KESTRL: A DUAL-ROUTE JAX LIBRARY FOR MODULAR REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

We present KestRL, a dual-route reinforcement learning library built on Flax NNX, designed to let researchers build algorithms from reusable components and run them with both host-driven and JAX-native environments. KestRL builds on NNX graph/state separation to organize learning updates around explicit training state. It provides two execution routes: a Standard route, which combines Gymnasium environments with compiled learner updates, and a Compiled route, which brings Brax environments and learning updates into compiled training loops and supports batched execution of independent runs. SAC and PPO illustrate how learning code can be shared across these routes as collection and scheduling adapt to each environment. Controlled update comparisons and complete training experiments examine the resulting execution costs. For host-dispatched updates, the controlled comparisons show improvements over naive NNX dispatch, alongside workloads where cached NNX or hand-written functional JAX remains faster. In a five-run Ant PPO study at a fixed environment-step budget, KestRL Standard reduces end-to-end time by 32.8% relative to eager CleanRL and 25.7% relative to the evaluated Inductor-enabled configuration. In the reported Brax SAC configurations, KestRL Compiled shows similar independent-run scaling to Rejax, with lower observed amortized runtime on Ant and Hopper across the tested batch sizes.

1 INTRODUCTION

Reinforcement learning (RL) researchers need to modify algorithms and run repeated experiments across different environment implementations. A change to a network or loss may be reusable across these settings, but the surrounding training loop must accommodate how each environment runs. JAX enables numerical computations to be compiled for accelerators (Bradbury et al., 2018). With a host-driven simulator, environment interaction remains outside the compiled learner updates. When the simulator is also implemented in JAX, rollout and learning can be compiled together. Supporting both settings therefore requires deciding which components to share and how to manage training state, data collection and update schedules in each.

We present KestRL[1], a dual-route reinforcement learning library built on Flax NNX, which provides object-oriented neural-network components for JAX. We built KestRL so that researchers can carry changes to shared networks and losses across execution settings without maintaining a separate version of this code for each. To this end, KestRL organizes reusable components around explicit training state. Its Standard route compiles learner updates within a host-driven training loop organized around algorithm objects. Its Compiled route carries learner, environment and storage state through JAX control flow and batches independent training runs. Both routes use NNX components and functional update interfaces.

KestRL builds on ideas established by existing RL libraries. CleanRL emphasizes self-contained implementations, Stable-Baselines3 provides reusable algorithm interfaces, and TorchRL offers composable components (Huang et al., 2022; Raffin et al., 2021; Bou et al., 2024). PureJaxRL, Rejax and Stoix support compiled training and independent-run batching (Lu et al., 2022; Liesen et al., 2024; Toledo, 2024), while Flax NNX provides graph/state separation for JAX transformations (Heek et al., 2026). We bring these ideas together in a library that uses NNX components across two execution routes, with explicit interfaces between learner updates and the surrounding training loop.

[1]We like to pronounce it *kestrel*, after the small falcon.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We use Soft Actor-Critic (SAC) and Proximal Policy Optimization (PPO) to examine how this design accommodates different learning requirements (Haarnoja et al., 2018; Schulman et al., 2017). SAC learns from replayed transitions, while PPO collects fixed-horizon rollouts and performs repeated on-policy updates. PPO, for example, uses the same loss functions and minibatch update builder in both routes, so changes to this learning code carry over while each route retains its own rollout procedure. We then ask two practical questions: what does this organization cost during host-dispatched learner updates, and how do the resulting implementations perform in complete experiments? Controlled update comparisons address the first question; training times and learning outcomes address the second.

The controlled comparisons show that explicit state reduces naive NNX dispatch costs, although cached NNX or hand-written functional JAX remains faster in some workloads. Following this organization into complete experiments, we find shorter end-to-end times for Standard-route Ant PPO than for the evaluated CleanRL configurations, and lower observed amortized runtime for Compiled-route Brax SAC than for Rejax in the reported settings.

2 Background and Related Work

Modular components and training state. Researchers can work directly with a complete algorithm, as in CleanRL's self-contained scripts, or extend reusable interfaces, such as Stable-Baselines3's algorithm and policy classes (Huang et al., 2022; Raffin et al., 2021). Reuse can also reach inside the algorithm: TorchRL exposes composable collectors and losses, ObjectRL organizes algorithms and networks as objects, and Garage supplies modular tools that connect algorithm development to collection and experiment management (Bou et al., 2024; Baykal et al., 2025; garage contributors, 2019). Many of these libraries build on PyTorch's object-oriented neural-network components (Paszke et al., 2019; Ansel et al., 2024). Bringing this programming model to JAX requires making the components' changing state accessible to its transformations. Flax NNX provides that separation between module structure and state (Heek et al., 2026), with a performance guide covering both cached graph handling and functional training loops that pass state explicitly.[2] KestRL uses the latter approach to make learner updates available to both execution routes.

Environment interaction and execution. The environment implementation determines how far compilation can extend through training. With a JAX-native simulator such as Brax, environment steps and learning updates can be compiled together (Freeman et al., 2021; Bradbury et al., 2018). For host-driven environments accessed through interfaces such as Gymnasium (Towers et al., 2025), collection must instead be coordinated with accelerator work; SEED RL, Sample Factory and EnvPool address related challenges in inference, actor–learner coordination and environment throughput (Espeholt et al., 2020; Petrenko et al., 2020; Weng et al., 2022). Stoix supports both settings through Podracer's Anakin and Sebulba architectures (Hessel et al., 2021; Toledo, 2024): Anakin compiles training end to end with JAX-native environments, whereas Sebulba connects asynchronous environment threads to acting devices, with configurable device allocation for acting and learning. Across these execution settings, KestRL focuses on how NNX learning components and explicit training state can be organized for reuse.

Batching independent experiments. PureJaxRL, Rejax and Stoix use JAX vectorization to execute independent training runs together (Lu et al., 2022; Liesen et al., 2024; Toledo, 2024). Because each run maintains its own policy and training state, batching these experiments on an accelerator serves a different purpose from using several environments to collect data for one shared policy. KestRL's Compiled route follows this independent-run approach by batching learner, environment and storage state. To examine how it scales, we hold the environment-step budget per run fixed and report the collection and update schedules alongside the timings.

Table 1 summarizes the libraries' documented authoring and execution models. Rejax supports subclass-based extensions, and Stoix supports both JAX-native and host environments. KestRL organizes NNX learning components for reuse across these execution settings, leaving each route to supply the collection and scheduling appropriate to its environment.

[2] https://flax.readthedocs.io/en/v0.12.9/guides/performance.html

Table 1: RL libraries and their documented authoring and execution models. Entries summarize supported workflows rather than exhaustive capability limits or performance rankings. End-to-end JAX execution requires compatible environments.

Library	Backend	Authoring model	Execution organization
Stable-Baselines3 (Raffin et al., 2021)	PyTorch	Algorithm and policy classes	Host environment collection with reusable training interfaces
CleanRL (Huang et al., 2022)	PyTorch; JAX	Self-contained algorithm scripts	Script-specific loops; JAX variants include compiled learner updates
ObjectRL (Baykal et al., 2025)	PyTorch	Inheritance-based algorithm and network components	Host environment interaction with modular actor–critic learning
TorchRL (Bou et al., 2024)	PyTorch	Composable environments, collectors, losses and TensorDict modules	Host and accelerator components; compilation and vectorization support depend on the composed program
Garage (garage contributors, 2019)	PyTorch; TensorFlow	Algorithms, policies, samplers and trainers	Sampling workers separated from algorithm optimization
PureJaxRL (Lu et al., 2022)	JAX	Self-contained functional implementations	JIT training with JAX environments; batched independent runs via `vmap`
Rejax (Liesen et al., 2024)	JAX	Extensible algorithm classes and explicit state	JIT training; independent training states vectorized via `vmap`
Stoix (Toledo, 2024)	JAX	Research-oriented algorithm scripts with shared components	Anakin: end-to-end JAX training; Sebulba: asynchronous environments with configurable acting/learning device roles; `vmap`/`pmap` support
KestRL Standard	JAX/NNX	Algorithm classes and NNX components	Host environment/storage loop with compiled learner updates
KestRL Compiled	JAX/NNX	NNX components and algorithm-specific factories	Scanned Brax rollout/update carry; independent runs via `vmap`

3 Architecture

KestRL separates learning components from the training loops that use them. We follow this separation from module structure and changing state to the Standard and Compiled routes, then use SAC and PPO to show how learning code is reused as collection and scheduling adapt to each execution setting.

3.1 Separate module structure, learner state and execution state

Standard transformations in JAX operate on pure functions, which express changes to state through their inputs and outputs (Bradbury et al., 2018). In frameworks such as PyTorch (Paszke et al., 2019; Ansel et al., 2024), neural-network code is commonly organized around objects whose parameters and optimizer state change during training. Flax NNX brings these programming styles together: networks and optimizers can be constructed and updated as ordinary Python objects, then exposed to JAX transformations through graph/state separation (Heek et al., 2026). Its `nnx.split` operation produces a graph definition that records object types, connections and static attributes, together with state that holds their changing numerical values. This state takes the form of a *PyTree*: a nested structure of containers, such as dictionaries and tuples, whose leaves hold the state arrays. With `nnx.merge`, the graph definition and state can be combined again to reconstruct the objects. Passing NNX objects directly to `nnx.jit` requires graph handling at each host call, even when their structure has not changed. Keeping the graph definition fixed and passing state explicitly avoids repeating this work.

KestRL applies this separation to the learner as a whole, collecting the networks, optimizers and auxiliary NNX modules in a single container during construction. Splitting them together gives

one graph definition for the combined components and one learner-state PyTree. Let G denote this static graph definition and S_t the learner state at iteration t, including trainable parameters, target parameters, optimizer moments and auxiliary variables. Given a batch B_t and random input K_t, a learning update has the interface

$$(S_{t+1}, m_t) = U_G(S_t, B_t, K_t), \tag{1}$$

where U_G denotes the learner update with graph definition G held fixed, and m_t contains the resulting metrics. KestRL constructs this function once for the chosen module structure and passes the current learner state S_t explicitly on each call. The module structure therefore stays fixed as the learner state evolves.

This interface lets KestRL compile the update while retaining NNX objects inside its implementation. With just-in-time (JIT) compilation, JAX first traces the function to record its array operations and then compiles those operations for execution. During this trace, the update reconstructs the learner container from G and S_t with `nnx.merge`, performs its learning operations through the contained modules, and extracts their updated state together with `nnx.split`. Once compiled, the update accepts new state and data without repeating the object reconstruction and state extraction on the host. Reuse depends on compatible inputs; changes to module structure or array shapes and types may require a new trace and compilation. Section 4.1 examines the resulting execution costs, which also depend on the array computation, call frequency and state layout.

To use this update in a complete training loop, we must also manage the environment state, collected data and information that determines when updates occur. This broader execution state connects the learner to its environment and update schedule. Figure 1 summarizes how the learner is constructed

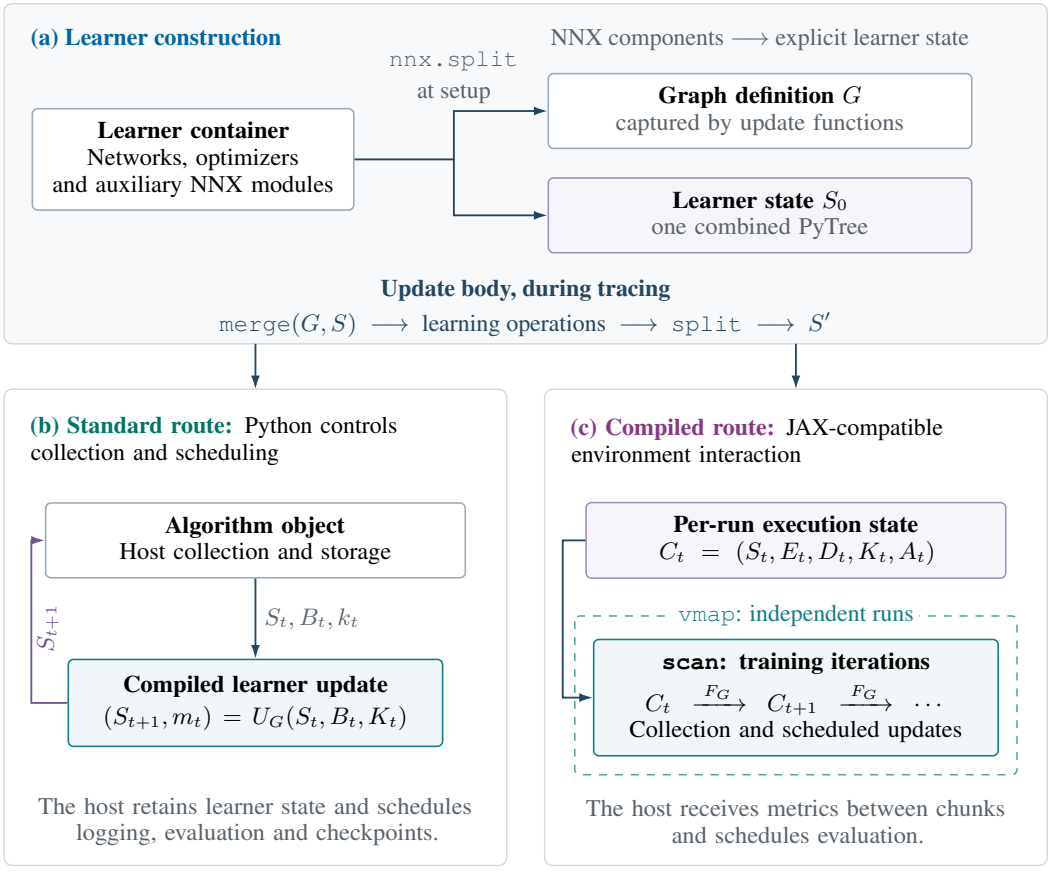


Figure 1: KestRL's learner organization and execution routes: **(a)** Components are split together at setup; the outer merge/split operations in each update body occur during tracing. **(b)** The Standard route retains host control, while **(c)** the Compiled route carries execution state through scanned iterations and batches independent runs.

and how the two routes organize this broader execution state. The following subsections describe each route in detail.

3.2 Standard route: retain host interaction, compile the learner

The Standard route keeps the training loop under Python control so that algorithm objects can work directly with host-driven environments and experiment tools, following an organization familiar from modular libraries such as Garage (garage contributors, 2019). Once an environment builder has created the Gymnasium environments, the selected algorithm constructs its networks, optimizers and data storage using the common environment information supplied by `BaseAlgorithm`. Subclasses build on this interface to define action selection, collection and learning, and inherit the progress tracking and evaluation methods used around these operations.

As training proceeds, the `Trainer` calls the algorithm's `train_step` method to collect data and apply the learning operations due at that point, and receives metrics in return. Through this common interface, the trainer monitors progress and schedules logging, evaluation and optional checkpoints, while the algorithm determines how much experience to collect and when to use it for learning.

Within this host-driven loop, policy inference and learner updates operate on JAX arrays. The algorithm prepares the inputs, passes the current learner state to the compiled update and retains the returned state for subsequent calls, following Equation 1. The compiled region can contain a single learning operation or a sequence of updates. Environment interaction remains outside that region, so the update can be organized independently of the simulator's Python interface. Inside the compiled region, loss functions remain written as direct network calls and array operations. Figure 2 compares a loss calculation from CleanRL with the corresponding KestRL helper to show how this familiar form is preserved within the explicit-state update interface.

CleanRL (PyTorch)

```
# cleanrl/sac_continuous_action.py

pi, log_pi, _ = actor.get_action(
    data.observations)



qf1_pi = qf1(data.observations, pi)
qf2_pi = qf2(data.observations, pi)
min_qf_pi = torch.min(qf1_pi, qf2_pi)



actor_loss = (
    (alpha * log_pi) - min_qf_pi
).mean()
```

KestRL (Flax NNX)

```
# src/kestrl/algorithms/sac/functions.py

action, log_prob, _ = \
    get_continuous_actor_action(
        actor, obs, action_scale,
        action_bias, key)

critics_input = jnp.concatenate(
    [obs, action], axis=1)
qf1_values = critic1(critics_input)
qf2_values = critic2(critics_input)
min_q = jnp.minimum(qf1_values, qf2_values)

actor_loss = jnp.mean(
    (alpha * log_prob) - min_q)
```

Figure 2: SAC actor-loss calculations in CleanRL (Huang et al., 2022) and KestRL: action sampling, critic evaluation and actor loss.

3.3 Compiled route: training through explicit execution state

The Standard route compiles learner updates, but environment interaction remains under Python control. Exchanging actions and observations between the accelerator and the host introduces data transfers and synchronization. Together with the time spent stepping the environment, these costs can limit accelerator utilization. When the environment runs on the accelerator through JAX, KestRL can bring data collection and learning into the same compiled loop. To support this loop, the Compiled route extends the learner-state interface to include everything needed to advance a training iteration. This requires reset and step functions compatible with JAX transformations; the current integration provides them through Brax Freeman et al. (2021).

To advance this loop, each iteration must receive the state needed to continue both collection and learning. We collect these values in a *carry*[3], denoted by $C_t = (S_t, E_t, D_t, K_t, A_t)$, where S_t is the

[3]See `https://docs.jax.dev/en/latest/_autosummary/jax.lax.scan.html`.

learner state, E_t the environment state, D_t any data storage that must persist between iterations, K_t the random state and A_t auxiliary counters and statistics. The algorithm determines which fields must persist; temporary data created and consumed within an iteration need not remain in the carry. With this broader state made explicit, a training iteration has the interface

$$(C_{t+1}, m_t) = F_G(C_t), \tag{2}$$

where F_G collects experience, manages storage and applies the scheduled learner updates. As in the learner-update interface, the module structure G stays fixed while the state evolves. The returned carry C_{t+1} supplies the next iteration, and m_t records the resulting metrics.

To turn this interface into an executable experiment, each algorithm provides a factory: a function that takes the environment and configuration and constructs the training functions. Through these functions, `CompiledTrainer` can initialize the carry, advance training for a single run or a batch of independent runs, and evaluate the learned policy, while the factory defines the collection procedure and learning schedule.

Within the training computation, the JAX loop primitive `jax.lax.scan` applies F_G repeatedly and passes the updated carry from one iteration to the next. This lets a single compiled call advance training through a chunk of iterations before returning its state and metrics to the trainer. Between chunks, the host can log progress or invoke evaluation, then resume training from the returned carry. The experiment therefore retains access to reporting and evaluation without requiring Python to coordinate each training iteration.

The same organization also supports repeated experiments. To execute multiple independent runs together, KestRL uses JAX's `vmap` to apply the initialization and chunk computation across a batch of separate carries. Each run maintains its own random stream, learner state, environment state and any persistent storage, so collection and learning proceed independently within the shared program. This batching requires memory for each run's state; memory use therefore grows with the number of runs.

3.4 Component reuse in practice: SAC and PPO

The two routes separate component definitions from the execution logic that uses them. SAC and PPO illustrate how this separation works for algorithms with different data requirements (Haarnoja et al., 2018; Schulman et al., 2017). Both use NNX networks and optimizers, but they assemble different learner states and organize experience differently.

In SAC, both routes reuse loss and action-selection helpers, but organize data collection and update execution separately. The Standard implementation manages replay storage on the host and calls compiled learning operations on sampled batches. The Compiled implementation keeps replay storage in the carry and performs sampling and learning within JAX control flow, with its own gradient-update loop.

In PPO, reuse extends to the minibatch update builder, which both routes use with data from their own rollout procedures. Because the compiled implementation creates and consumes its rollout within each training iteration, its carry does not need persistent rollout storage. The Standard implementation can also run its minibatch updates as one compiled optimization phase between host-driven rollouts.

A researcher experimenting with a different PPO objective can modify the corresponding loss function, which both routes use through the same minibatch update builder. If the experiment also calls for a different rollout procedure, that part can be adapted to the environment without duplicating the loss. A new algorithm can draw on shared components in much the same way, with its collection and learning schedules expressed through the Standard algorithm interface or the Compiled factory interface.

4 Evaluation

This section addresses the two questions introduced earlier: *what does KestRL's organization cost during host-dispatched learner updates*, and *how do the resulting implementations perform in complete experiments?* We begin with controlled comparisons that hold the update computation and inputs fixed, then follow the Standard and Compiled routes through complete training workloads to

examine runtime alongside learning outcomes. The controlled comparisons and Standard-route PPO study use an RTX A6000, while the SAC batching study uses an H100; runtime comparisons are made within each study, with detailed protocols in Appendix A.

4.1 Controlled cost of modular updates

To measure the cost of dispatching learner updates from the host, we compare KestRL's explicit-state interface with alternative ways of handling NNX objects and with hand-written functional JAX. Within each comparison, the update computation, initial state, data and random inputs are held fixed, so the variants execute the same learning workload. We time these updates separately from collection, evaluation and logging.

For the NNX comparison, we use HalfCheetah data captured before timing to compare naive `nnx.jit`, `nnx.cached_partial` and KestRL's explicit-state updates. PPO processes a 2,048-transition rollout through ten epochs and 32 minibatches per epoch, including minibatch gathering in the timed host loop, while SAC performs a critic, actor/temperature and target update sequence. After checking agreement in metrics and full learner state, we run two warm-ups followed by 20 synchronized timing repetitions; Table 2 reports their medians and interquartile ranges (IQRs).

Table 2: Captured-data update control on RTX A6000. Median (IQR), milliseconds. The measured region contains the specified host update loop and excludes collection, evaluation and logging.

Workload (ms)	`nnx.jit`	Cached NNX	KestRL
PPO: 320 minibatches	3267.4 (86.1)	2775.8 (64.3)	2614.7 (160.2)
SAC: one update sequence	11.170 (0.547)	3.129 (0.096)	3.548 (0.067)

Exposing state explicitly reduces the cost of host dispatch: KestRL is 1.25 times as fast as naive NNX for PPO and 3.15 times as fast for SAC in these workloads. Cached NNX also reduces this cost and is faster than KestRL in the SAC comparison. Its `nnx.cached_partial` callable binds learner objects and propagates changes through their shared variables (Heek et al., 2026), so it cannot be used unchanged as a pure state transition inside `jax.lax.scan` or `jax.vmap`. KestRL's state inputs and outputs allow the learning operation to take part in these larger computations, extending its use from host-dispatched updates to scanned training and batches of independent runs.

We next compare KestRL with a hand-written JAX/Optax implementation on captured Ant inputs, measuring both a single update and a longer sequence for each algorithm (Table 3). On these fixed inputs, both alternatives agree with naive NNX, with zero maximum differences in metrics and trainable parameters across all four workloads. PPO uses minibatches prepared before timing for the same host-driven update sequence in each variant. The SAC sequence repeats a cycle of two critic/target updates and one actor/temperature update 100 times, retaining separate critic, actor/temperature and target JIT boundaries in every variant. These choices of data preparation and update boundaries differ from the NNX comparison above, so timings should only be compared within each table.

Reducing naive dispatch overhead brings KestRL closer to the functional reference, reaching 85.2% of its throughput in the longer PPO workload and 72.1% in SAC. The remaining gap measures the cost of the NNX-based update implementation in these controlled workloads. To see how this cost figures in an experiment, we next include environment interaction and the rest of the training process.

4.2 PPO: runtime and learning across routes

For the complete-run comparison, KestRL Standard, eager CleanRL and an Inductor-enabled CleanRL configuration each train Ant-v5 PPO over five runs at a budget of 500,000 requested steps. Completing the final rollout brings each run to 501,760 steps. We follow the experiment from construction through training and evaluation to finalization, so the total includes both compilation and the work surrounding the learner updates. Runs execute sequentially on an exclusively used A6000, with method order rotated across runs; the evaluation schedule and full timing protocol are given in Appendix A.

Table 3: Functional-reference control on RTX A6000. Median (IQR), 20 synchronized repetitions. The last column is KestRL throughput divided by reference throughput, equivalently $T_{\text{reference}}/T_{\text{KestRL}}$. The longer PPO workload uses host-dispatched updates; the full-run study separately evaluates a fused optimizer.

Workload	Naive NNX	KestRL	JAX + Optax	Throughput
PPO: one minibatch (ms)	2.821 (0.020)	0.616 (0.013)	0.494 (0.010)	80.2%
PPO: 320 minibatches (s)	1.245 (0.039)	0.486 (0.081)	0.414 (0.013)	85.2%
SAC: one cycle (ms)	30.223 (3.140)	6.920 (0.134)	4.756 (0.142)	68.7%
SAC: 100 cycles (s)	1.994 (0.407)	0.742 (0.014)	0.536 (0.011)	72.1%

Table 4: Ant-v5 PPO study: mean $\pm$ sample SD across five runs. The cold first iteration contains rollout/update work and compilation where applicable. Steady iterations include collection and optimization. Full phase accounting is in Appendix B.1.

Measurement	KestRL	CleanRL eager	CleanRL + Inductor
Cold first iteration (s)	14.93 ± 1.58	5.63 ± 0.08	5.10 ± 0.14
Steady iterations (s)	1341.22 ± 69.52	2165.75 ± 113.21	1935.77 ± 121.11
Evaluation (s)	182.13 ± 29.47	121.73 ± 21.48	131.61 ± 29.32
End-to-end (s)	1549.61 ± 79.35	2304.83 ± 101.49	2085.82 ± 142.03
Final return	1006.00 ± 259.90	1196.40 ± 225.70	1120.00 ± 82.70

KestRL completes this workload in 32.8% less end-to-end time than eager CleanRL and 25.7% less than the Inductor configuration (Table 4). The shorter steady training phase outweighs its higher cold-start and evaluation costs. In KestRL, minibatch scheduling and updates run together in a compiled optimization phase. In the Inductor configuration, compilation covers the agent graph and associated automatic differentiation; loss construction, optimizer operations and minibatch scheduling remain outside that graph. The timings reflect these implementations' update and compilation choices; configuration details are given in Appendix B.1. KestRL's final return is lower on average, so the time saving concerns completion of the fixed training budget rather than time to a common policy quality.

Separate five-run suites follow Standard-route PPO learning on discrete CartPole and continuous MuJoCo Ant. CartPole approaches its 500-step episode ceiling after approximately 100,000 training steps, with a final return of 499.5 ± 1.1 across seeds. MuJoCo Ant is more variable, reaching 1165.1 ± 234.6 at roughly one million steps, twice the requested budget of the timing study. Appendix B.2 gives the individual evaluation means behind these summaries, together with their sample standard deviations and bootstrap intervals.

The same PPO minibatch update builder is used with Brax rollouts in the Compiled route, where KestRL trains five initializations together. At roughly one million steps per run, both KestRL and a separately executed Rejax configuration remain in a low-return regime under the same nominal architecture, rollout budget, optimization schedule and deterministic evaluation protocol. These outcomes therefore offer limited evidence of learning performance in the compiled PPO setting; the returns and their variation across seeds are reported in Appendix B.2.

4.3 SAC ACROSS ROUTES: RUNTIME, BATCHING AND LEARNING

SAC brings both routes into the comparison on Ant and Hopper, with a budget of one million environment steps per run on an H100. In MuJoCo, KestRL Standard completes both tasks in times close to Stable-Baselines3's and shorter than CleanRL's; on Ant, for example, it takes approximately 74 minutes against CleanRL's 118. The left-hand plot in Figure 3 places these results at $N = 1$, since the CleanRL and SB3 SAC implementations do not vectorize independent training runs. The Compiled route is included as a reference alongside this comparison of host-driven implementations.

With Brax, environment interaction and learning can run in the same compiled loop, allowing us to follow batches of 1, 5, 10, 50 and 100 independent runs. The right-hand plot compares KestRL Compiled with Rejax and Stoix in this setting. We report both the time T_N to complete a batch and

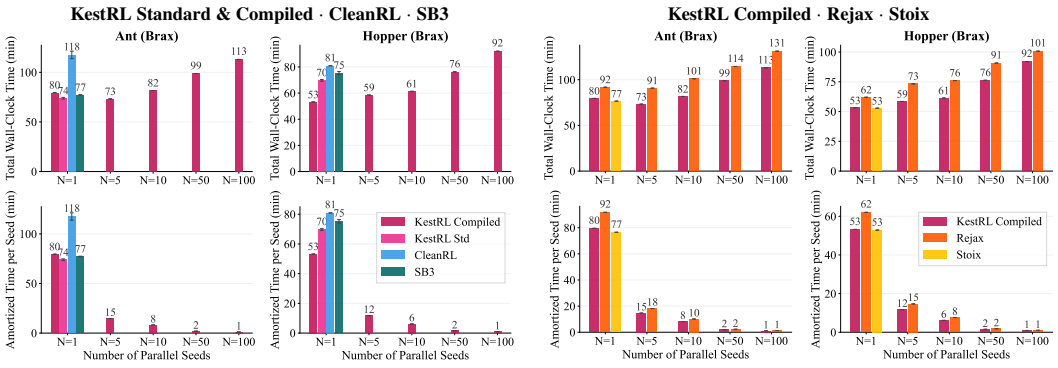


Figure 3: SAC wall-clock times on H100, 10^6 environment steps per run. *Left:* KestRL Standard, CleanRL and SB3 on MuJoCo at $N = 1$, with KestRL Compiled on Brax for reference. *Right:* KestRL Compiled, Rejax and Stoix on Brax; Stoix is shown at $N = 1$. Upper panels show batch time T_N; lower panels show time per run T_N/N. Error bars summarize variation across five seeds at $N = 1$ and the N seeds in larger batches. See Appendix A.3 for protocols.

the amortized time T_N/N per run. The two routes use different simulators, so their relative times reflect the environment as well as the training implementation.

For KestRL, increasing the batch from one to 100 runs raises total time from approximately 80 to 113 minutes on Ant and from 53 to 92 minutes on Hopper. The resulting amortized times are 1.13 and 0.92 minutes per run. Rejax follows a similar scaling pattern, with higher observed amortized times at every reported batch size on both tasks.

Therefore, batching lets these configurations complete multiple independent experiments with a smaller increase in runtime compared with sequential execution.

KestRL's Ant traces reach similar final returns across batch sizes despite differences earlier in training; on Hopper, the $N = 1$ curve fluctuates more than the larger-batch curves (Figure 4). Reading these traces alongside the baselines requires accounting for how returns are collected: KestRL reports training returns, whereas Rejax and Stoix report evaluations. Rejax's periodic evaluations also add work to its timed runs. Appendix A.3 describes these protocols and the differences in learning rates.

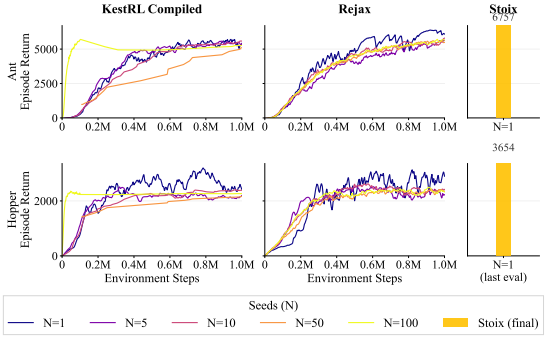


Figure 4: Brax SAC learning curves accompanying Figure 3 (right). Colors indicate batch size N; return collection differs as described alongside the plot.

5 Limitations and Conclusion

The evidence presented here comes from feed-forward SAC and PPO experiments with Gymnasium and Brax. Although KestRL was designed to accommodate image-based learning, this setting remains to be evaluated, as do recurrent policies, model-based methods and other simulator integrations.

KestRL brings familiar, object-based development and compiled execution into the same framework through reusable NNX learning components. SAC and PPO show how these components fit into training loops with different data requirements and execution schedules, and the experiments follow their performance from individual updates to complete runs. A researcher can thus develop a loss or another shared learning operation in a host-driven experiment and carry that work into compiled training.

AI USE STATEMENT

KestRL grew out of the corresponding author's doctoral research and the need for a unified framework that combined a familiar development interface with fast execution. The library's conception and design, including its two execution routes, were developed by the author without generative AI assistance.

The code for the baseline experiments was developed using generative AI. During manuscript preparation, the authors also used generative AI tools to iterate on an existing skeleton, giving detailed guidance on the organization of the argument and the refinement of the prose. This work included checking the text against the implementation and supplied experimental materials, and bringing those materials into the appendix and its reproducibility tables, with assistance in document formatting. The authors directed these revisions and retain responsibility for the experimental work, the scientific claims and the final content of the paper.

REPRODUCIBILITY STATEMENT

To reconstruct an experiment, readers can follow Appendix A from its source files and configurations to the software and hardware settings, training budget, evaluation procedure and treatment of seeds. The controlled comparisons include agreement checks before timing; for the complete PPO experiment, Appendix B.1 identifies the phases included in the measured runtime. The configuration and timing tables are generated from YAML files and retained records, with the principal timing summaries checked against individual run logs by the generation script. Readers can likewise trace the aggregate learning results to the individual outcomes in Appendix B.2, then use the source layout in Appendix C to locate the shared learning components and the implementations specific to each route.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi: 10.1145/3620665.3640366. URL https://docs.pytorch.org/assets/pytorch2-2.pdf.

Gulcin Baykal, Abdullah Akgül, Manuel Haussmann, Bahareh Tasdighi, Nicklas Werge, Yi-Shan Wu, and Melih Kandemir. Objectrl: An object-oriented reinforcement learning codebase, 2025. URL https://arxiv.org/abs/2507.03487.

Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. Torchrl: A data-driven decision-making library for pytorch. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=QxItoEAVMb.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: scalable and efficient deep-rl with accelerated central inference. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rkgvXlrKwH.

C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A differentiable physics engine for large scale rigid body simulation. In Joaquin Vanschoren and Sai-Kit Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/d1f491a404d6854880943e5c3cd9ca25-Abstract-round1.html.

The garage contributors. Garage: A toolkit for reproducible reinforcement learning research. https://github.com/rlworkgroup/garage, 2019.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1861–1870. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/haarnoja18b.html.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2026. URL http://github.com/google/flax.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning, 2021. URL https://arxiv.org/abs/2104.06272.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html.

Jarek Liesen, Chris Lu, and Robert Lange. rejax, 2024. URL https://github.com/keraJLi/rejax.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019. URL http://arxiv.org/abs/1912.01703.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3D control from pixels at 100000 FPS with asynchronous reinforcement learning. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7652–7662. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/petrenko20a.html.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Edan Toledo. Stoix: Distributed single-agent reinforcement learning end-to-end in jax, April 2024. URL https://github.com/EdanToledo/Stoix.

Mark Towers, Ariel Kwiatkowski, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun Kg, Rodrigo Perez-Vicente, J. K. Terry, Andrea Pierré, Sander V. Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2407.17032. Datasets and Benchmarks Track.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, Zhongwen Xu, and Shuicheng Yan. Envpool: A highly parallel reinforcement learning environment execution engine. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 22409–22421. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/8caaf08e49ddbad6694fae067442ee21-Paper-Datasets_and_Benchmarks.pdf.

A EXPERIMENTAL SETUP AND REPRODUCIBILITY

The experiments follow KestRL from individual learner updates to complete training runs. Each setting answers a different question, so reproducing a result begins with its workload and timing boundary. The controlled comparisons use captured data and exclude environment interaction. The PPO timing study follows a complete host-driven experiment, including evaluation and finalization. The SAC study then examines how complete runs scale when independent learners are batched together.

The repository contains the benchmark drivers under `baselines/` and the retained PPO and update-control records under `misc/rebuttal_artifacts/`. The tables below connect these experiments to their software environment, configuration files and run-specific overrides. `misc/appendix/build_tables.py` generates the configuration and timing tables from these sources and checks the principal timing summaries against the 15 retained audit logs.

A.1 SOFTWARE AND HARDWARE

Table 5 records the software versions used throughout the experiments. The controlled update comparisons and Standard-route PPO study use an RTX A6000; the SAC batching study uses an H100, with the additional TPU experiments described in Appendix D.

Table 5: Software versions used across the experiments. The accelerator entry identifies the GPU used for the controlled update comparisons and Standard-route PPO study.

Component	Version or setting
Accelerator	NVIDIA RTX A6000, one GPU
Python	3.11.15
CUDA toolkit	12.8.1
JAX / JAXlib	0.9.0.1 / 0.9.0.1
Flax NNX	Flax 0.12.4
Optax	0.2.6
Chex	0.1.91
NumPy	2.3.5
PyTorch	2.11.0+cu128
Gymnasium	1.2.3
MuJoCo	3.5.0
Installed Brax package	0.14.2
Hydra / OmegaConf	1.3.2 / 2.3.0
Experiment tracking	Weights & Biases 0.19.11

The experiment entry point composes three Hydra configuration groups: algorithm, environment and experiment. Algorithm YAML files define the networks and learning settings; the environment group selects the simulator and its wrappers; the experiment group controls seeds, reporting and evaluation. Command-line overrides select the settings of an individual study. In particular, a YAML training budget is a default, not necessarily the budget of every reported run.

A.2 PPO CONFIGURATIONS AND EVALUATION

The CartPole experiments use `configs/algorithm/ppo_cartpole.yaml`, and the MuJoCo and Brax experiments use `configs/algorithm/ppo_mujoco.yaml`, relative to `src/kestrl/`. Table 6 gives their configured values. Policy and value networks are separate MLPs. The rollout lengths and minibatch counts give minibatches of 32 transitions for CartPole and 64 for Ant.

The Standard trainer completes the rollout that crosses the requested budget, whereas the compiled suite executes an integer number of complete rollout iterations. Table 7 therefore gives both requested and executed steps. This also distinguishes the 500,000-step timing study from the longer Standard Ant learning suite: their final returns belong to different training budgets.

In the timing audit, each method trains with one Ant-v5 environment and evaluates its policy in a separate environment, without observation or reward normalization. The measured run includes five

Table 6: PPO YAML settings. The MuJoCo configuration also supplies the Compiled route's Brax PPO settings. Experiment-specific budgets and evaluation counts are listed in Table 7.

Setting	CartPole	MuJoCo / Brax
Policy and value hidden layers	64–64	64–64
Activation	tanh	tanh
Requested steps (YAML default)	500000	1000000
Rollout steps	128	2048
Initial learning rate	0.00025	0.0003
Discount γ	0.99	0.99
GAE λ	0.95	0.95
Update epochs	4	10
Minibatches per epoch	4	32
Policy clipping coefficient	0.2	0.2
Value clipping	Enabled	Enabled
Entropy coefficient	0.01	0.0
Configured `vf_coef`	0.5	0.5
Maximum gradient norm	0.5	0.5
Advantage normalization	Enabled	Enabled
Linear learning-rate annealing	Enabled	Enabled
Target KL	None	None

Table 7: PPO study budgets and final evaluation. Every setting has five independent training seeds. Episode counts apply to each trained policy; actions are deterministic during evaluation.

Study	Requested steps	Executed steps	Eval. episodes
Ant-v5 timing audit, all methods	500,000	501,760	20
CartPole-v1 / Standard	100,000	100,096	20
Ant-v5 / Standard learning	1,000,000	1,001,472	20
Brax Ant / KestRL Compiled	1,000,000	999,424	10
Brax Ant / Rejax	1,000,000	999,424	10

periodic evaluations and a final evaluation of 20 episodes each. Tracking remains enabled throughout, but checkpointing is disabled for all three methods, so saving checkpoints contributes no work to this comparison.

For the Brax learning comparison, the two implementations use five initializations with the same nominal network size and rollout length. Each rollout is processed through the same numbers of epochs and minibatches, and the resulting policies are evaluated using their Gaussian means. In Rejax, the learning rate decreases linearly from 3×10^{-4} to zero over the resulting 156,160 optimizer updates: 488 rollouts with 320 updates per rollout.

A.3 SAC CONFIGURATIONS AND SEED BATCHING

The continuous-control SAC experiments use `configs/algorithm/sac_mujoco.yaml`. Table 8 records its network and learning settings. One environment supplies experience to each learner. Increasing the number of independent seeds therefore creates additional learners, replay buffers and random streams, rather than additional environments feeding a common policy.

The evaluated baseline drivers use networks with two hidden layers of 256 units, replay capacity 10^6, minibatches of 256, 5,000 warm-up steps and discount 0.99. These settings span both simulator groups: KestRL Standard, CleanRL and SB3 use Gymnasium/MuJoCo, and KestRL Compiled, Rejax and Stoix use Brax. The target interpolation factors are also equivalent despite their different parameterizations: Rejax's retention factor of 0.995 corresponds to the interpolation coefficient $\tau = 0.005$ in the other configurations. Rejax differs in its learning rates, using 3×10^{-4} for both actor and critic where KestRL, CleanRL, SB3 and Stoix use 3×10^{-4} and 10^{-3}, respectively. These are the settings selected in the evaluated drivers, not necessarily the libraries' defaults.

The H100 campaign uses $N \in \{1, 5, 10, 50, 100\}$ for KestRL Compiled and Rejax. The $N = 1$ results aggregate five seeds; larger batches use their N independent seeds for the plotted variation.

Table 8: KestRL's continuous-control SAC configuration, shared by the MuJoCo and Brax experiment entry points. Intervals are expressed in environment steps; learning rates are constant.

Setting	Value
Actor and critic hidden layers	256–256
Activation	relu
Environment steps	1000000
Replay capacity	1000000
Warm-up steps	5000
Minibatch size	256
Collection interval	1
Gradient steps	1
Policy-update interval	2
Target-update interval	1
Discount γ	0.99
Target interpolation τ	0.005
Actor learning rate	3e-4
Critic learning rate	1e-3
Temperature learning rate	3e-4
Temperature autotuning	Enabled

Host-driven results and Stoix appear at $N = 1$ on the batch-size axis. Their five-seed summaries remain comparisons of independently trained policies. The SAC summaries use means and sample standard deviations across seeds; the bars are not confidence intervals.

Learning measurements follow the execution of each driver. KestRL records training returns. The Rejax callback evaluates two episodes every 5,000 steps, and the Stoix reference supplies final evaluation values. The retained Stoix speed launcher uses `-pure-speed`, which selects a final evaluation rather than intermediate evaluation checkpoints. Evaluation work therefore enters the elapsed-time comparison as well as the interpretation of the return curves. Table 9 connects these settings to the corresponding entry points.

Table 9: Entry points for the SAC comparison. The KestRL command selects its route from the environment configuration. Paths are relative to the repository root.

Implementation	Entry point and relevant selection
KestRL Standard	`src/kestrl/experiments/run.py`; `algorithm=sac_mujoco`, `environment=mujoco`
KestRL Compiled	Same entry point; `environment=brax`, with `experiment.num_seeds` selecting the batch size
CleanRL	`baselines/cleanrl_sac.py`; one seed per invocation
Stable-Baselines3	`baselines/sb3_sac.py`; one seed per model instance
Rejax	`baselines/rejax_sac.py`; `-num-seeds`, `-eval-freq 5000`
Stoix	`baselines/stoix_sac.py`; Brax environment, `-pure-speed`

A.4 Controlled update workloads

The captured-data NNX comparison uses `baselines/frozen_bundle_realistic_ablation.py`, whose filename retains the earlier internal name for the explicit-state implementation. It collects HalfCheetah-v5 data before timing, then supplies identical initial learner state, batches, permutations and random inputs to naive NNX, cached NNX and KestRL. PPO processes 2,048 transitions through ten epochs and 32 minibatches per epoch; minibatch gathering is included in its timed host loop. SAC executes a critic, actor/temperature and target update sequence. Agreement in metrics and complete learner state is checked before measurement.

The functional-reference comparison uses `baselines/functional_jax_ablation/functional_jax_ablation.py` with captured Ant-v5 data. PPO minibatches are prepared before timing. Its short workload is one minibatch and its longer workload is the 320-minibatch host-dispatched sequence. A SAC cycle contains two critic updates, two target updates and one

actor/temperature update; the longer workload repeats this cycle 100 times. Each variant retains separate critic, actor/temperature and target JIT boundaries. The recorded checks compare metrics and trainable parameters across NNX and raw Optax representations and report zero maximum differences on the tested inputs.

After two warm-ups, both controls measure 20 synchronized repetitions of the selected update workload, excluding collection, evaluation and logging. The tables summarize those repetitions with medians and interquartile ranges. Although the timing procedure is shared, the data preparation and call boundaries differ between the controls, so their absolute times should be compared within their respective tables.

B Detailed timing and learning results

B.1 Accounting for the complete PPO experiment

The phase breakdown in Table 10 follows an Ant-v5 PPO run from construction to finalization. It expands Table 4 with the costs of tracking setup, evaluation work and shutdown. The analysis includes the five completed runs per method from the fused-update, no-checkpoint sweep. Earlier checkpointed or unfused runs are separate configurations and are not pooled with this study.

Table 10: Complete PPO wall-clock audit on RTX A6000: mean ± sample SD over five runs per method. Times are seconds unless the row specifies steps, steps/s or return. Rates are means of per-run ratios, so a ratio of table means need not reproduce them. Training wall time includes training iterations, evaluation and runtime reporting; its constituent rows must not be added to it again. A dash denotes an inapplicable phase.

Measurement	KestRL	CleanRL eager	CleanRL + Inductor
Environment/model construction	5.01 ± 0.99	0.97 ± 0.04	0.81 ± 0.30
Logging setup	1.68 ± 0.16	2.48 ± 0.14	2.80 ± 0.66
Inductor warm-up	–	–	4.48 ± 0.18
Cold first iteration	14.93 ± 1.58	5.63 ± 0.08	5.10 ± 0.14
Steady training iterations	$1,341.22 \pm 69.52$	$2,165.75 \pm 113.21$	$1,935.77 \pm 121.11$
Evaluation rollouts	182.13 ± 29.47	121.73 ± 21.48	131.61 ± 29.32
Checkpointing	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Host logging	1.56 ± 0.15	0.86 ± 0.07	0.82 ± 0.03
Training wall time	$1,540.21 \pm 79.20$	$2,293.90 \pm 105.85$	$2,073.28 \pm 140.44$
Finalization / tracking	2.66 ± 0.35	6.32 ± 7.31	4.30 ± 3.03
End-to-end	$1,549.61 \pm 79.35$	$2,304.83 \pm 101.49$	$2,085.82 \pm 142.03$
Evaluation environment steps	$93,077 \pm 16,733$	$87,385 \pm 9,501$	$92,261 \pm 13,261$
Evaluation steps/s	511.03 ± 38.36	734.60 ± 142.90	711.86 ± 78.84
Training steps/s	326.47 ± 17.02	219.11 ± 10.07	242.93 ± 16.86
Final deterministic return	$1,006.00 \pm 259.90$	$1,196.40 \pm 225.70$	$1,120.00 \pm 82.70$

The cold first KestRL iteration includes both a real rollout/update and lazy JAX compilation. It is therefore distinct from a measurement of compilation alone. Subsequent PPO optimization phases gather and update minibatches within a scanned, compiled computation. The Inductor configuration applies `torch.compile` to the agent with `backend="inductor"`, `mode="max-autotune"`, `fullgraph=True` and `dynamic=False`. Its fixed-shape forward and associated differentiation are compiled; loss construction, optimizer operations and minibatch scheduling remain outside that graph. The eager configuration uses the same driver without this compilation step.

KestRL's shorter steady training phase more than offsets the additional time spent on construction, its first iteration and evaluation, yielding a shorter complete run. The evaluation cost depends in part on the learned policy: even with the same episode count, policies that produce different episode lengths require different numbers of environment transitions. Reading the total runtime alongside the final-return row therefore keeps both the cost of completing the fixed training budget and the resulting policy quality in view.

B.2 PPO OUTCOMES ACROSS SEEDS

Tables 11 and 12 report the learning suites separately from the shorter timing audit. Each training seed contributes one mean over its final evaluation episodes. The sample standard deviation summarizes variation among those trained policies; the bootstrap interval summarizes uncertainty in their mean. Run indices identify replicates within each suite, not matched trajectories across implementations.[4]

Table 11: PPO outcomes over five training seeds: mean ± sample SD and nonparametric bootstrap 95% intervals for the mean. Budgets and evaluation episode counts are given in Table 7.

Setting	Final return	95% interval
CartPole-v1 / Standard	499.5 ± 1.1	[498.6, 500.0]
MuJoCo Ant-v5 / Standard	1165.1 ± 234.6	[1013.9, 1377.6]
Brax Ant / Compiled	152.3 ± 110.3	[70.6, 239.2]
Brax Ant / Rejax	100.8 ± 26.4	[81.8, 123.2]

Table 12: Run-level PPO evaluation means. CartPole and MuJoCo Ant use 20 episodes per policy; Brax uses ten.

Run index	CartPole	MuJoCo Ant	Brax Ant	Rejax Brax Ant
1	500.00	1219.62	190.55	82.11
2	500.00	1010.82	128.97	141.16
3	500.00	1006.58	118.58	94.29
4	500.00	1034.68	312.15	111.28
5	497.65	1553.97	11.13	75.27

The learning outcomes vary substantially across these suites. CartPole reaches its 500-step episode limit in four of the five runs, whereas Standard Ant produces a broader spread of final returns. In the Brax setting, returns remain low for both KestRL and Rejax under the reported configuration. The suites thus exercise PPO in both routes, but their outcomes must be read within each environment: equal training budgets do not imply equal task difficulty across simulators.

[4]The Standard Ant suite retains the successful retry of its second seed after an unsuccessful attempt; that attempt is not an additional replicate.

B.3 SAC LEARNING ACROSS EXECUTION SETTINGS

Figure 5 complements the compiled comparisons with the host-driven SAC learning traces. Its columns separate the MuJoCo implementations from KestRL's Brax implementation, so the reader can follow learning in each execution setting. Ant and Hopper supply the timing comparisons in the main text; HalfCheetah and Walker2d extend the view of learning behavior. Their returns should be interpreted within their simulator settings: a common task name does not make the MuJoCo and Brax dynamics identical.

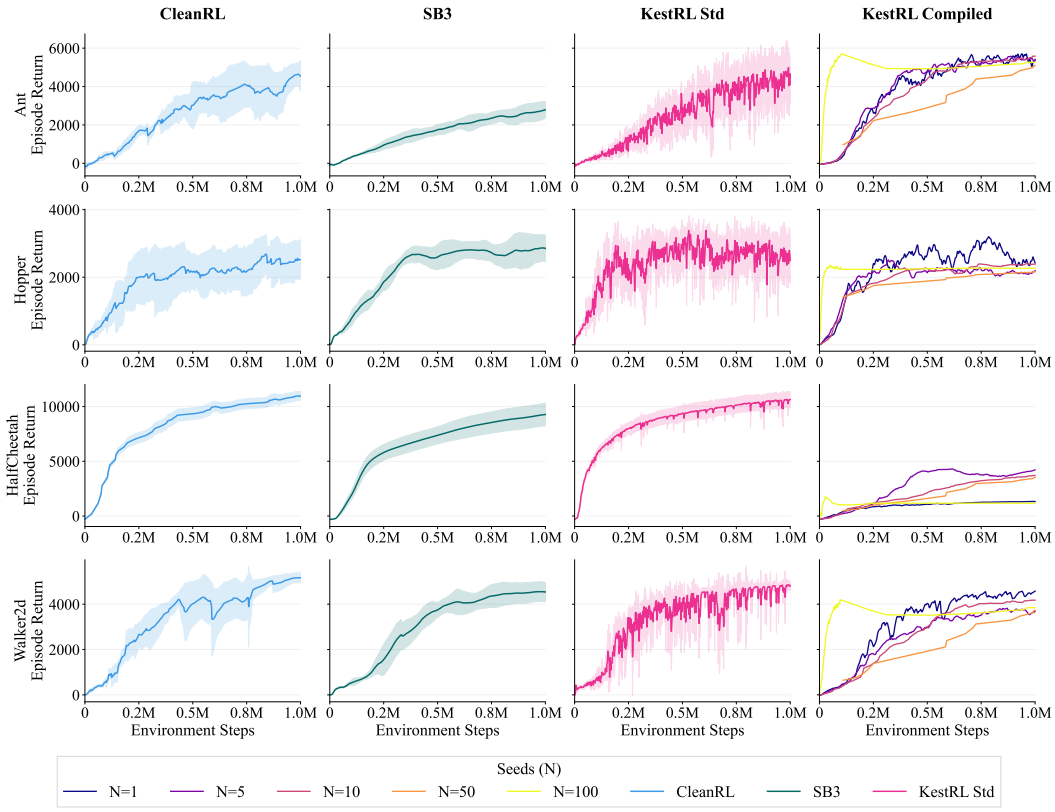


Figure 5: SAC learning across four tasks. CleanRL, SB3 and KestRL Standard use MuJoCo; KestRL Compiled uses Brax. The compiled curves distinguish batches of independent seeds. These traces complement the Ant and Hopper timing comparisons and the Brax learning curves in the main text.

For the H100 batching study, T_N is the time to complete N independent runs and T_N/N is the amortized time per run. Comparing a batch with sequential repetition of its own $N = 1$ configuration gives NT_1/T_N. At $N = 100$, this ratio is approximately 70.8 for KestRL on Ant and 57.6 on Hopper. These ratios describe the benefit of batching relative to extrapolated sequential execution of the same configuration. The comparisons with other libraries use their measured times at the reported batch sizes.

B.4 GPU UTILIZATION DURING SAC TRAINING

The utilization traces in Figure 6 accompany the H100 SAC timing comparisons and show how device activity evolves over a run. CleanRL and SB3 remain at roughly 20–40% GPU utilization on Ant and Hopper. KestRL Compiled, Rejax and Stoix instead spend most of the plotted training period near 100%, with brief interruptions visible in some traces. The contrast is consistent with the execution settings: the former use host-driven MuJoCo environments, whereas the latter combine Brax interaction and learning in compiled computations.

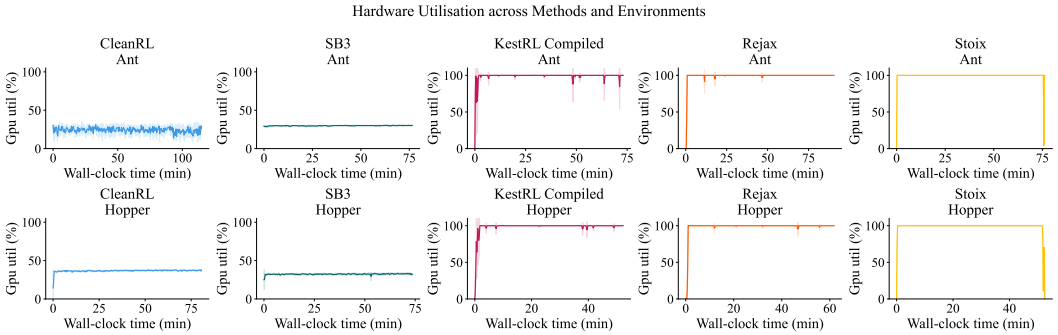


Figure 6: GPU utilization during SAC training on H100. Columns show CleanRL, SB3, KestRL Compiled, Rejax and Stoix; rows show Ant and Hopper. Each trace follows utilization over elapsed wall-clock time. CleanRL and SB3 use MuJoCo, and the three compiled implementations use Brax.

Near-continuous device activity is common to the three compiled implementations, even though their completion times differ. Utilization therefore complements the wall-clock results: it shows that KestRL's use of NNX learning components is compatible with sustained GPU activity, and the timing measurements establish how much training each configuration completes. The batching comparison in Figure 3 then follows the additional work accommodated as the number of independent runs grows.

C LIBRARY ORGANIZATION AND COMPONENT REUSE

The source tree makes the separation in Section 3 useful when locating code to change. A change to a loss or learning schedule begins in the algorithm directory; from there, its implementation draws on the network and storage modules, and its training functions connect to the trainers that manage the experiment. Figure 7 gives a selected view generated from the current repository, including PB-SAC and its posterior distributions alongside the SAC and PPO implementations evaluated here.

```
src/kestrl/
|-- algorithms/
|   |-- base.py
|   |-- sac/
|   |   |-- sac.py
|   |   |-- functions.py
|   |   |-- updates.py
|   |   `-- compiled/
|   |-- ppo/
|   |   |-- ppo.py
|   |   |-- functions.py
|   |   |-- updates.py
|   |   `-- compiled/
|   `-- pbsac/
|-- buffers/
|-- configs/
|   |-- algorithm/
|   |-- environment/
|   |-- experiment/
|   `-- config.yaml
|-- distributions/
|-- environments/
|   |-- builders/
|   |-- wrappers/
|   `-- registry.py
|-- experiments/
|   `-- run.py
|-- networks/
|-- trainers/
|   |-- trainer.py
|   `-- compiled_trainer.py
`-- tests/
```

Figure 7: Selected KestRL source structure. Each algorithm's `compiled/` directory contains its factory, rollout and update orchestration. Package initializers and individual tests are omitted.

For someone changing PPO's objective, the useful starting point is `algorithms/ppo/functions.py`. The minibatch builder in `algorithms/ppo/updates.py` calls those loss functions and constructs the learner-state transition used by both routes. The Standard implementation receives data from host-driven rollouts; the compiled factory constructs its own Brax rollout and schedules the same minibatch update within JAX control flow. A loss change can therefore pass through both implementations without being copied into their collection code.

SAC shares a narrower part of this organization. Its routes use common loss and action-selection helpers, but their learning-operation wrappers and gradient-update loops are separate. The Standard implementation samples from host replay storage; the Compiled implementation carries JAX replay storage through its training iterations. This distinction matters when choosing where a change belongs: modifying a common loss differs from changing when an actor update occurs or how a replay batch is produced.

The environment registry and `experiments/run.py` connect these implementations to the configuration groups in Appendix A. The selected environment determines whether the experiment uses the algorithm object through `Trainer` or the configured factory through `CompiledTrainer`.

D COMPILED SAC ON TPU

The TPU experiments use the same Compiled-route SAC organization with Brax and a budget of one million environment steps per seed. Figure 8 reports the TPU measurements alongside the H100 series. The recorded KestRL batches contain 1, 5 or 10 independent runs; Stoix supplies the $N = 1$ reference. The launcher in `baselines/run_tpu.sh` selects these batch sizes and executes the experiments sequentially on the TPU host.

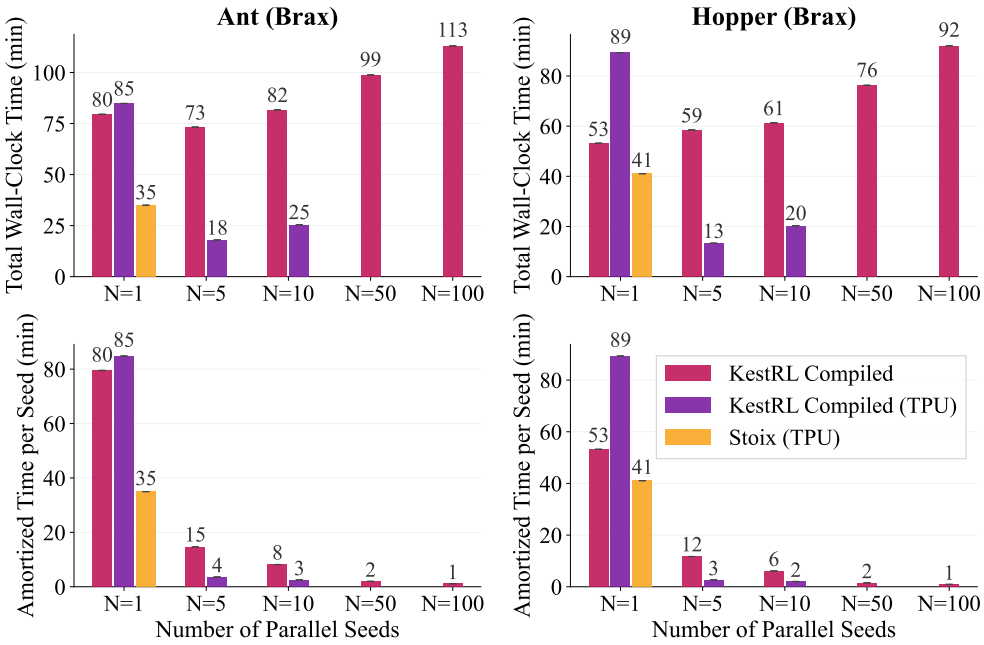


Figure 8: Compiled SAC on TPU and H100, one million environment steps per seed. Upper panels show total batch time and lower panels show amortized time per run. KestRL TPU measurements cover $N \in \{1, 5, 10\}$; Stoix TPU is shown at $N = 1$. The H100 series is repeated from the main text for context.

The TPU comparison changes with batch size. KestRL's single-seed TPU runs take longer than its H100 runs and the Stoix TPU reference, yet its five- and ten-seed TPU batches finish sooner than the corresponding H100 batches. On Ant, for example, total TPU time falls from approximately 85 minutes for one run to 18 minutes for five, or 3.6 minutes per run. Increasing the batch to ten further reduces the amortized time to 2.5 minutes on Ant and 2.0 on Hopper, even though total batch time rises from the five-seed setting.

The total times are non-monotonic: the five-seed batch finishes sooner than the one-seed configuration. Each batch size determines the shapes of a separately compiled program, so this result should not be read as a per-seed runtime extrapolation from $N = 1$. The measurements establish the observed performance of those programs; they do not isolate compilation, device placement or matrix-unit utilization as the cause. Their role here is to show how the Compiled route behaves on another accelerator backend, with the learning and execution-state interfaces retained.