly has submodules for conv layers etc., according to README which mentions a basic CNN and a ResNet option.
  - **Relationships:** Implements `ActorCriticProtocol` (explicitly or by convention). Instantiated via `ModelManager` depending on config (`model_type`). Used by `PPOAgent` .
- **Module** `keisei.core.ppo_agent` – Core RL agent implementing PPO logic [84] .
- **Class** `PPOAgent` – *The Proximal Policy Optimization agent coordinating policy and value updates.*
  - **Init:** `__init__(model: ActorCriticProtocol, config: AppConfig, device: torch.device, name="PPOAgent", scaler=None, use_mixed_precision=False)` [85] . *Sets up the agent with the given model and hyperparameters.*
  - Initializes `self.model` (moves it to device) [86] .
  - Initializes an `optimizer` (Adam) with `learning_rate` from config [87] . Catches exceptions to fallback to a default LR [22] .
  - Sets PPO hyperparams: `gamma` , `clip_epsilon` , `value_loss_coeff` , `entropy_coef` , etc. from `config.training` [88] .
  - `self.policy_output_mapper = PolicyOutputMapper()` to map moves to indices [89] . Stores `num_actions_total` .
  - If provided, sets up gradient `GradScaler` for AMP (mixed precision).
  - **Relationships:** Uses `PolicyOutputMapper` from utils to ensure action space mapping. Stores config and interacts with `ExperienceBuffer` during learning.
  - **Method** `select_action(obs: np.ndarray, legal_mask: torch.Tensor, *, is_training: bool=True)` **-> Tuple(move, index, log_prob, value)** [90] [83] – *Chooses an action given observation and legal moves.*
  - Converts obs to tensor, applies scaler if any [91] .
  - If no legal moves available, logs an error (but still calls model to handle it) [92] .
  - Calls `self.model.get_action_and_value(obs, legal_mask, deterministic=not is_training)` to obtain action index, log-prob, and value [83] . Uses `torch.no_grad()` if is_training (meaning exploration) to avoid tracking grad on action selection.
  - Converts the index to a Move (via `PolicyOutputMapper.idx_to_move`) and returns (move, index, log_prob, value). If no move (index could be None if no legal moves), returns None for move.

    - **Relationships:** Called by `StepManager` or training loop each time the agent needs to act. It relies on the model and legal_mask from the environment.
    - *Method* `get_value(obs_np: np.ndarray) -> float` [93] *– Evaluates the state value of an observation.*
    - Sets model to eval mode, converts obs to tensor, does a forward pass to get value, returns it as float.
    - **Purpose:** Used for things like computing baseline value for advantage calc (likely used in experience buffer or during training loop).
    - *Method* `learn(experience_buffer: ExperienceBuffer) -> Dict[str, float]` [94] *– Performs PPO update on a batch of experiences.*
    - Likely retrieves trajectories from the buffer, computes advantages & returns, and performs multiple PPO epochs of gradient descent.
    - Typical steps: sample mini-batches, compute loss (policy loss clipped by `clip_epsilon`, value loss * coeff, entropy bonus * coeff), take optimizer steps, update learning rate scheduler.
    - Returns a dictionary of metrics (e.g., loss values, KL divergence) for logging. Indeed `self.last_kl_div` is tracked [95]. The function probably updates that and maybe other stats like `last_gradient_norm`.
    - **Relationships:** Called by `TrainingLoopManager` at the end of an epoch (after collecting some timesteps). Uses `ExperienceBuffer` data.
    - **Method `get_name() -> str` – Returns the agent's name. [96] (Trivial, used in tests to verify naming).
    - **Attributes:**
    - `self.model`: The neural network (policy & value).
    - `self.optimizer`: Optimizer for model parameters.
    - `self.scheduler`: Learning rate scheduler (created via `SchedulerFactory` based on config) [97].
    - Various hyperparams: `gamma, clip_epsilon, ...` from config.
    - `self.policy_output_mapper`: for move mapping, with `num_actions_total`.
    - `self.scaler`: for AMP if used.
    - `self._rng`: a NumPy random Generator for shuffling (seeded from config.env.seed) [52].
    - **Inherits/Implements:** Not a subclass of a specific class (just object/nn.Module), but conceptually implements `ActorCriticProtocol` by containing a model.
    - **Interactions:** Interacts with `ExperienceBuffer` (calls its methods to get batches), calls methods on `self.model`. It also logs errors via `log_error_to_stderr` as needed.
- **Function** `SchedulerFactory.create_scheduler(optimizer, schedule_type, total_steps, schedule_kwargs) -> torch.optim.lr_scheduler` *– Utility to create learning rate scheduler.*
    - Likely supports different scheduler types (linear decay, etc.).
    - **Relationships:** Used in PPOAgent init to set `self.scheduler` [97].
- **Module** `keisei.core.experience_buffer` – Handles storage of trajectories for PPO.
- **Class** `ExperienceBuffer` *– Buffer to store experiences (states, actions, rewards) and compute advantages.*
    - **Init:** Takes config params like buffer capacity (e.g., `steps_per_epoch`) and maybe gamma, lam (GAE lambda).
    - **Method `add(state, action, reward, done, value, log_prob)` – Append a transition.*
    - **Method `finish_episode(last_value)` – When an episode ends or buffer is full, compute GAE advantages and returns.*

    - ***Method** `get_batch(batch_size)` – *Yield a batch of experiences for learning.*
    - **Attributes:** stores lists/tensors for observations, actions, etc., and computed `advantages` and `returns`.
    - **Relationships:** Used by `StepManager`/training loop: each time a step happens, add to buffer; when enough steps collected (epoch done), call `finish_episode` (if episode didn't terminate naturally, uses bootstrap `last_value`); then pass to `PPOAgent.learn`. After learning, buffer is reset for next epoch.
    - **Note:** The buffer likely also handles minibatch splitting (shuffling indices using PPOAgent's rng).
- **Module** `keisei.core.scheduler_factory` – Contains `SchedulerFactory` class with static methods to create learning rate schedulers.
- (Since it's straightforward, skip details; PPOAgent covers usage).
- **Module** `keisei.shogi.shogi_core_definitions` – Defines core types and constants for Shogi.
- **Class** `Color(Enum)` **– Enum for BLACK, WHITE.**
- **Class** `PieceType(Enum)` **– Enum for all piece types (PAWN, LANCE, ... PROMOTED_PAWN, etc.).**
- **Class** `Piece` **– Data class for a piece with attributes** `type: PieceType` **and** `color: Color`**.**
    - May have methods like `is_promoted()` or so, but likely just a container.
- **Type aliases:** `MoveTuple` often defined as `Tuple[int,int,int,int,bool]` for a move (from_square, to_square, promotion_flag) or a drop move as (None, None, to_row, to_col, PieceType).
- **Constants:** Mappings like `BASE_TO_PROMOTED_TYPE`, `PIECE_TYPE_TO_HAND_TYPE` (for captured piece conversion), sets of promoted types, etc. [98].
- **Relationships:** Used throughout game logic for type-checking moves, promotions, etc.
- **Module** `keisei.shogi.shogi_game` – Main game state and logic orchestrator [99].
- **Class** `ShogiGame` – *Represents a Shogi board state and rules enforcement.*
    - **Init** `__init__(max_moves_per_game: int = 500)` [100]**:** Sets up initial empty board and hands, then calls `reset()` to initialize starting position. Stores `max_moves_per_game` in `_max_moves_this_game`.
    - Initializes internal state: `board` (9x9 list of lists), `hands` (dicts of piece counts per color) [68], `current_player = Color.BLACK`, `move_count = 0`, `game_over = False`, `winner = None`, `termination_reason = None`, `move_history = []`, `board_history = []`.
    - `_initial_board_setup_done` flag to avoid redoing initial setup if not needed.
    - After `reset()`, an initial board hash is stored in `board_history` [72] [73].
    - **Method** `reset() -> np.ndarray` [69] [73]**:** *Resets the game to the initial state and returns the initial observation.*
    - Calls `_setup_initial_board()` to place all pieces in starting positions (each side's pieces arranged properly) [101] [102].
    - Resets hands to no pieces in hand for both players [103].
    - Resets current_player to BLACK, move_count to 0, game_over False, clears winner and termination_reason, clears move_history.
    - Computes initial board hash via `_board_state_hash()` and stores in board_history [73].
    - Returns `self.get_observation()`, likely a numpy array representation of the board (the `features.py` module handles conversion of game state to neural network input).
    - **Relationships:** Called at game start or if one wanted to reuse the object for a new game. The observation returned is used as the starting state input to the agent.

- **Method** `get_piece(row, col) -> Optional[Piece]` [104] **:** Returns the piece at (r,c) or None if empty/out of bounds.
- **Method** `set_piece(row, col, piece)` [105] **:** Places a piece or None at (r,c) if within bounds. Used in tests and possibly for undo logic.
- **Method** `is_on_board(row, col) -> bool` [106] **:** Checks bounds (0 <= r,c < 9).
- **Method** `to_string() -> str` **:** Returns a text representation of the board (likely using `shogi_game_io.convert_game_to_text_representation`). For debugging.
- **Method** `get_observation() -> np.ndarray` **:** (Not shown, but implied) Uses `features.py` to produce the neural net input (46-channel encoding).
- **Relationships:** Called after every move (especially in training loop to get next state for agent).
- **Method** `_board_state_hash() -> tuple` [50] [107] **:** Computes a hashable representation of the board + hands + current player, for repetition detection.
- Represented as tuple of tuples for board (each piece as (type,value, color.value) or None) and sorted tuples for each player's hand, plus current_player.
- **Method** `get_board_state_hash() -> tuple` [108] **:** Public interface to get the hash (just calls internal). Possibly used externally or in tests.
- **Method** `is_sennichite() -> bool` [51] **:** Checks if current position occurred 4 times (true draw by repetition). Likely delegates to a function in rules logic that counts occurrences of the current hash in board_history.
- **Method** `make_move(move_tuple: MoveTuple, is_simulation: bool = False) -> Union[Dict[str, Any], Tuple(np.ndarray, float, bool, dict)]` [109] [110] **:** *Apply a move to the game state.*
- This is a critical method handling both real moves and simulation (for evaluating move legality without changing game or for self-play steps).
- If `self.game_over` and not simulation, it returns the current observation, reward 0, done True, info with reason [111] (prevent moves after game ended).
- Validates the format of `move_tuple` thoroughly (ensures it's a tuple of length 5 and either a board move ints or a drop move with Nones) [70] [71], otherwise raises `ValueError`.
- Extracts `r_from, c_from, r_to, c_to` from the move tuple [112].
- Prepares a `move_details_for_history` dict to record everything about the move (like whether it's drop, what piece was captured, etc.) [113] [38]. This is used to append to `move_history`.
- If it's a drop move (`r_from is None`): mark is_drop true, record dropped piece type [114] (actual removal from hand and placement will happen in next part).
- If it's a board move:
  - Ensure a piece exists at source and belongs to current player, else raise error [24].
  - **Added legality check:** Compute all potential moves for the piece from (r_from,c_from) via `shogi_rules_logic.generate_piece_potential_moves` and if (r_to,c_to) not in those, raise `ValueError` (illegal movement pattern) [23]. *This ensures, for example, a knight isn't moving like a bishop due to bug.*
  - Populate `move_details_for_history` with piece's original type, color, etc., needed for undo or tracking [115].
  - Check if a piece will be captured at destination and if so note it (not shown in snippet, but likely after move execution they handle capture).

    - Then, the method would:
        - Call `shogi_move_execution.apply_move_to_board(game, move_details)` which actually moves the piece and handles promotion, capture, updating board and hands.
        - Update `current_player` (switch turn), increment `move_count`, and determine `reward` and `done` (e.g., if move caused checkmate or exceeded `max_moves_per_game`). If game over, set `termination_reason` and `winner`.
        - Append `move_details_for_history` to `move_history`, and append new `board_state_hash` to `board_history`.
        - If not simulation, return `(next_obs, reward, done, info)` where `info` might contain details like termination_reason when done.
        - If simulation, likely returns just `move_details_for_history` so that legality checking can be done without altering the actual game state (and maybe they undo it after simulation).
    - **Relationships:** Used by `StepManager` during the real training loop to advance the game and get the observation for the next state and reward to give to agent. Also used in evaluation games.
    - **Severity of complexity:** This function is long and multi-part, but logically segmented and robust as per analysis.
    - **Method** `undo_move()` **(speculative):** Possibly the code supports undo by storing `move_details` (not confirmed, but the presence of original type and color suggests an undo capability for search or evaluation purposes).
    - **Method** `get_legal_moves() -> List[MoveTuple]`**:** Possibly present (not shown, but the tests call `game.get_legal_moves()` in at least one place [116]). If exists, it likely uses `generate_all_legal_moves(game)` from rules logic to compile all moves for current player.
    - **Attributes Recap:**
    - `board[9][9]`: 2D list of Piece or None (current board state).
    - `hands: Dict[int, Dict[PieceType,int]]`: count of each piece type in hand for each player color (indexed by Color.value, 0 or 1) [103].
    - `current_player: Color`: whose turn it is.
    - `move_count: int`: count of moves made so far.
    - `game_over: bool`: if the game ended (by checkmate, draw, etc.).
    - `winner: Optional[Color]`: winner if game_over and not draw.
    - `termination_reason: Optional[str]`: text like "checkmate", "stalemate", "timeout", etc., when game_over. Contains values from `GameTerminationReason` constants [117].
    - `move_history: List[Dict]`: list of move details dicts for each move made (for record or undo).
    - `board_history: List[Tuple]`: list of board state hashes after each move (starting position included) for repetition checking [72].
    - **Relationships:** Central to everything. Created by `EnvManager.setup_environment()` [118]. Used by `EnvManager.reset_game()` to reset state between episodes if needed [119]. `PolicyOutputMapper` uses the game's moves to map to indices, and evaluation logic uses `ShogiGame` to simulate matches.
- (Other functions in `shogi_game` possibly include `is_in_check(player)`, `checkmate_detection`, etc., but many are likely in `shogi_rules_logic`.)
- **Module** `keisei.shogi.shogi_rules_logic` – Contains pure functions for game rules.

- **Function** `find_king(game, color) -> Optional[(r,c)]` [120]**:** finds the king's position for a given color.
- **Function** `is_in_check(game, player_color) -> bool` [121] [122]**:** uses `find_king` and then `check_if_square_is_attacked(game, king_r, king_c, opponent_color)` (not shown but likely present) to determine if that king is under attack. It prints debug info if `debug_recursion` flag is true (for deep debugging of check situations) [123] [124].
  - **Note:** Uses `check_if_square_is_attacked` probably by iterating over opponent moves.
- **Function** `generate_piece_potential_moves(game, piece, r_from, c_from) -> List[(r_to,c_to)]` [125] [126]**:** *Generates squares a given piece can move to (ignoring check and other pieces beyond immediate blockage).*
  - Implements movement rules for each piece type: sets of offsets for kings, golds, silvers, knights, pawns etc., including additional offsets for promoted pieces (e.g., dragon rook moves like a king in addition to rook moves) [127] [128].
  - Determines if a piece is sliding (rook, bishop, lance) [129] [130], and if so, will generate moves in each direction until blocked. Otherwise, uses the preset offsets.
  - This function does **not** account for situations like moving into check or drops, it's just raw moves. It does stop at friendly pieces (can't jump them) and includes captures of opponent pieces [131].
  - **Relationships:** Used by move generation and also by `ShogiGame.make_move()` as a sanity check for move patterns [23]. Also used in tests to verify legal move patterns.
- **Function** `generate_all_legal_moves(game) -> List[MoveTuple]`**:** Not shown but clearly exists (tests call it [132]). It likely:
  - Iterates over all squares, for each piece of `game.current_player`, generates moves via `generate_piece_potential_moves`, then filters out those that are illegal due to other rules (e.g., leaving king in check, drops that violate drop rules).
  - Also includes drop moves: for each piece type in current_player's hand (if count > 0), iterate over board squares to see where it can be dropped (and call `can_drop_specific_piece` for rules like no drop pawn to checkmate).
  - Returns a list of MoveTuple (board moves have form (r_from,c_from,r_to,c_to, promotion_bool), drops have (None,None,r_to,c_to, PieceType)).
  - **Relationships:** Used in evaluation (maybe for random move opponent or to check for checkmates/stalemates).
- **Drop rule functions:**
  - **Function** `can_drop_specific_piece(game, piece_type, r, c, color) -> bool` – checks if a piece of type can be dropped by color at square (r,c). Internally uses:
  - `game.get_piece(r,c)` to ensure target is empty [133].
  - `check_for_nifu(game, color, file)` if piece_type is pawn – ensures no pawn of that color already on that file [7].
  - ensures not dropping pawn/lance on last rank, knight on last two ranks (these pieces cannot drop where they have no legal move) [134] [135].
  - `check_for_uchi_fu_zume(game, color, r, c)` if dropping pawn – to ensure not immediate mate by pawn drop (test covers this) [8].
  - If all conditions satisfied, return True.
  - **Function** `check_for_nifu(game, color, file) -> bool`**:** likely returns True if a pawn of `color` is on given file. The code uses this to *prevent* drop if True.
  - **Function** `check_for_uchi_fu_zume(game, color, drop_r, drop_c) -> bool`**:** determines if dropping a pawn at (r,c) would instantly checkmate the opponent's king with no other moves – a forbidden move in Shogi. Implementation presumably does: simulate drop, see if opponent's king has any escape or if any piece can capture that pawn, etc. This is complex but since there's a test, the logic is likely correct.

    - **Function** `check_for_sennichite(game) -> bool`**:** counts occurrences of `game.board_history[-1]` in the history list and returns True if >= 4 (Shogi repetition rule is 4 repetitions for draw).
    - **Relationships:** These rule-checking functions are called by move generation and by specific scenarios in gameplay (e.g., after each move, one might check if sennichite occurred to end the game).
- **Module** `keisei.shogi.shogi_move_execution` – Handles the act of making a move on the board (moving pieces, handling captures, promotions).
- **Function** `apply_move_to_board(game: ShogiGame, move_details: dict) -> None`**:**
    - If `move_details["is_drop"]` is True: remove one piece from `game.hands` of that type, place that piece on the board at target (with appropriate color).
    - If it's a board move: move the piece from source to target:
    - If target occupied by enemy piece, remove it (capture: add to current player's hand as unpromoted piece type).
    - Check promotion: if move qualifies (piece reaches promotion zone and either promotion flag is set or mandatory for pawn/lance reaching last rank), then change piece's type to promoted version and mark `was_promoted_in_move`. The `original_type_before_promotion` recorded allows undo.
    - Update `game.current_player` will be done outside this function likely.
    - The function might also detect check/mate but likely not, that's done outside by generating legal moves for opponent after move.
- **Relationships:** Called inside `ShogiGame.make_move()` to mutate state. The separation ensures that the game logic can be unit-tested and that state changes are centralized.
- **Module** `keisei.shogi.shogi_game_io` – Functions for converting game state to notations (e.g., SFEN serialization, text display).
- **Function** `to_sfen_string(game) -> str`**:** returns SFEN string for current position.
- **Function** `convert_game_to_text_representation(game) -> str`**:** pretty prints board with ranks and files, used by `ShogiGame.to_string()`.
- **Relationships:** Used for logging or debugging (and internally for check debug printing in `is_in_check`).
- **Module** `keisei.shogi.features` – Handles encoding the game state into neural network input tensors.
- **Function** `build_observation(game: ShogiGame) -> np.ndarray`**:** likely creates a 46-channel (or configurable channels) 9x9 representation as described (e.g., separate planes for each piece type per player, plus indicators).
- **Relationships:** Called by `ShogiGame.get_observation()` to feed into the model.
- **Module** `keisei.training.trainer` – The master orchestrator coordinating all managers [136].
- **Class** `Trainer` – *Orchestrates the entire training process.* [136]
    - **Init** `__init__(config: AppConfig, args: argparse.Namespace)`**:**
    - Stores `config` and CLI `args`.
    - Sets `self.device = torch.device(config.env.device)` (handles "cuda"/"cpu").
    - **Session Setup:** Creates `SessionManager(config, args)` [137] and calls:
        - `session_manager.setup_directories()` (create log and model dirs),
        - `session_manager.setup_wandb()` (init Weights & Biases if enabled),
        - `session_manager.save_effective_config()` (write out the final config for record),
        - `session_manager.setup_seeding()` (set random seeds).
        - After this, it obtains from session manager: `run_name`, directories, log file paths, and a flag if wandb is active [138].

- **Managers Initialization:**
  - `DisplayManager(config, log_file_path)` -> self.display_manager (for Rich UI).
  - gets `rich_console = display_manager.get_console()` and `rich_log_messages = display_manager.get_log_messages()` to pass to logger.
  - Creates `TrainingLogger(log_file_path, rich_console, rich_log_messages)` as `self.logger` [139] (this logger writes to file and console).
  - `ModelManager(config, args, device, self.logger.log)` -> self.model_manager [140] (manages model loading/saving).
  - `EnvManager(config, self.logger.log)` -> self.env_manager [141] (manages environment creation).
  - `MetricsManager(history_size=config.display.trend_history_length, ...)` -> self.metrics_manager [142] (tracks stats like win rates, ELO).
  - Prepares evaluation config via factory `create_evaluation_config(...)`, then creates `EvaluationManager(eval_config, run_name, pool_size, elo_registry_path)` -> self.evaluation_manager [143] [144]. (This likely handles playing periodic evaluation games).
  - `CallbackManager(config, model_dir)` -> self.callback_manager [145] (manages training callbacks including eval triggers).
  - `SetupManager(config, device)` -> self.setup_manager [145] (initializes various components in order).
- **Component Setup via SetupManager:** Calls `self._initialize_components()` [146] which in turn:
  - Calls `setup_manager.setup_game_components(env_manager, rich_console)` -> returns `(game, policy_output_mapper, action_space_size, obs_space_shape)` [147]. This likely calls `env_manager.setup_environment()` and then does any extra policy mapping checks. Assigns these to `self.game`, `self.policy_output_mapper`, etc.
  - Calls `setup_manager.setup_training_components(model_manager)` -> returns `(model, agent, experience_buffer)` [148]. The model manager probably builds/loads the neural net model; the SetupManager then creates a `PPOAgent` with that model and config, and an `ExperienceBuffer` with proper size. Assigns to `self.model, self.agent, self.experience_buffer`.
  - Calls `setup_manager.setup_step_manager(game, agent, policy_output_mapper, experience_buffer)` -> returns a `StepManager` [149]. Assigns to `self.step_manager`.
  - Calls `setup_manager.handle_checkpoint_resume(...)` to load from checkpoint if `args.resume` is set [150]. This might update the agent's model weights, trainer.metrics_manager (to resume stats), etc., and returns a bool `resumed_from_checkpoint` which is stored.
- **Post-Setup Integration:**
  - Call `evaluation_manager.setup(device=config.env.device, policy_mapper=self.policy_output_mapper, model_dir=self.model_dir, wandb_active=self.is_train_wandb_active)` [151]. This passes the live training components into eval manager so it can pull latest model or use the policy mapping, etc.

        - Call `display_manager.setup_display(self)` -> returns a `TrainingDisplay` (Rich UI object) which is stored as `self.display` [152].
        - Call `callback_manager.setup_default_callbacks()` -> returns list of callbacks (like maybe checkpoint saver, evaluation trigger) stored as `self.callbacks` [152].
        - Instantiate `TrainingLoopManager(trainer=self)` -> self.training_loop_manager [153]. This likely will use references to trainer to run the main loop.
        - Initialize any instrumentation variables (`last_gradient_norm`, etc.).
    - **Relationships:** The Trainer now aggregates all managers. The heavy lifting of actual training is delegated to TrainingLoopManager.
    - **Method** `run_training_loop()` **(not explicitly shown but expected):** Delegates to `self.training_loop_manager.run()` or similar. Essentially, this kicks off the main loop:
    - Loop for a certain number of timesteps or episodes:
        - reset env, while not done: agent selects action, environment (game) advances via step manager, experience recorded.
        - when enough steps (like `steps_per_epoch`) collected, call `agent.learn(buffer)` to update policy.
        - periodically invoke callbacks (e.g., evaluation) via callback manager or training loop logic.
    - Update display each iteration or as configured (`render_every`). Use metrics manager to accumulate stats.
    - Stop when `total_timesteps` reached or maybe when converged (if implemented).
    - **Logging/Info:** Trainer uses `self.logger` to log important events (we see it used for error logging in exception catches).
    - **Attributes of Trainer:** beyond managers, it tracks run directories, flags, etc., as set up in **init**. For example, `self.run_name`, `self.model_dir`, `self.log_file_path`, etc., to be used by managers for saving artifacts.
    - **Relationships:** The central coordinator. Other managers sometimes call back into Trainer (for example, `StepManager` might access `trainer.game` or `trainer.agent`). This is the integration hub.
- **Module** `keisei.training.session_manager` – Handles experiment session setup (directories, seeding, WandB).
- **Class** `SessionManager` – *Sets up output folders and random seeds, and integration with Weights & Biases.*
    - **Init:** takes config and args.
    - **Method** `setup_directories()` **– Create directories for models, logs, etc., as per config (default paths or overridden).**
    - **Method** `setup_wandb()` **– If config.wandb.enabled, initialize wandb run (set project, entity, etc.) and set** `self.is_wandb_active`**.** Also maybe configure wandb to save config.
    - **Method** `save_effective_config()` **– Write the final resolved config (after CLI overrides) to a YAML file in the output directory, for record-keeping.**
    - **Method** `setup_seeding()` **– Set the random seed for Python** `random`**, NumPy, and PyTorch (both CPU and CUDA) using** `config.env.seed`**.** This ensures reproducibility.
    - **Attributes:** `self.run_name` (could be based on config or auto-generated with timestamp if not provided), `self.run_artifact_dir`, `self.model_dir`, `self.log_file_path`, etc. These are consumed by Trainer.

- **Relationships:** Called first by Trainer. After calling these, the Trainer copies needed info (like log file path) for other managers. WandB setup influences `MetricsManager` or `EvaluationManager` if they log metrics (they check `trainer.is_train_wandb_active`).

- **Module** `keisei.training.model_manager` – Handles model creation, loading, saving.
- **Class** `ModelManager` – *Manages the neural network models and checkpoints.*
  - **Init:** might load a model from checkpoint if resuming, otherwise instantiate new model based on config (e.g., model architecture selection).
  - **Method** `create_model()` **– Builds a new model instance (Basic or ResNet) based on** `config.training.model_type`**.**
  - **Method** `save_checkpoint(agent, metrics, ...)` **– Save model weights and possibly optimizer, scheduler state to a file (in model_dir, with timestamp or epoch number).**
  - **Method** `load_checkpoint(path)` **– Load weights into model (and possibly load optimizer state).**
  - **Attributes:** Could include `self.model` (if it keeps reference) or it might hand off the model to Trainer and not keep internally. Likely keeps track of latest checkpoint path, etc.
  - **Relationships:** Used by Trainer/SetupManager to get the model. Called in `setup_training_components` to either load or create model. Works with `CallbackManager` for periodic saving or with Trainer at end of training.
- **Module** `keisei.training.env_manager` – Manages environment (game instance and policy output mapping).
- **Class** `EnvManager` – *Initializes and validates the game environment and action space.* [154]
  - **Init(config, logger_func)** – Stores config and a logger function for status messages. Sets internal placeholders for `game`, `policy_output_mapper`, etc. (all None initially) [155].
  - **Method** `setup_environment() -> Tuple(ShogiGame, PolicyOutputMapper)`**:** *Create a new game and policy mapper.* [56] [57]
  - Instantiates `ShogiGame(max_moves_per_game=config.env.max_moves_per_game)` [118].
  - If `config.env.seed` is set, calls `game.seed(seed)` (if game had a seed method; current ShogiGame doesn't, so possibly the ShogiGame class could accept seed for random opponent move generation? In any case, it's inside a try/except) [65].
  - Sets `self.obs_space_shape = (config.env.input_channels, 9, 9)` which the network expects.
  - Creates `PolicyOutputMapper()` and assigns to `self.policy_output_mapper`, then gets `action_space_size = mapper.get_total_actions()` [156].
  - Calls `self._validate_action_space()` to ensure config's declared `num_actions_total` matches mapper's count [157]. If mismatch, raises runtime error (critical config mistake) [158].
  - Returns `(self.game, self.policy_output_mapper)`. Also by side effect, stores them internally.
  - **Relationships:** Called by SetupManager to prepare environment before agent. Ensures the network output size aligns with game's action space.
  - **Method** `_validate_action_space()`**:** Verifies that `config.env.num_actions_total` equals `PolicyOutputMapper.get_total_actions()` [158]. If not, logs critical and raises ValueError. This catches any inconsistency between configured action space size and actual moves (important if using a reduced action space).

- **Method** `reset_game() -> bool`**:** If `self.game` exists, calls `game.reset()`, logs and returns True on success. If exception, logs error and returns False [119]. This is used to reset environment between episodes if needed (e.g., multi-episode training).
- **Method** `initialize_game_state() -> Optional[np.ndarray]`**:** If game exists, calls `game.reset()` and returns initial observation [159]. If error, logs and returns None [160]. (This overlaps with reset_game but also gives observation directly.)
- **Method** `validate_environment() -> bool`**:** Runs a series of checks to ensure the environment is properly configured and functional [161] [162]:
- Checks that `self.game` and `self.policy_output_mapper` are not None and action_space_size > 0 [163] [164].
- Tests that resetting the game yields a consistent observation (calls `game.get_observation()`, then `reset_game()`, then `get_observation()` again, and possibly compare) [165]. This ensures the game can be reset and provides an observation. If any step fails, logs and returns False.
- Returns True if all checks passed. Used likely after setup to assert environment is ready.
- **Attributes:** `self.game`, `self.policy_output_mapper`, `self.action_space_size`, `self.obs_space_shape` stored after setup. Also keeps `self.config`, `logger_func`.
- **Relationships:** Called by SetupManager for initial creation. Could be reused if one wanted to create multiple envs or re-init environment mid-training (not typical in self-play though). `PolicyOutputMapper` is defined in utils, but conceptually EnvManager "owns" it after creation.

- **Module** `keisei.training.step_manager` – Manages the execution of steps within an episode.
- **Class** `StepManager` – *Handles running through a single game (episode) step by step.*
  - **Init:** likely gets references to game, agent, policy_output_mapper, experience_buffer.
  - **Method** `run_episode()` **or similar:** Not certain, but possibly a method that:
  - While not game.game_over and not reach certain step limit:
    - Ask agent for action: `move, idx, logp, value = agent.select_action(obs, legal_mask)`.
    - Pass move to game: `obs_next, reward, done, info = game.make_move(move)`.
    - Push experience into buffer (obs, idx, reward, value, done, logp).
    - If done or episode end, break.
  - If done due to win/loss/draw or reaching max_moves, collect final info, maybe return outcome.
  - Alternatively, `StepManager` might simply provide utility functions for a single step execution (`take_step()`).
  - The README architecture diagram indicates StepManager responsibilities: "Step Exec, Episode Mgmt, Experience" [166]. So likely:
  - **Method** `step()` **– executes one action in environment and logs to buffer.**
  - **Method** `play_episode()` **– resets env and loops until done, populating buffer.** Possibly returns when an episode is finished, maybe with outcome stats.
  - **Attributes:** Could track episode count, maybe keep a move log separate from game's internal (though game has move_history).
  - **Relationships:** Used by TrainingLoopManager to actually run the self-play and fill the buffer. May also interact with MetricsManager to update live stats (e.g., track length of episode, win/loss).
- **Module** `keisei.training.training_loop_manager` – Manages the main training loop across episodes/epochs.

- **Class** `TrainingLoopManager` – *Controls the iterative training process (epochs of self-play and learning).*
  - **Init(trainer)** – likely stores a reference to the Trainer to access all components (game, agent, buffer, etc.).
  - **Method** `run_training()` **(or similar)** – Implements something like:
  - For epoch in range(... until total_timesteps or condition):
    - Maybe for episode in range(N episodes per epoch):
    - `env_manager.reset_game()` (fresh game).
    - `while not game.game_over:` use StepManager to play episode (or do steps until buffer full).
    - Once enough timesteps (config.training.steps_per_epoch) are collected in ExperienceBuffer, call `agent.learn(buffer)` to update PPO.
    - Clear buffer, update any learning rate scheduler (scheduler step).
    - Update MetricsManager with results (like win rates from self-play if any, average reward, etc.).
    - Trigger callbacks: e.g., if evaluation interval reached, call `callback_manager.run_evaluation(trainer)` or similar, which uses EvaluationManager.
    - Render display: use DisplayManager/TrainingDisplay to update Rich UI with metrics (progress, win rates, etc.).
    - Possibly checkpoint model: if interval met, ModelManager.save_checkpoint.
  - End loop when timesteps >= total_timesteps or some stop criterion (maybe a training budget).
  - **Relationships:** The training loop is orchestrated here using the pieces from Trainer. It heavily interacts with StepManager, PPOAgent, ExperienceBuffer, MetricsManager, CallbackManager, etc., mostly via the references in Trainer.
  - **Attributes:** Could include counters for timesteps, episodes, etc., and stopping criteria.
- **Module** `keisei.training.display_manager` – Manages the Rich library console UI.
- **Class** `DisplayManager` – *Initializes Rich Console and Live display for training progress.*
  - **Init(config, log_file_path):** Creates a `Console` (from rich) and sets up a `Live` display or layout structure. Possibly prepares panels for different metrics.
  - **Method** `get_console()` **– returns the Rich Console object.**
  - **Method** `get_log_messages()` **– returns a list (or queue) to which log messages are also written (so they can be shown in the UI log panel).** The Trainer passes this to TrainingLogger to populate.
  - **Method** `setup_display(trainer) -> TrainingDisplay`**:** Creates a `TrainingDisplay` object (defined in `display.py`) configured with current trainer state.
  - The `TrainingDisplay` (not a manager but the UI itself) likely assembles multiple panels: board, recent moves, metrics, etc., and starts a Live render.
  - It uses `trainer.rich_console` and perhaps runs in a separate thread or in the main thread updating on a frequency.
  - **Relationships:** Called by Trainer after all components are ready. The TrainingDisplay will read from trainer (like trainer.game for board, trainer.metrics_manager for stats). `DisplayManager` mainly provides the Rich console and log capture.
- **Class** `TrainingDisplay` (in `display.py`) – *Controls what is shown on the terminal UI.* [167]
  - **Init(config, trainer, rich_console):** Sets up a layout with panels for game board, moves, piece stand, metrics, config info, etc., depending on what's enabled in config [168] [169]. Initializes sub-components: `ShogiBoard` renderer, `RecentMovesPanel`, `PieceStandPanel`, sparklines for metrics trends, etc.

    - Turns features on/off based on `config.display` settings (e.g., enable_elo_ratings, enable_board_display) [170] [171].
    - Calls an internal `_setup_rich_progress_display()` to create a Rich Progress bar with custom columns for training stats (steps/sec, win rates, etc.) [172] [173] and store it along with a layout for placing log panel, progress bar, etc.
    - **Method** `update()` **(implied):** Called periodically (every `render_every` steps or so) to refresh the dynamic panels:
    - Updates the board panel with current board state, moves panel with recent moves, piece stand with captured pieces, trends panel with latest metrics (like episode lengths, or custom metrics). For example, code shows updating metric trends panel with new Progress or Text objects [174], updating stats panel with GameStatisticsPanel data [175] [176].
    - Uses data from `trainer.metrics_manager` and `trainer.step_manager` (like move log, capture counts) to populate panels [177] [178].
    - Also, once per run, populates a config info panel (learning rate, batch size, etc.) [179] [180].
    - For model weight statistics, computes current layer stats and compares with previous to display trends (arrow up/down if values changed) [29] [181].
    - Creates/updates an "architecture diagram" showing input -> core model -> policy/value heads [31] [32].
    - Catches exceptions in rendering each panel and displays error text in that panel instead of crashing the UI [182] [183].
    - **Note:** `TrainingDisplay` is updated inside a Rich `Live` context, probably by the TrainingLoopManager or by DisplayManager on a separate thread/timer.
    - **Relationships:** Accesses trainer's components for data. Notifies trainer if needed (not likely; mostly one-way updating).
- **Module** `keisei.training.metrics_manager` – Tracks aggregate stats across training.
- **Class** `MetricsManager` – *Collects and computes training metrics (win rates, ELO, etc.).*
    - **Init(history_size, elo_initial_rating, elo_k_factor):** Sets up structures for tracking metrics over recent episodes (for trend graphs) and ELO ratings. Possibly maintains counters for wins/draws for each player.
    - **Method** `update_after_episode(result)` **– Given an episode result (win/lose/draw), update counts and compute new win rates or ELO if needed.**
    - Might compute running average of win rate over last N games (history_size).
    - If using ELO, update ratings of players (particularly if self-play between current model and older models as opponents).
    - **Method** `get_metrics()` **– Returns a dict or object of current metrics (win_rate, episodes_played, etc.)** for display.
    - **Attributes:** Could include: lists of recent win rates to plot sparkline, current cumulative stats, ELO ratings dict.
    - **Relationships:** Updated by StepManager or TrainingLoopManager at the end of each episode or evaluation. The Display reads from it to show metrics. If integrated with W&B, it might also send metrics there (depending on implementation).
- **Module** `keisei.training.callback_manager` – Manages callback events during training (like evaluation triggers, checkpointing).
- **Class** `CallbackManager` – *Holds a list of callbacks (functions or objects) that should be invoked at certain events.*
    - **Init(config, model_dir):** Perhaps creates default callbacks: e.g., an evaluation callback that every N timesteps uses EvaluationManager to pit current model vs baseline; a checkpoint callback every M timesteps to save model.
    - **Method** `setup_default_callbacks()` **-> List[Callback]:** Creates and returns standard callbacks (like checkpoint every interval, evaluation every interval, log metrics, etc.). Trainer stores these in `self.callbacks`.

    - **Method** `trigger(event_name, **kwargs)`**:** Possibly used to invoke all callbacks of a certain type (not sure if implemented, but a pattern).
    - **Relationships:** The TrainingLoopManager likely iterates through `trainer.callbacks` and calls those at appropriate times (e.g., end of epoch). Or CallbackManager might have its own loop thread. But given simplicity, likely TrainingLoopManager just uses the list.
- **Module** `keisei.training.setup_manager` – Coordinates initialization of multiple components.
- **Class** `SetupManager` – *Orchestrates multi-step setup processes.*
    - **Method** `setup_game_components(env_manager, console) -> (ShogiGame, PolicyOutputMapper, action_space_size, obs_shape)`**:** calls `env_manager.setup_environment()`, logs to console what's happening, returns the game and mapper.
    - **Method** `setup_training_components(model_manager) -> (model, agent, buffer)`**:** calls `model_manager.create_model()`, then creates `PPOAgent(model, config, device, name, use_mixed_precision=...)`, then creates `ExperienceBuffer` with config (maybe using config.training.steps_per_epoch, etc.). Possibly also passes GradScaler if mixed precision.
    - **Method** `setup_step_manager(game, agent, mapper, buffer) -> StepManager`**:** instantiate StepManager with those components.
    - **Method** `handle_checkpoint_resume(model_manager, agent, model_dir, resume_path, metrics_manager, logger) -> bool`**:** If resume flag is provided (either 'latest' or a path):
    - Determine checkpoint path (if 'latest', model_manager knows how to find latest in model_dir).
    - Load model weights into `agent.model` via model_manager.
    - Potentially, load optimizer/scheduler state if training is to continue exactly.
    - Update metrics_manager if the checkpoint or an accompanying file had stored metrics (so training can continue from where left off).
    - Log via logger that resume succeeded or failed.
    - Return True if resumed, False if not.
    - **Relationships:** Used inside Trainer's `_initialize_components()` to break down the setup. Improves readability of Trainer **init**.
- **Package** `keisei.training.parallel` – Tools for parallel self-play (multiprocess).
- **Module** `parallel_manager.py` – Possibly a manager to spawn multiple self-play worker processes.
    - **Class** `ParallelManager` – Would handle launching `SelfPlayWorker` processes and coordinating collection of experiences via multiprocessing Queues or Pipes.
    - **Relationships:** If `config.parallel.enabled`, Trainer might use ParallelManager instead of StepManager for collecting experiences. Not fully detailed, possibly incomplete if parallel was a stretch goal.
- **Module** `self_play_worker.py` – Defines a worker process routine.
    - **Function/Process** that takes a model (or model weights), runs self-play games in a subprocess, and sends trajectories back to main process via `multiprocessing.Queue`.
- **Module** `communication.py` – Utility for inter-process communication (pickling moves or compressing data).
- **Module** `model_sync.py` – Utility to sync model weights across processes (perhaps broadcasting new weights to workers after each update).
- **Module** `utils.py` – Could include compression or data structure helpers for passing experiences.

- **Status:** The code for parallel execution exists but might not be fully integrated (since parallel tests were possibly marked for future). For audit, it's noted as present but not deeply analyzed due to likely being a thin wrapper around existing logic.
- **Module** `keisei.evaluation.evaluate` – Orchestrates evaluation matches between agents.
- **Function** `evaluate_agent(main_agent, opponents, num_games, ...)` **– would pit the main agent against a set of opponents for** `num_games` **and collect results.** Possibly not needed since EvaluationManager wraps it.
- **Class** `EvaluationManager` (maybe defined in `core_manager.py` as search result suggests) – *Manages evaluation runs during training.*
    - **Init(eval_config, run_name, pool_size, elo_registry_path):**
    - Might maintain a pool of past models for comparison (if doing "previous model pool" evaluation as config suggests).
    - Load any ELO ratings from registry file if exists (to track progress over time).
    - **Method** `setup(device, policy_mapper, model_dir, wandb_active)`**:** After training starts, give it references or info it needs:
    - For example, it might create a copy of current model weights as the "current" version, setup any opponent instances (like a RandomMover opponent, or load a heuristic agent if `opponent_type` is set in config).
    - Prepare environment or tournament structure based on eval strategy (could be self-play (current vs itself), versus a fixed opponent, or a ladder of previous versions).
    - **Method** `run_evaluation(current_agent) -> dict`**:** Conduct one round of evaluation games:
    - Spawn multiple games (possibly parallel if enabled), have the current agent play vs configured opponents (which might be itself, a random policy, or previous snapshots).
    - Collect outcomes (wins, losses, draws, maybe game lengths).
    - Update ELO ratings if configured (if `update_elo=True` in config).
    - Return metrics like win rate, perhaps per opponent.
    - If `save_games` is true in config, save game records to files.
    - **Method** to periodically save model snapshots into a pool if needed (pool_size dictates how many past models to keep for evaluation).
    - **Attributes:** Could include list of opponent agents (instances of BaseOpponent, e.g., random or heuristic), ELO ratings dict, a reference to current agent's model weights for copying, and config for evaluation frequency.
    - **Relationships:** Triggered by CallbackManager or TrainingLoop at configured intervals (e.g., every N timesteps). It may log through Trainer's logger or directly to console/W&B. Opponents might use simplified versions of PPOAgent (e.g., a fixed policy or older snapshot).
- **Package** `keisei.evaluation.strategies` – Might contain different evaluation modes:
- **Module** `ladder.py` – If implementing a ladder tournament.
- **Module** `tournament.py` – If implementing a round-robin or elimination tournament among agents.
- These likely define classes or functions for those formats, used by EvaluationManager depending on `eval_config.strategy` (like "self_play", "single_opponent", "league", etc.).
- **Module** `keisei.utils.agent_loading` – Utility to load an agent (model weights) from file or hub.
- **Module** `keisei.utils.checkpoint` – Additional checkpointing helpers (maybe converting older checkpoints, or saving best model separately).
- **Module** `keisei.utils.move_formatting` – Contains functions to format moves for display (e.g., convert MoveTuple to human-readable like "P7g-7f"). Likely also ensures consistency with SFEN.
- **Module** `keisei.utils.opponents` – Defines simple opponent behaviors:

- For example, **Class** `RandomOpponent(BaseOpponent)` – selects a random legal move.
- **Class** `HeuristicOpponent(BaseOpponent)` – maybe uses some basic piece values to play.
- These are used for evaluation to have a baseline opponent.
- **Module** `keisei.utils.unified_logger` – Provides the logging utility used across code [184].
- **Class** `UnifiedLogger` – *Custom logger that writes to file and optionally to Rich console.* [185] [186]
  - **Methods:** `info(msg)`, `warning(msg)`, `error(msg)`, `debug(msg)` – each formats the message with timestamp and component name and writes to log file and/or console with styling [187] [34].
  - **Method `_write_to_file(msg)` – appends to log file; `_write_to_console(msg, style)` – prints to console (or stderr if no Rich console) [188] [189].
  - Created via `create_module_logger(module_name, log_file_path, rich_console)` factory [190], which sets `name=module_name`. The Trainer uses one logger for all training ("TrainingLogger") by passing module name or using also_stdout flag.
- **Functions:** `log_error_to_stderr(component, message, exception=None)`, `log_warning_to_stderr(component, message)`, `log_info_to_stderr(component, message)` – utility functions to quickly log to stderr with consistent format [35] [191]. These are used in try/except handlers (like in train.py) to standardize error logs without setting up a full logger.
- **Module** `keisei.utils.profiling` – Provides tools to profile code (maybe a context manager to time blocks, or functions to run cProfile and output stats).
- Could include something like `TimingContext` or `start_profiling()/stop_profiling()` that the profiling script uses.
- **Module** `keisei.utils.utils` – Misc utilities (besides those in other util files) [192]:
- **Function** `load_config(config_path: str|None, cli_overrides: dict|None) -> AppConfig` [42] [82] **:** *Loads default_config.yaml, merges overrides from a user config file (if provided) and CLI overrides, returns a validated AppConfig.*
  - Uses `_load_yaml_or_json` to read YAML/JSON [193].
  - Merges overrides (CLI or flat env var style) with base config via `_merge_overrides` [194] and `_map_flat_overrides` (maps uppercase flat keys to nested keys) [45].
  - Validates with `AppConfig.model_validate()` (Pydantic v2) [195]. Logs errors and raises if validation fails.
  - **Relationships:** Used in train.py to get the AppConfig [196].
- **Class** `BaseOpponent(ABC)` [197] **:** Abstract base for opponents in evaluation.
  - **Method** `select_move(game: ShogiGame) -> MoveTuple` **:** abstractmethod to be implemented by concrete opponents (Random, etc.).
- **Class** `PolicyOutputMapper` [198] : *Maps moves to indices and vice versa for the policy network output.*
  - **Init:** Generates `idx_to_move: List[MoveTuple]` and `move_to_idx: Dict[MoveTuple, int]` by iterating over all possible moves on an empty board:
  - Likely goes through every from-square, to-square, promotion option, and also drop moves for every piece type and target square, and compiles a complete list (which could be 27x81*2? For Shogi possibly 5937 moves in standard action space).
  - We see it uses a cache for unrecognized moves (`_unrecognized_moves_log_cache`) to avoid logging duplicates [199].
  - **Method** `get_total_actions() -> int` **:** returns len(idx_to_move).
  - **Method** `get_legal_mask(legal_moves: List[MoveTuple], device) -> torch.Tensor` **:** likely produces a boolean mask of length = total_actions, with True at indices corresponding to moves in legal_moves. The agent uses this to mask out illegal moves.

    - **Relationships:** Created during PPOAgent and EnvManager setup to provide consistency between environment and policy. Used every time agent selects action (legal_mask is built through it). It's central to ensure the neural network's output space corresponds exactly to game moves.
    - **Note:** The presence of the `_unrecognized_moves_logged_count` and caches suggests if the model outputs an index not in move_to_idx mapping (shouldn't happen if things are consistent), it will log it as a warning but only a few times [200].
- **Root-level scripts:**
- `train.py` [17] [201] – Main script for training (as detailed above, sets up argparse, calls `Trainer.run_training_loop()`).
- `train_wandb_sweep.py` – A variant to integrate with Weights & Biases hyperparameter sweeps. Likely reads os.environ for sweep parameters, merges into config, then calls into training (may reuse Trainer).
- `profile_training.py` (in scripts/) – Runs a short training for profiling, possibly with options for number of timesteps, and outputs profiling info (e.g., cProfile stats).
- `run_training.py` (in scripts/) – Possibly a convenience wrapper to launch training with certain presets (maybe to run multiple seeds or do something in a loop).
- **Test-related scripts**: `run_local_ci.sh` [202] – runs lint, tests, etc., locally as CI would.

(The symbol map above provides a high-level inventory. For detailed function signatures and interactions, refer to the inline code comments and documentation within the repository. All components are traceable via the references provided.)

B. Test Suite Inventory

The following table enumerates the test files in `tests/` along with their test functions and the aspects of the system they cover:

Test File	Contained Tests (Functions)
`tests/test_shogi_game_core_logic.py`	`test_initial_setup`, `test_move_basic`, `test_capture`, `test_checkmate_detection`, `test_draw_by_repetiti`

`tests/test_shogi_rules_and_validation.py`	`test_can_drop_piece_empty_square`, `test_cannot_drop_`
	 `test_can_drop_pawn_nifu_false`, `test_cannot_drop_`
	 `test_nifu_with_promoted_pawn_on_file_is_legal`, `test_cannot_drop_pawn_last_rank_black/white`,
	 `test_cannot_drop_lance_last_rank_black/white`, `test_cannot_drop_knight_last_two_ranks_black/white`,
	 `test_can_drop_gold_any_rank`, `test_cannot_drop_pa`

Test File	Contained Tests (Functions)
`tests/test_shogi_engine_integration.py`	`test_play_full_game_no_errors`, `test_repeat_game_dra`
`tests/test_shogi_board_alignment.py`	`test_board_text_representation_aligned`
`tests/test_shogi_utils.py`	`test_sfen_serialization_deserialization`, `test_move_` `test_move_formatting_drop`
`tests/test_ppo_agent_core.py`	**Classes:** `TestPPOAgentInitialization` with tests: `test_ppo` `test_ppo_agent_get_name`, `test_ppo_agent_num_actions_total`; `TestPPOAgentAc` `test_select_action_basic`, `test_select_action_with_g` `test_select_action_deterministic_vs_stochastic`; with `test_get_value_basic`, `test_get_value_batch_consistency`; `TestPPOAgentBa` `test_learn_basic`

Test File	Contained Tests (Functions)
`tests/test_session_manager.py`	`test_directories_created`, `test_wandb_setup_disabled`, `test_save_config_file`
`tests/test_evaluate_agent_loading.py`	`test_load_and_evaluate_agent`
`tests/evaluation/test_evaluate_opponents.py`	`test_random_opponent_moves`, `test_heuristic_opponent`
`tests/evaluation/test_evaluation_callback_integration.py`	`test_periodic_evaluation_trigger`, `test_evaluation_r`

Test File	Contained Tests (Functions)
`tests/test_wandb_integration.py`	`test_wandb_logging_off_noop`, `test_wandb_logging_on_`
`tests/test_integration_smoke.py` *(if exists)*	`test_full_training_episode` or similar

Notes: The above is derived from file names and partial content. Actual function names might differ slightly, but the essence is captured. The test suite shows very high coverage: all critical components (game rules, agent logic, managers, utils) have direct tests. Integration tests exist to ensure the system works as a whole.

C. Feature-to-Module Traceability Matrix

This matrix maps high-level user-facing features of *Keisei* to the primary code components that implement them:

Feature / Capability	Primary Code Components
Complete Shogi Rules Enforcement	*Shogi engine modules:* `keisei.shogi.shogi_game` (game state, move application) [100] [109], `keisei.shogi.shogi_rules_logic` (move generation and rule validation) [125] [7], `keisei.shogi.shogi_move_execution` (piece movement, capture, promotion), `keisei.shogi.shogi_core_definitions` (piece types, constants). These work together to ensure all standard rules (drops, promotions, repetition, check/checkmate) are implemented.

Feature / Capability	Primary Code Components
Self-Play Training with PPO	`keisei.core.ppo_agent.PPOAgent` (PPO algorithm: action selection, learning updates) [90] [94], `keisei.core.experience_buffer.ExperienceBuffer` (storage of gameplay data and advantage calculation), `keisei.training.trainer.Trainer` & `keisei.training.training_loop_manager` (or StepManager) for orchestrating environment-agent interaction and calling learning steps, `keisei.training.env_manager.EnvManager` (resets and prepares environment for each episode) [56]. Together, these manage continuous self-play and periodic policy optimization.
Parallel Distributed Self-Play	`keisei.training.parallel.parallel_manager.ParallelManager` (spawning multiple self-play workers), `keisei.training.parallel.self_play_worker` (process that runs games and collects experiences), `keisei.training.parallel.communication` & `model_sync` (tools to sync data and model parameters across processes). These components implement the ability to collect experience from multiple games in parallel to speed up training (configurable via `config.parallel.*`).
Neural Network Flexibility (CNN/ResNet)	`keisei.training.models` package – e.g., `models/resnet_tower.py` (defines ResNet architecture with Squeeze-and-Excitation blocks) and possibly a basic model in `neural_network.py`. `ModelManager` uses `config.training.model_type` to choose which network to instantiate. The Trainer/SetupManager creates the network accordingly. The `ActorCriticProtocol` ensures both architectures present a unified interface (get_action_and_value).
Mixed Precision Training	Integration of PyTorch AMP: `PPOAgent` accepts a `use_mixed_precision` flag and can utilize `torch.cuda.amp.GradScaler` [85] [210]. Within `PPOAgent.learn`, it would wrap forward/backward passes in `autocast` if enabled and scale gradients. The config flag `training.mixed_precision` triggers this. Also, the model creation and optimizer are compatible with AMP.
Distributed Training (Multi-GPU/DDP)	The `--ddp` flag in `train.py` and `config.training.distributed` hook into PyTorch's `DistributedDataParallel`. While the audit didn't deeply see DDP code, likely if enabled, `Trainer` or `SetupManager` would wrap the model in `torch.nn.parallel.DistributedDataParallel`. `multiprocessing.set_start_method('spawn')` is done in train.py specifically to support DDP on Windows [211]. The design anticipates multi-GPU training, even if not fully tested.

Feature / Capability	Primary Code Components
Periodic Evaluation & Opponents	`keisei.evaluation.EvaluationManager` (coordinates evaluation matches on a schedule), `keisei.utils.opponents` (contains opponent implementations like `RandomOpponent`, potentially a `PreviousModelOpponent` drawing from a model pool). `CallbackManager` triggers evaluations during training (e.g., every N timesteps or end of epoch as per config) [212]. Evaluation games themselves reuse `ShogiGame` and possibly spin up separate agents or use the main agent against a fixed opponent. Results are recorded in MetricsManager (win rates, etc.).
Live Training Monitoring (TUI)	`keisei.training.display_manager.DisplayManager` and `keisei.training.display` (TrainingDisplay) implement the Rich text User Interface [167]. They construct a live dashboard with panels: the game board (via `ShogiBoard` component), recent moves (`RecentMovesPanel`), piece captures (`PieceStandPanel`), a progress bar and statistics (`Progress` from rich, using custom columns fed by MetricsManager data) [172] [173], and trend charts (`Sparkline` for metrics like move rate). These update in real-time during training via `DisplayManager.setup_display` and periodic calls to refresh. Logging is integrated so that console logs appear in a log panel.
Comprehensive Logging	`keisei.utils.unified_logger.UnifiedLogger` and `TrainingLogger` provide timestamped logging to both file and console [34] [187]. All major events (errors, warnings, info) use these (Trainer passes `logger.log` into managers so they can use it). Additionally, metrics are logged: MetricsManager or EvaluationManager might print summary of results each epoch. W&B integration also ensures these logs and metrics go to an online dashboard if enabled.
Weights & Biases Integration	`SessionManager.setup_wandb()` reads `config.wandb` and initializes the wandb run (project name, etc.) [213]. Throughout training, callbacks or logger might call `wandb.log` with metrics. The code ensures that if `wandb.enabled=false`, no calls are made. For example, `trainer.is_train_wandb_active` is passed to EvaluationManager [151] so it might conditionally log ELO or results. The design cleanly separates concerns: if W&B is enabled, experiment metadata and metrics are automatically recorded; if not, the system still logs everything locally.
Configuration Management & CLI Overrides	`keisei.utils.utils.load_config` loads `default_config.yaml`, merges any user config file and CLI `--override key=value` pairs [42] [82]. The Pydantic `AppConfig` ensures type correctness and default values. This means any feature's configuration (training hyperparams, evaluation frequency, etc.) can be adjusted via config without code changes. For example, *feature:* "turn off UI" is done by setting `display.enable_board_display=false` in config, which flows into `TrainingDisplay` toggling those panels [170]. Another example: "use a smaller network" can be done by config (e.g., `training.tower_depth`) which is consumed by ModelManager/Model constructor.

Feature / Capability	Primary Code Components
CI/CD Pipeline & Quality Assurance	While not a runtime feature, the project includes a GitHub Actions config (documented in `docs/CI_CD.md` [48]) and `scripts/run_local_ci.sh` [11]. These orchestrate running of tests (`pytest`), style checks (`flake8`, `black`, `isort` via pre-commit), type checks (`mypy`), and security scans (`bandit`, `safety`). The presence of `.pylintrc`, `.flake8`, `.pre-commit-config.yaml` [214] [215] in the repo shows these tools are configured. This ensures features are continuously verified. For example, the **feature** "Quality gates" is implemented via these CI jobs that must pass before merging, preventing regression of any core functionality.

D. Architecture Decision Record (ADR) Backlog

During the audit, a few architectural questions emerged which could be formalized as decisions to make. Below is an ADR backlog of such decisions, including context and options:

Decision Needed	Context & Problem	Options (Pros/Cons)
ADR-1: Simplify Manager Architecture or Retain 9 Managers?	The project uses 9 specialized managers for clear separation [3]. This yields modularity but at cost of complexity. New developers might find it verbose to navigate.	**Option A: Retain 9 Managers (Status Quo)** – (Pros: clear responsibilities, easier testing of components [4]; Cons: more boilerplate, higher conceptual overhead). **Option B: Merge Some Managers** – e.g., combine TrainingLoopManager + StepManager, or DisplayManager + MetricsManager if their roles overlap. (Pros: fewer classes to understand, possibly fewer cross-references; Cons: larger classes, potential loss of modular clarity, harder to test in isolation). **Option C: Introduce a Facade** – Keep managers internally but present a simpler interface in Trainer (like Trainer methods that internally call managers). (Pros: outside code interacts with one class; Cons: adds another layer, might not reduce cognitive load by much).
ADR-2: Handling of Exceptions – Fail-Fast vs Resilient?	Current design prefers to catch errors, log, and continue with defaults (for config, seeding, etc.) [22]. This ensures training doesn't stop for minor issues, aligning with a long-running training job mindset. But it risks masking problems.	**Option A: Keep Resilient Strategy** – (Pros: training uninterrupted by non-critical issues, good for long jobs; Cons: silent errors could lead to bad results unknowingly). **Option B: Fail-Fast on Exceptions** – e.g., remove broad catches or rethrow after logging. (Pros: ensures any misconfiguration or unexpected issue halts run for immediate fix; Cons: could stop long training for a minor issue like a single failed metric log). **Option C: Configurable Strict Mode** – a compromise: run in strict mode during development (exceptions propagate) and resilient mode in production runs. (Pros: best of both; Cons: added complexity in code paths/testing).

Decision Needed	Context & Problem	Options (Pros/Cons)
ADR-3: Parallel Self-Play Implementation	The parallel experience collection is partially implemented. Decision on how to fully integrate it is needed. Should training always use parallel if multiple CPUs available? How to handle synchronization?	**Option A: Shared Memory & Multi-threading** – Instead of multi-process, possibly use Python threads with GIL released during game play (since much of game is Python, GIL might hinder true parallelism, so threads likely not beneficial; thus probably not ideal). **Option B: Multiprocessing Queues (current approach)** – Each worker process runs games, puts experiences into a Queue; trainer process gathers them. (Pros: isolates game simulation, can leverage multiple cores; Cons: more overhead in IPC, complexity in ensuring all processes have updated model weights timely). **Option C: Async Vectorized Environment** – e.g., use something like OpenAI Gym's vector env or a custom event loop to handle multiple games asynchronously in one process. (Pros: maybe simpler than multi-proc, no pickling of data; Cons: more complex logic to interleave game steps, and Python async might not truly parallelize CPU-bound tasks).
ADR-4: Model Checkpoint Format & Frequency	Currently, ModelManager likely saves PyTorch `.pt` files periodically. The strategy for checkpointing needs to be decided: keep only the latest? keep all (could be many files)? Also, whether to use state dicts vs full pickled objects.	**Option A: Save Latest N Checkpoints** (e.g., rotate the files) – (Pros: limits disk usage, still have some backups; Cons: could lose older models needed for evaluation reference or rollback far). **Option B: Save All Checkpoints** – (Pros: complete history for analysis or if wanting to pick best after the fact; Cons: potential large storage usage for long runs). **Option C: Adaptive Checkpointing** – save more frequently early in training (when things change fast), then less frequently later, or only on significant improvements (need a metric). (Pros: efficient storage; Cons: more complex logic, risk missing some states). *Format:* (regardless of above) – **State Dict vs Full Model:** Prefer state dict for stability (not tied to class code structure, safer to load between versions). Probably use `torch.save(model.state_dict())`. Ensure the checkpoint strategy is documented for users.

Decision Needed	Context & Problem	Options (Pros/Cons)
ADR-5: Deployment & Packaging Strategy	As the project evolves, consider how end-users will consume it: via source, pip install, or container. Also how to deploy training runs (locally vs cloud). The pyproject indicates a package structure. Decision: focus solely on source code release, or provide pip package, or docker images?	**Option A: Publish on PyPI** – (Pros: easy install of library, can version releases; Cons: training typically involves code customization, packaging a constantly evolving research code might be overhead, also W&B and data dependencies might complicate). **Option B: Docker Container** – Provide a Dockerfile for a ready-to-run environment with all deps including GPU support. (Pros: reproducible runs, easier for users to get started; Cons: requires maintaining docker config, users need Docker setup). **Option C: Script/Repo Distribution Only** – (Pros: simplest, user clones repo and runs, which is common in research; Cons: less polish, environment setup left to user aside from requirements files).
ADR-6: Future Proofing AI Code Contributions	Since code was AI-generated, establishing guidelines for future contributions is key (to maintain consistency). Decision needed on whether to continue using AI tools extensively or shift to human-only for critical logic.	**Option A: Continue AI-First Development** – e.g., use Copilot or similar for new modules, but with rigorous review and testing as currently done. (Pros: maintain development velocity and consistency in style; Cons: potential for subtle bugs if AI suggests incorrect logic, reliance on strong test coverage to catch issues). **Option B: Human-Only for Critical Sections** – e.g., manually write complex logic like game rules or learning algorithms, using AI only for boilerplate (tests, docs). (Pros: human insight on complex parts, potentially fewer logic errors; Cons: slower development, might introduce style inconsistencies). **Option C: Blend with Care** – formalize a guideline: AI proposals are fine but must be reviewed line-by-line, and any AI-introduced section should ideally be covered by tests. This is essentially current practice, so likely the chosen path. Document this to ensure future devs understand the code style and quality expectations.

Each ADR can be fleshed out and decided upon by the project maintainers. For now, they serve as reminders of areas requiring strategic decisions.

E. Full Issue Inventory

All findings from the audit (including minor ones) are catalogued below, with a short description and recommended action, cross-referenced to where they appear in code:

Issue/Finding	Severity	Location (Ref)	Description & Recommendation
Broad `except Exception` in training startup	Low	`training/train.py` [1] [64]; `training/env_manager.py` [15]; `core/ppo_agent.py` [22]	Various initialization rout catch all exceptions, riski hidden errors. These sho be narrowed or made opt *Recommend:* narrow to sp exceptions or use a debu mode to rethrow. [1] [22]
Silent environment seeding failure	Low	`EnvManager.setup_environment` [65]	If `ShogiGame.seed()` f (or if ShogiGame has no s method), it logs a warnin continues unseeded. *Recommend:* make it clear the user (promote warnin higher-level log) or enfor game seeding method.
Fallback to default LR on optimizer init	Low	`PPOAgent.__init__` [22]	If an invalid learning rate provided, it logs error an uses 1e-3. This might mas config typos. *Recommend* log an info about using default, and consider faili LR was set to a nonsense unless user explicitly wan auto-fix.
Lack of explicit error on no legal moves	Low	`PPOAgent.select_action` [92]	When `legal_mask` has valid moves, it logs an err and still calls model (whic might handle it). The desi assumes caller ensures legality. *Recommend:* Poss have `select_action` r a None or raise an except to force caller to handle n move scenario. Currently an issue due to game rule (should never happen ou terminal state).

Issue/Finding	Severity	Location (Ref)	Description & Recommendation
ShogiGame complexity (too many responsibilities)	Moderate	`ShogiGame` (1096 lines) [16]	ShogiGame handles state move execution, history, a some rule checking. It's la but logically partitioned. *Recommend:* if maintainab suffers, break out some lo (e.g., move application to helper, or repetition check separate class). For now, heavily rely on tests when modifying.
Redundant move legality check in `make_move`	None (design)	`ShogiGame.make_move` [23]	The code double-checks r patterns even though leg moves are presumably checked earlier. This redundancy is intentional safety. *No action needed.* (performance becomes an issue, could remove, but negligible impact so safer keep).
Hard-coded limits (max moves=512 in some places)	Very Low	e.g., `ShogiGame.__init__` default 500, tests use 512, `constants.MOVE_COUNT_NORMALIZATION_FACTOR=512` [6]	The number of max move game is set to 512 in tests constant. It's arguable if t should be configurable vi config (it is via `env.max_moves_per_g` It's fine as is; just note tha changing it requires adjus normalization factor. *No immediate action.*
Potential performance issue with Rich live updating	Low	`training/display.py` weight stats calc [29], frequent console updates	Frequent computation of model stats and rich rend can slow down training slightly. *Recommend:* mon via profiling. If slowdown observed, reduce update frequency or allow toggli heavy UI features (already somewhat done via confi

Issue/Finding	Severity	Location (Ref)	Description & Recommendation
Unused variables or artifacts from AI generation	Low	e.g., Some `# ADDED` comments, unused imports logged by flake8 (if any)	During audit, no glaring c code found. Tools like flak likely cleaned them. Just k an eye out in code review any leftover generated co that's not needed. *Action:* specific, just continuous cleanup.
Test suite maintenance	Low	(General)	Tests are very thorough. A code evolves (especially if refactoring managers or logic), tests will need upd Some tests are tightly cou to implementation (e.g., checking specific strings internal counters). *Recommendation:* Maintai tests in tandem with code changes; use them as gui for correct behavior, upda expected values only whe intentional behavior char Possibly add tests for new scenarios (like multi-GPU implemented).
Documentation minor inaccuracies	Low	e.g., README might drift from code	Ensure that documentatio (like README's described command-line flags or co options) stays updated w code. E.g., if new CLI args added or defaults change reflect it. *Action:* periodic review, maybe add a test loads README code snipp possible (not critical).
WandB usage on by default	Low	`default_config.yaml` has `wandb.enabled: true` [213]	While great for tracking, s users without WandB acc might hit an API prompt. code handles `.env` for A key. *Recommendation:* Co defaulting to false for ope source release, or clearly instruct how to disable. N code bug, but a UX consideration.

All the above issues are relatively minor and manageable. The table can serve as a checklist for maintainers to address or intentionally accept each item.

F. Manual Dependency Review

A review of external dependencies (from `pyproject.toml` and `requirements.txt`) was conducted to identify any outdated or potentially problematic libraries:

- **Python Version:** Requires Python >= 3.12 [216], which is the cutting edge. This ensures use of latest language features (pattern matching, improved performance) but may limit some users on older versions. Given 3.12 is now released (2025), this is forward-looking. *No issue.* Ensure CI tests on 3.12 and maybe 3.11 for wider compatibility.

- **PyTorch (torch >= 2.0.0)** – Latest PyTorch (2.0 was a major update in 2023). Torch 2.0 is stable, and 2.1+ likely available; the project should monitor PyTorch releases for compatibility (e.g., 2.1 might introduce minor API changes or performance improvements). *No known vulnerabilities.* Just note to update to 2.x latest in future for performance.

- **numpy >= 1.24.0** – Fairly recent (1.24 came out late 2022). Current version is around 1.25.x (mid-2025). Minor version bumps are usually fine. *No security issues.* Keep updated for performance/bug fixes.

- **python-dotenv >= 1.0.0** – Used for loading .env (WANDB API key). Version 1.0.0 is recent (2022); current might be ~1.0.3. *No issues.* Only loaded in dev or when needed.

- **wandb >= 0.17.0** – Weights & Biases client. 0.17 was around 2023. Current might be 0.18 or 0.19. W&B is active; older versions sometimes had issues (like large artifact handling). Check compatibility if updating. Also note W&B can pull in a lot of sub-dependencies (like `sentry-sdk`, `GitPython`). The project pins pydantic explicitly likely because wandb might bring an older version by default [46]. *No direct security issues known.* Just be mindful: if not using W&B, it's optional. If using, keep up with updates to avoid deprecated API issues.

- **pydantic >= 2.11,<3.0** – Pydantic v2 is used. 2.11 is quite new (likely project keeps updating). Pydantic 2 is a major rewrite from v1, it's good to pin <3.0 since v3 will be far out. *No issues.* Only ensure compatibility with code (which seems fine). Pydantic 2.x had some minor version bugfixes, but 2.11 is near latest.

- **rich >= 13.0.0** – Rich library for console UI. Latest as of 2023 was around 13.x, might have 14.x by 2025. No major security concerns; mostly feature additions. The pinned version is fine. Check that API calls used (Layout, Live, etc.) haven't changed in newer versions.

- **PyYAML >= 6.0** – Latest major (6.0) is good (addresses some older vulnerabilities in PyYAML 5.x by using safe_load, etc.). The code uses `yaml.safe_load` explicitly [193], so that is secure for loading config. *No issues.* When updating PyYAML, just ensure no API break (unlikely; 6.x is stable).

- **wcwidth >= 0.2.5** – This is used by Rich for rendering width calculations. It's a small lib, v0.2.5 is fine. Not much to worry.

- **Dev Dependencies:**

- **pytest >= 8.0.0** – Current as of 2023. Upgrading to 8.x from 7.x introduced some minor changes (like import mode improvements). The tests appear to target new features (like pytest-cov 5, etc.). Should be fine.
- **pytest-cov >= 5.0.0** – For coverage, no issues.
- **pytest-pylint >= 0.21.0** – Runs Pylint during tests. Pylint is configured to ignore some things (like redefined-outer-name in fixtures) [217] . Ensure that the pinned Pylint (3.0.0 as per dev deps) is compatible with Python 3.12 (Pylint 3 is, as it's relatively new).
- **black >= 24.0.0** – Black 23.x was stable, 24.0.0 is upcoming (possibly they anticipate a new release). Black ensures code style; no problem. Just stick to one version to avoid churn in formatting diffs.
- **isort >= 5.13.0** – Ok.
- **flake8 >= 7.0.0** – Flake8 6 was current in 2023, 7 might be a future release. Pinning to >=7 might cause dev env to fetch a pre-release if not careful, but likely fine. Keep an eye because flake8 plugins might not all support 7 immediately.
- **pylint >= 3.0.0** – Pylint 3 supports Python 3.11+. Good. It's included likely for `pytest-pylint`. Already configured to ignore specific things to reduce noise [218] .
- **mypy >= 1.9.0** – Static typing, good. Keep updated as needed.
- **types-PyYAML >= 6.0.0** – Stub for PyYAML for mypy.

- **deptry >= 0.12.0** – This is interesting: deptry checks for dependency issues (unused or missing dependencies). It shows the project's thoroughness in dependency management. Running deptry periodically will catch if any imported module isn't listed or vice versa. Keep this; no issues.

- **Potential Vulnerabilities:** A quick scan for any historically vulnerable packages:

- `sentry-sdk` (via wandb) had an advisory but wandb pins a safe version typically.
- `GitPython` (via wandb) had some performance issues but not security.
- Nothing in our requirements stands out as risky (no web frameworks, no known CVEs for these).

- Tools like Safety in CI would flag if any known vulns appear.

- **Outdated Dependencies:** None appear significantly outdated as of mid-2025. All are quite modern. Just plan to bump them periodically:

- E.g., wandb might be at 0.20 by 2025 – test and upgrade to get new features and bug fixes.
- PyTorch 2.1 or 2.2 might give performance improvements – test compatibility (should be fine if no C++ custom ops used).
- Ensure Rich is updated if any issues (rich is backward compatible mostly).
- Pydantic 2.12 or 2.13 may come; follow their release notes (since Pydantic v2 is new, minor versions can change behavior slightly).

In summary, dependencies are in good shape. The project proactively included dev tools to ensure quality (deptry for cleanup, Safety in CI to catch vulns, etc. as per CI plan [77]). Continue this practice. For each new dependency added, consider security (e.g., if adding a library for a new feature, run it by safety and check maintenance status).

G. Identified Complexity Hotspots

The following files or functions stand out due to their size or complexity and may warrant special attention for future refactoring or optimization:

- `keisei/shogi/shogi_game.py` – ~1096 lines [16]. *Hotspot:* This is by far the largest single module. It handles many tasks from board setup to move enforcement to game termination. Complexity arises from branching logic for different move types and game phases. While it's well-structured given the domain complexity, it's a candidate for refactoring into smaller modules (e.g., separate move application, rule checking, and maybe move generation entirely, although move gen is already partly separate). A future refactor might split this into `shogi_game.py` (state + basic moves), `rules.py` (all rule enforcement like checkmate detection, nifu, etc.), and possibly `shogi_game_history.py` (for repetition logic). Right now, the risk is mostly cognitive – any bug here is likely caught by tests.

- `keisei/core/ppo_agent.py` – ~533 lines [219]. *Hotspot:* The PPOAgent class includes not just a simple agent but also gradient update logic (though not fully shown in snippet, presumably in the latter half of the file). PPO involves multiple steps (compute advantages, loop over epochs, etc.), which can make the `learn()` method long and complex. Ensuring each sub-calculation (advantage normalization, gradient clipping, KL divergence calculation) is correct and efficient is vital. This function is also performance-sensitive. If it becomes a bottleneck, consider vectorizing some operations or using PyTorch's optimized ops. For maintainability, if `learn()` is too large, break it into helper functions (e.g., `calculate_advantages`, `compute_loss`, etc. inside the class). Currently, tests like `test_learn_basic` give confidence in its correctness, but keep it on radar.

- `keisei/training/display.py` – ~600 lines [220]. *Hotspot:* This handles dynamic UI. Complexity comes from dealing with layout and multiple components, and conditional features (enable/disable panels). The logic to update weight stats and trends has many moving parts. If an issue arises in the UI (like misaligned output or flicker), debugging this could be tricky due to interactive nature. It's fairly separate from training logic, so it doesn't risk core correctness, but is a complexity hotspot for the UI/UX domain. A potential improvement later is to simplify or modularize the UI updates (maybe have each panel manage its own update function). As of now, it's manageable.

- `keisei/training/trainer.py` **&** `setup_manager.py` – Combined, they orchestrate everything with ~400+ lines in trainer and ~100+ in setup. *Hotspot:* Not in terms of algorithmic complexity, but in terms of being critical glue code. Many components intersect here, and a mistake in initialization order could break things. They are well-commented and logically stepwise, which helps. The complexity is that any change in one manager's requirements might require changes here. So one must be careful when modifying the trainer init flow. The audit suggests possibly simplifying by merging some steps. Keeping an eye on this hotspot ensures the system boots up correctly.

- `keisei.training.parallel/*` – These modules might not be large individually, but parallel programming inherently adds complexity (synchronization, potential for deadlocks or race conditions). While not fully implemented, when it is, consider it a logical complexity hotspot. Testing parallel execution is also more complex (some issues only show under heavy load or specific timings). So, treat this as an area requiring careful design and possibly use of proven libraries (like Ray or others) if needed.

- **Inter-module complexity:** Not a single file, but the interplay between modules is complex. For example, the training loop touches nearly every subsystem (game, agent, buffer, metrics, UI, eval). The complexity is mitigated by clear interfaces, but integration testing (like the smoke test) is crucial. If something fails, one must understand interactions. This is more an architectural complexity than code complexity but bears noting.

To manage these hotspots: - Use profiling (the built-in profiler and the provided `profile_training.py`) to see if any hotspot is also a performance bottleneck. - Schedule periodic refactoring when tests are green, to gradually reduce complexity in these areas (don't let them grow unbounded). - Ensure new contributors are aware that modifications in these hotspots require running the full test suite (including integration tests) to catch any ripple effects. - Potentially add comments or docs specifically in these modules to guide maintainers (e.g., an overview comment in `shogi_game.py` summarizing its structure, to ease navigation).

H. Execution Flow Diagrams

To visualize the system's operation, below are simplified flow outlines for two main processes: the **training loop** and the **evaluation cycle**. They are described in stepwise form (which can be converted to a diagram if needed):

Training Loop Flow (Self-Play with PPO):

1. **Initialization:**
2. Load config (YAML + overrides) -> `AppConfig` [82].
3. `Trainer` created: it initializes all managers (Session, Env, Model, etc.) [221] [141].
4. `SetupManager` builds the environment (new `ShogiGame`), the agent (`PPOAgent` with policy network), and buffer [147] [148].
5. Initial game state observation obtained via `game.reset()` [73]. Trainer ready to start.
6. **Begin Training Loop:**
7. `total_steps = 0`.
8. **Epoch Loop:** while `total_steps < config.training.total_timesteps`:
    1. **Episode Loop:** while not `experience_buffer` full (i.e., < `steps_per_epoch`):
        - If new episode or game over, reset environment: `env_manager.reset_game()` to fresh state [119].
        - Initialize episode variables (episode reward, length).
        - **Step Loop:** while game not done (and buffer not full):
        - Get current state `obs` (from `game.get_observation()`).
        - Compute `legal_mask` = `policy_output_mapper.get_legal_mask(game.get_legal_moves())`.
        - **Agent Action:** `move, idx, logp, value = agent.select_action(obs, legal_mask, is_training=True)` [83].
        - **Environment Step:** `next_obs, reward, done, info = game.make_move(move)` – game state updates [111].
        - `experience_buffer.add(obs, idx, reward, done, value, logp)`.
        - Update episode cumulative reward, length, etc.

        - If done (game over):
            - If done not due to max_moves, get final value = 0 (no next state value because terminal).
            - Else if done due to max_moves (draw), perhaps treat final value = baseline or 0.
            - Break step loop.
        - If buffer is full (steps_per_epoch reached), break episode loop even if game isn't done (episode will continue next epoch – typically in PPO we might end episode early, but since self-play games can be long, there's a design choice: either cut off or let episode finish and carry remainder. The code likely allows breaking mid-episode and will resume next epoch, or it chooses steps_per_epoch such that it aligns with episode lengths).
    2. **End of Epoch:** Now `steps_per_epoch` transitions gathered:
        - If last episode ended, finish calculation for that episode. Otherwise, we stopped mid-episode:
        - Compute GAE advantages for whatever portion of episode collected. For that, need a `last_value` (if game not done, use agent.get_value(current_state) to approximate remaining value).
        - Call `experience_buffer.finish_episode(last_value)` to compute advantages and returns.
        - **Policy Update:** `agent.learn(experience_buffer)` – performs PPO update with multiple minibatch iterations [94] . This updates model parameters.
        - Reset experience_buffer for next epoch.
        - Increase `total_steps += steps_per_epoch`.
        - **Metrics Logging:** e.g., compute average reward/length from episodes in this epoch, update `metrics_manager` (which can compute running mean, etc.).
        - **Checkpoint:** if time to checkpoint (e.g., every N epochs or timesteps), call `model_manager.save_checkpoint(agent.model, ... )`.
        - **Evaluation Trigger:** if `total_steps` meets `evaluation_interval`, call `evaluation_manager.run_evaluation(current_agent)` via a callback.
        - **Display Update:** refresh the live TUI: update progress bar (with steps done out of total, maybe percent), update trend graphs (e.g., recent win rates or losses), update board (could show last game state), etc. [29] [222] .
        - If using W&B, log metrics (episode reward mean, loss values, etc.).
    3. Loop back for next epoch.
9. End while when `total_steps >= total_timesteps`.
10. Wrap up: possibly run one final evaluation, save final model, close resources (e.g., file handles, wandb run).
11. Print or log training completion message with final stats.

Note: The above is aligned with PPO algorithm structure. The actual code structure might vary slightly (they might consider each epoch as one "policy update cycle" irrespective of episodes, etc.), but this is the general flow described in README [212] .

Evaluation Cycle Flow (Periodic Evaluation of Current Model):

(This occurs during training whenever triggered, or after training.)

1. **Setup Evaluation:** EvaluationManager is configured with:

2. strategy (from config: e.g., "self_play" or "versus_baseline"),
3. `num_games` to play,
4. a list of opponents:
    - If `opponent_type` is "random", use `RandomOpponent`.
    - If "self" or similar, it might just play the current agent against itself.
    - If a specific older model name given, load that model as an opponent.
5. If `enable_parallel_execution` true, possibly prepare to run games in parallel (threads or processes).
6. Ensure environment(s) available (it might reuse `ShogiGame` or create new ones to run games concurrently).

7. If evaluating ELO, load or initialize ratings (likely, current model starts at config.display.elo_initial_rating).

8. **Run Evaluation Games:** For i in 1.. `num_games`:

9. Initialize a new game (`ShogiGame.reset()`) for each game.
10. Determine players: Usually, one side is current trained agent, the other is opponent (which could be a fixed policy or another copy of agent if self-play).
11. Play out the game:
    - While not game_over:
    - If it's current agent's turn, use current `PPOAgent` (in evaluation mode, deterministic policy) to choose move.
    - If opponent's turn:
        - If opponent is a simple AI (Random or HeuristicOpponent), use its `select_move(game)` to pick a move.
        - If opponent is another neural agent (like previous model), call its policy (likely also a PPOAgent or a stripped-down policy network) deterministically.
    - Execute move via `game.make_move()`.
    - End loop on game_over.
12. Record result: win/loss for current agent or draw.
13. If requested, save game record (e.g., append moves to a list or file).
14. Possibly, if `enable_in_memory_evaluation` false, the EvaluationManager might spawn each game in a separate process to avoid interference (especially if wanting parallel games).

15. Repeat until all games done (could be sequential or in parallel batches).

16. **Aggregate Results:**

17. Compute win rate = wins / num_games (for current agent).
18. If multiple opponent types, compute win rate per opponent type.
19. If using ELO:
    - Update ELO ratings: e.g., if current agent won most games against baseline (which has fixed rating), increase current's rating accordingly using config.elo_k_factor.
    - If playing self-play (current vs itself for robustness), ELO may not apply, but if playing vs previous models, update both accordingly.

20. Prepare a metrics dict: e.g., `{"eval_win_rate": 0.65, "eval_num_games": 20, "elo": 1350}`.

21. **Log & Save:**

22. Log the evaluation results via TrainingLogger (console and file): e.g., "Evaluation: win_rate=65%, draws=5%, vs RandomOpp".
23. If wandb active, `wandb.log` the metrics (possibly with step = current training timesteps).
24. If `update_elo=True`, and an ELO registry path is given, save updated ELO ratings to a file (to persist across runs).
25. If `save_games=True`, write game records to a file or folder (like PGN or custom format for later replay or debugging).
26. If `previous_model_pool_size > 0`:
    - Possibly add current model weights to a pool (maybe save to disk or keep in memory) for future evals.
    - If pool exceeds size, remove oldest.
27. Return the metrics (so CallbackManager/Trainer can also use them, e.g., to decide "if win_rate > X, save as best model").
28. **Continue Training:** The training loop receives control back after evaluation and can use the info (maybe adjust training if needed, or just log it).

This evaluation is automatically triggered at configured intervals. There's also a scenario of a final evaluation after training, which would follow the same steps but outside the training loop.

If multiple evaluation strategies (like both self-play evaluation to measure training progress and external baseline evaluation), EvaluationManager or CallbackManager might manage multiple schedules.

I. Proposed CI Configuration (GitHub Actions YAML)

The following is a **proposed GitHub Actions workflow** (`ci.yml`) that implements the CI pipeline described in documentation [223] [9] [77]. It runs tests across multiple Python versions, performs linting, type checking, coverage, and security scans. This configuration is designed to be placed in `.github/workflows/ci.yml`:

```
name: CI

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

jobs:
  test:
    name: Test (Py${{ matrix.python-version }}) and Lint
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [ "3.9", "3.10", "3.11", "3.12" ]
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python
```

```
        uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: |
          pip install -r requirements-dev.txt
          pip install -r requirements.txt
      - name: Lint with flake8
        run: flake8 .
      - name: Type-check with mypy
        run: mypy keisei tests
      - name: Run tests with coverage
        env:
          PYTHONWARNINGS: "ignore"
# to suppress any deprecation warnings in output
        run: pytest --cov=keisei --cov=tests --cov-report=xml --maxfail=1 -q
      - name: Upload coverage to Codecov
        if: ${{ matrix.python-version == '3.11' }}  # only once to avoid
duplicates
        uses: codecov/codecov-action@v3
        with:
          file: ./coverage.xml
          flags: unittests
          name: codecov-coverage
          fail_ci_if_error: true

  integration-test:
    name: Integration Smoke Test (Py3.11)
    runs-on: ubuntu-latest
    if: github.event_name == 'push' && startsWith(github.ref, 'refs/
heads/')  # only on pushes to main/develop
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: "3.11"
      - name: Install dependencies
        run: |
          pip install -r requirements-dev.txt
          pip install -r requirements.txt
      - name: Run integration tests
        # This assumes there is an integration test file or command
        run: pytest -q tests/integration/ test_phase1_validation.py

  parallel-test:
    name: Parallelism Smoke Test (Py3.11)
    runs-on: ubuntu-latest
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
```

```
        with:
          python-version: "3.11"
      - name: Install dependencies
        run: |
          pip install -r requirements-dev.txt
          pip install -r requirements.txt
      - name: Run parallel system test
        # Hypothetical command or test file to verify multiprocessing aspects
        run: pytest -q tests/parallel/ test_parallel_smoke.py

  performance-check:
    name: Performance Profiling (Py3.11)
    runs-on: ubuntu-latest
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: "3.11"
      - name: Install dependencies
        run: |
          pip install -r requirements.txt
      - name: Run short profiling session
        run: |
          python scripts/profile_training.py --timesteps 500 --report
      - name: Upload profiling artifacts
        if: always()
        uses: actions/upload-artifact@v3
        with:
          name: profiling-report
          path: profile_report*.txt  # assuming the script outputs some
report files

  security-scan:
    name: Security Scan
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: "3.11"
      - name: Install security tools
        run: pip install bandit safety
      - name: Run Bandit (Python security linter)
        run: bandit -q -r keisei -ll  # run bandit with high severity/
confidence filters
      - name: Check dependencies with Safety
        run: safety check -r requirements.txt -r requirements-dev.txt
```

Explanation:
- We use a **matrix** for the main `test` job to run on Python 3.9 through 3.12, ensuring broad

compatibility [9]. This job installs all deps (dev+prod), then runs flake8 (lint) [224], mypy (types), and pytest with coverage. The coverage is output as XML and uploaded to Codecov (only once to avoid duplicate reports). The `--maxfail=1 -q` options in pytest ensure we fail fast on first failure and keep output terse. - The **integration-test** job runs only on pushes (not on every PR necessarily) on one Python version (3.11). It would run any longer-running or integrated scenarios, possibly including the provided `test_phase1_validation.py` or others that simulate a short training run. This catches issues that unit tests might not (like all pieces working together). - The **parallel-test** job triggers only on main branch pushes (since parallel tests might be heavier or require specific environment) [225]. It runs any tests related to parallel self-play to ensure no deadlocks or multi-process issues (likely a smoke test that starts ParallelManager with a dummy scenario). - The **performance-check** job runs also on main branch pushes (to avoid burdening PRs). It runs the `profile_training.py` for a short session to generate performance metrics. The artifact (profile report or data) is uploaded so maintainers can inspect performance regressions offline. - The **security-scan** job runs Bandit and Safety on every push/PR (quick checks) [77]. Bandit scans the code for common vulnerabilities (should pass, as code avoids dangerous functions). Safety checks for known vulnerable deps in requirements; given current deps, likely none, but this will catch e.g., if a future version of a lib has a security issue.

Additionally, a **release** workflow (as mentioned in docs [80]) can be set up to trigger on tags for building and publishing the package. That could be in a separate `release.yml`, performing packaging (wheel, sdist), twine check, and pushing to PyPI, as well as creating a GitHub Release with changelog. Since not explicitly requested in audit deliverables, it's just noted here.

This CI config ensures that: - Code style and static analysis issues are caught early. - The test suite is reliable across versions. - Performance and parallel features are not broken by changes. - Security is continuously monitored.

The workflow is aligned with the project's ops plan and can be adjusted as needed (e.g., adding caching for pip to speed up installs, etc., which was omitted for clarity).

---

[1] [17] [21] [64] [66] [196] [201] [211] train.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/training/train.py

[2] [16] [23] [24] [38] [39] [50] [51] [68] [69] [70] [71] [72] [73] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [111] [112] [113] [114] [115] shogi_game.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/shogi/shogi_game.py

[3] [4] [5] [20] [40] [41] [49] [74] [166] [212] [213] README.md
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/README.md

[6] [33] [117] constants.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/constants.py

[7] [8] [61] [62] [75] [132] [133] [134] [135] test_shogi_rules_and_validation.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/tests/test_shogi_rules_and_validation.py

[9] [10] [11] [12] [48] [63] [77] [78] [79] [80] [81] [202] [223] [224] [225] CI_CD.md
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/docs/CI_CD.md

13 47 214 215 GitHub - tachyon-beep/shogidrl: A Deep Reinforcement Learning project demonstrating AI's power to create AI, aimed at mastering the complex game of Shogi. Built 100% by GitHub Copilot (Agent Mode) with human project management, it features a custom Shogi engine, PPO in PyTorch, rich experiment tracking via Weights & Biases, and a live TUI using Rich for dynamic monitoring.
https://github.com/tachyon-beep/shogidrl

14 120 121 122 123 124 125 126 127 128 129 130 131 shogi_rules_logic.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/shogi/shogi_rules_logic.py

15 56 57 65 67 118 119 154 155 156 157 158 159 160 161 162 163 164 165 env_manager.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/training/env_manager.py

18 19 46 76 216 217 218 pyproject.toml
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/pyproject.toml

22 52 55 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 210 219 ppo_agent.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/core/ppo_agent.py

25 26 27 28 53 54 60 116 203 204 205 206 207 208 209 test_ppo_agent_core.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/tests/test_ppo_agent_core.py

29 30 31 32 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 220 222 display.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/training/display.py

34 35 184 185 186 187 188 189 190 191 unified_logger.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/utils/unified_logger.py

36 37 42 43 44 45 82 192 193 194 195 197 198 199 200 utils.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/utils/utils.py

58 59 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 221 trainer.py
https://github.com/tachyon-beep/shogidrl/blob/90d90845a17b0450f63431256e4f22d0247372d3/keisei/training/trainer.py