ReCollect: Learned Garbage Collection for Python

Arul Saxena
UC Santa Barbara
saxena@ucsb.edu

Abstract

Garbage collection (GC) in modern language runtimes relies on static, hand-tuned heuristics that are engineered for average-case workloads but cannot adapt to the allocation dynamics of individual applications. We present **ReCollect**, a system that replaces fixed GC trigger policies with a learned control policy trained via tabular Q-learning. ReCollect instruments the CPython and PyPy runtimes to expose heap state as an observation vector, intercepts collection trigger points, and issues GC decisions through a reinforcement learning agent optimized for a user-defined performance objective. We evaluate ReCollect on three realistic, GC-intensive Python workloads: a web server, an LRU cache, and a transaction graph cycle detector. On the web server benchmark, the learned policy achieves over 200% improvement in operations per second relative to the default heuristic; on the LRU cache, it achieves a 25% improvement. Performance is comparable on the graph workload, where the default heuristic already operates near-optimally. These results demonstrate that learned GC is a practically viable technique for performance-critical Python applications, while also surfacing important constraints around overhead, reward design, and cold-start instability that must be resolved before broader deployment.

1 Introduction

Garbage-collected runtimes occupy a central role in modern systems programming, yet their collection policies have changed little in decades. Most production collectors—including CPython's generational collector and PyPy's incremental mark-and-sweep—rely on fixed threshold-based heuristics: collect when heap occupancy exceeds a generation-specific limit, or when the allocation counter since the last collection crosses a predetermined bound [10, 9]. These heuristics are designed to perform acceptably across a wide range of workloads, but their static nature is a fundamental limitation.

Real applications are not average-case workloads. A web server experiences bursty allocation during request bursts and near-idle periods during I/O waits. A graph analytics engine allocates large numbers of short-lived edge objects in tight loops. A caching layer retains long-lived objects that survive many collection cycles. In each case, the optimal GC trigger timing differs—and may vary dynamically within a single application's lifetime. Static heuristics cannot capture this structure.

The insight behind ReCollect is straightforward: GC trigger timing is a sequential decision-making problem, and reinforcement learning provides a principled framework for solving such problems without requiring an explicit model of the environment. Rather than deciding *when* to collect based on hard-coded thresholds, a learned policy observes compact runtime state features and selects GC actions to maximize a user-defined performance objective such as throughput or tail latency. This formulation allows the policy to adapt online to workload structure that static rules cannot express—for example, deferring collections during high-throughput phases and scheduling them during naturally occurring I/O slack.

Preprint. Under review.

Although prior work by Cen et al. [2] introduced the theoretical framework for learned GC and demonstrated its potential in a controlled setting, no public implementation exists. ReCollect closes this gap, providing a working prototype that integrates tabular Q-learning with real Python runtimes. Our contributions are:

- A runtime integration layer for both CPython and PyPy that exposes heap state, intercepts GC decision points, and computes reward signals with low overhead.
- Three GC-intensive benchmark workloads—web server, LRU cache, and transaction graph—designed to surface distinct allocation patterns.
- An empirical evaluation demonstrating that learned policies can yield substantial throughput gains in workloads with exploitable GC slack, with a characterization of the conditions under which learning does and does not help.

Code contributions can be found at: github.com/ajs808/ReCollect

2 Background

2.1 Garbage Collection in Python

CPython manages memory through a combination of reference counting and a cyclic garbage collector [10]. Reference counting reclaims most objects immediately upon deallocation, while the cyclic collector handles reference cycles that reference counting cannot resolve. The cyclic collector is generational, maintaining three generations; objects that survive a collection are promoted to the next generation. Collection of a given generation is triggered when the number of new object allocations since the last collection of that generation exceeds a configurable threshold.

PyPy employs a different strategy: a moving generational collector based on a minimark design [9]. PyPy's collector is more sophisticated than CPython's but equally threshold-driven. Both runtimes expose programmatic interfaces for querying heap statistics and manually triggering or disabling collections, which makes them suitable substrates for intercepting and overriding GC decisions.

The core limitation of both approaches is their inability to condition collection decisions on application-level objectives. The runtime has no notion of whether a collection occurring at a given moment will help or hurt end-to-end throughput—it simply fires when its internal counter trips a threshold.

2.2 Reinforcement Learning and Q-Learning

Reinforcement learning (RL) formalizes sequential decision-making as a Markov Decision Process (MDP) $\langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, P is the transition dynamics, R is the reward function, and $\gamma \in [0, 1)$ is the discount factor [11]. At each timestep t, the agent observes state $s_t \in \mathcal{S}$, selects action $a_t \in \mathcal{A}$, receives reward $r_t = R(s_t, a_t)$, and transitions to s_{t+1}.

Q-learning [14] is a model-free, off-policy RL algorithm that learns an action-value function $Q(s, a)$ estimating the expected discounted return from taking action a in state s and following the optimal policy thereafter. The update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

where α is the learning rate. In the tabular setting, Q is represented as a lookup table over discretized state-action pairs.

Q-learning is well-suited to the GC scheduling problem: it is model-free (no prior knowledge of the program's allocation behavior is required), operates online with low per-step overhead (table lookups and updates rather than neural network inference), and converges to the optimal policy under standard conditions [14, 11].

3 Related Work

Learned Policies for Systems Problems A growing body of work applies reinforcement learning to low-level systems decisions that were historically handled by heuristics. Mao et al. [5] train

neural network policies for cluster job scheduling, demonstrating that learned schedulers can exploit workload-specific structure that hand-tuned heuristics miss. Pensieve [6] applies deep RL to adaptive bitrate video streaming, achieving substantial quality-of-experience improvements over rule-based algorithms. In the memory management domain, Cen et al. [2] introduced the first formal treatment of learned GC, framing it as an MDP and evaluating Q-learning-based policies in a JVM setting. ReCollect directly builds on their formulation, extending it to CPython and PyPy and providing the first open implementation.

Adaptive Memory Management Beyond RL-based approaches, several lines of work have explored dynamic or adaptive memory management. Generational collectors [13] exploit the empirical observation that most objects die young, reducing collection cost by focusing effort on the youngest generation. Concurrent and incremental collectors [3] overlap collection work with mutator execution to reduce pause times. Region-based memory management [12] ties object lifetimes to lexical scope, eliminating collection entirely in some cases. These approaches are complementary to ReCollect: learned GC does not replace the underlying collector but instead learns *when* to invoke it.

Compiler and Runtime Autotuning Learned autotuning has been applied to compiler optimization selection [4], hardware prefetching [1], and database query optimization [7]. These systems share the common thread of replacing hand-designed heuristics with policies learned from workload feedback. ReCollect applies this philosophy to runtime memory management.

4 System Design

4.1 Problem Formulation

We formulate GC scheduling as an MDP. At each decision point—triggered either by a time-based tick or an allocation event—the agent observes the current runtime state, selects a GC action, and receives a reward reflecting the application's performance since the last decision.

State space. The state $s_t \in \mathcal{S}$ is a discrete encoding of runtime metrics that influence the cost and benefit of collection: heap occupancy across generations, the allocation rate since the last collection, and the time elapsed since the last collection. Continuous features are discretized into bins to enable tabular Q-learning.

Action space. The action space is $\mathcal{A} = \{\texttt{noop}, \texttt{collect\_gen0}, \texttt{collect\_gen1}, \texttt{collect\_gen2}\}$, corresponding to deferring collection or triggering a generational sweep at a specified depth.

Reward. The reward r_t is a scalar reflecting the user's performance objective over the interval since the last decision step. In our experiments, we use operations per second (OPS) as the reward signal. Alternative objectives—tail latency, memory footprint—can be substituted by changing the reward computation hook.

4.2 Runtime Integration

ReCollect instruments both CPython and PyPy to expose the above MDP interface. The integration is implemented as a thin Python layer that wraps the standard `gc` module (CPython) and PyPy's `gc` module, respectively.

At each decision step, the integration layer: (1) queries heap statistics via `gc.get_count()` (CPython) or the equivalent PyPy API; (2) serializes the statistics into the discretized state vector; (3) invokes the Q-learning agent to select an action; and (4) either calls `gc.collect(generation)` or passes control back to the mutator. After a configurable interval, the reward is computed and fed into the Q-learning update. The architecture of the learned GC system is illustrated in Figure 1.

4.3 Q-Learning Agent

The agent maintains a sparse Q-table initialized to zero. Actions are selected using an ε-greedy policy: with probability ε a random action is chosen (exploration), and with probability $1 - \varepsilon$ the action maximizing $Q(s_t, \cdot)$ is chosen (exploitation). ε decays over time as the policy converges. Hyperparameters (learning rate α, discount factor γ, initial ε, decay schedule) were tuned per workload.

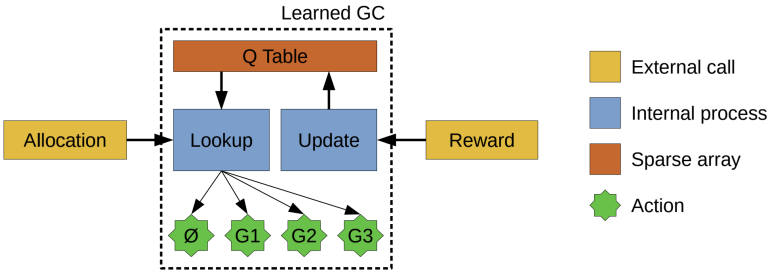

Figure 1: Learned GC architecture. At each allocation event, the agent consults the Q-table to select a GC action (no-op or collect generation n). The reward signal is computed from application-level performance metrics and used to update the table. Adapted from Cen et al. [2].

5 Evaluation

5.1 Workloads

We evaluate ReCollect on three workloads designed to exhibit qualitatively different allocation patterns:

- **Web server.** A simulated HTTP request handler that allocates short-lived response objects at a bursty rate, with I/O-bound intervals between request batches. This workload is designed to expose opportunities for deferring GC to I/O slack periods.
- **LRU cache.** A fixed-capacity LRU cache. The cache retains a mix of long-lived hot entries and rapidly evicted cold entries, creating a moderate but sustained allocation rate.
- **Transaction graph.** A cycle-detection algorithm over a dynamically growing transaction graph. This workload allocates large numbers of medium-lived node and edge objects in short bursts followed by quiet periods.

Each workload was run for 300 seconds. We report operations per second over time for both the learned policy and the default GC heuristic across CPython and PyPy.

5.2 Results

Figure 2 summarizes the steady-state throughput of the learned and baseline policies across all three workloads and both runtimes.

Web server. The learned policy achieves a sustained throughput improvement of over 200% relative to the default GC heuristic on both CPython and PyPy. Post-hoc analysis of the learned policy's action distribution reveals that the agent discovered a deferred-collection strategy: it consistently suppresses GC during high-request-rate phases and schedules collections during I/O-bound intervals where the mutator is blocked. This behavior aligns with the intuition that GC pauses are less costly when they overlap with wait time. Importantly, this strategy cannot be expressed by threshold-based heuristics, which fire based on heap state alone without awareness of I/O scheduling.

LRU cache. On the LRU cache benchmark, the learned policy achieves a 25% improvement in steady-state OPS. The improvement is more modest than in the web server case, consistent with the more uniform allocation rate of this workload. The cache exhibits less I/O slack for the agent to exploit, so the primary lever is timing collections to avoid interrupting runs of hot-path operations.

Transaction graph. On the graph workload, the learned policies perform comparably to the default heuristics, with no statistically meaningful improvement. This result is not surprising: the transaction graph exhibits a predictable allocation pattern with moderate object lifetimes, a setting where threshold-based heuristics are already well-calibrated. The learned policy converges to behavior

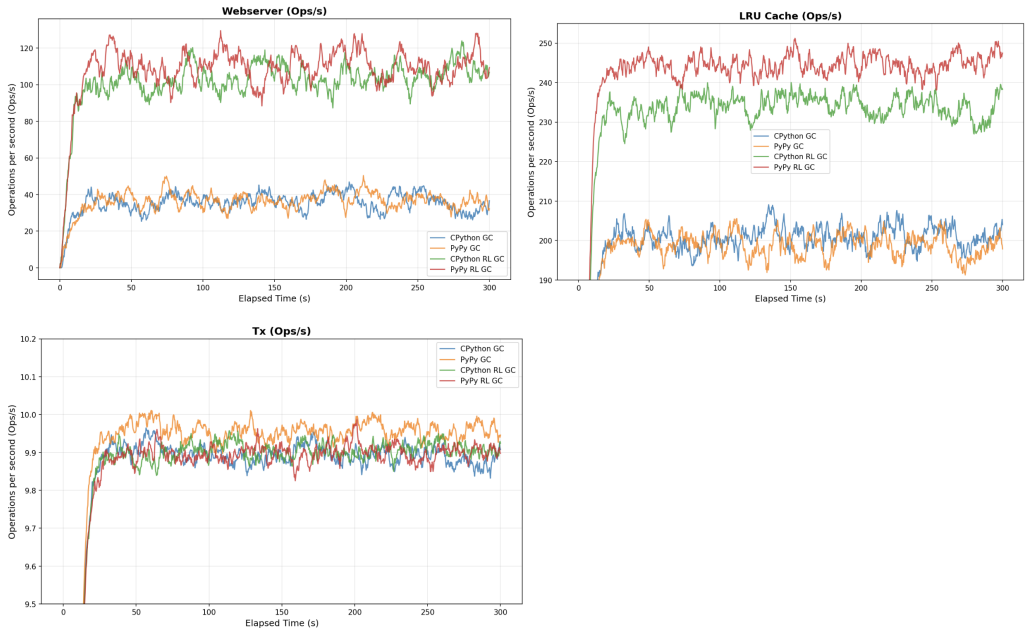

Figure 2: Operations per second vs. elapsed time across the three benchmarks. Solid lines denote baseline GC heuristics; dashed lines denote learned (RL) policies. Results are shown for both CPython and PyPy.

that closely approximates the default, confirming that RL-based control degenerates gracefully when the heuristic is already near-optimal.

6 Discussion

6.1 When Does Learned GC Help?

The evaluation results suggest a clear condition for learned GC to be beneficial: the workload must exhibit structure that the static heuristic cannot exploit. In the web server case, this structure is the temporal correlation between I/O waits and low-cost GC windows. In the LRU cache case, it is the periodic hot-path runs during which a collection would be particularly disruptive. In the graph case, no such structure exists at a scale the heuristic cannot already track.

This pattern mirrors findings from learned systems work more broadly [5, 6]: learned policies outperform heuristics most when the reward landscape has clear structure that simple rules cannot capture, and provide limited benefit in well-conditioned settings where heuristics are already near-optimal.

6.2 Overhead Considerations

Q-learning is computationally lightweight—each decision step requires only a table lookup and, after each step, a single table update. Nevertheless, this overhead is not zero, and for learned GC to yield a net improvement, the performance gain from better-timed collections must exceed the cost of RL computation. In the workloads we tested, this condition is met. However, in workloads with extremely high allocation rates (many GC decision points per second) or narrow performance margins, the overhead budget may be tighter.

6.3 Limitations

Cold-start instability. Online Q-learning requires exploration, which means the agent may issue suboptimal GC decisions during the early phase of execution. In our experiments, the agent converged

quickly and the cold-start period was short, but this is not guaranteed for complex or noisy workloads. An agent that defers GC aggressively while exploring could cause memory pressure or program failure in the worst case.

Reward definition. ReCollect requires the developer to define a reward signal and expose it from the application. In the local benchmarks we evaluate, this is straightforward—OPS is directly measurable. In distributed production systems, application-level metrics such as tail latency or end-to-end throughput may be expensive to measure at the frequency required for per-step RL updates.

Nondeterminism. Reinforcement learning introduces stochastic behavior into a system component—the GC—that is traditionally expected to be deterministic. This complicates debugging and reproducibility, and may conflict with reliability guarantees expected of production runtimes.

Tabular representation limits. The discretized tabular Q-table scales poorly with the dimensionality of the state space. Adding additional state features (e.g., allocation rate per object type, GC pause history) would require exponentially more table entries or a function approximation approach such as deep Q-learning [8].

7 Future Work

Offline and pretrained policies. A natural extension of ReCollect is to decouple training from deployment. Rather than learning online during application execution, policies could be trained offline on representative workload traces and shipped as frozen Q-tables embedded in the runtime. This would eliminate online overhead, avoid cold-start instability, and restore determinism—while still capturing the workload-specific optimization that static heuristics cannot provide.

Hybrid online/offline approaches. A more flexible design would ship a pretrained baseline policy as a warm start and allow the runtime to continue updating it online. This hybrid approach would provide stable initial performance while retaining adaptivity to workload characteristics not captured during offline training—a design pattern common in modern RL deployments [11].

Workload classification. Rather than learning a single universal policy, a runtime equipped with lightweight workload classification could select from a library of pretrained policies based on observed allocation patterns. This multi-policy approach would allow specialized strategies for bursty, steady-state, and memory-intensive regimes without requiring online learning.

Deep RL and richer state representations. Replacing the tabular Q-table with a neural network function approximator would allow ReCollect to condition decisions on richer state features—including object type distributions, GC pause history, and application-level context signals—and to generalize across workloads rather than learning from scratch for each application.

8 Conclusion

ReCollect demonstrates that garbage collection scheduling is a tractable reinforcement learning problem in real Python runtimes. By treating GC trigger decisions as a learned control policy rather than a fixed heuristic, the system can discover workload-specific strategies—such as overlapping collections with I/O waits—that yield substantial throughput improvements in settings where the default heuristic is suboptimal. At the same time, our evaluation surfaces concrete limitations: learned policies provide diminishing returns when heuristics are already well-calibrated, and practical deployment requires addressing cold-start instability, reward instrumentation overhead, and nondeterminism.

Taken together, these findings position learned GC as a promising but targeted optimization: most applicable in performance-critical applications with bursty or structured allocation behavior, and most practically deployed via offline training with optional online fine-tuning. We hope ReCollect serves as a foundation for further exploration of ML-driven runtime systems in Python and beyond.

References

[1] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2011.

[2] Lujing Cen, Ryan Marcus, Hongzi Mao, Justin Gottschlich, Mohammad Alizadeh, and Tim Kraska. Learned garbage collection. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2020, pages 38–44. ACM, 2020.

[3] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2011.

[4] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning–based optimising compilation. In *International Symposium on Code Generation and Optimization*, pages 81–91. IEEE, 2009.

[5] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[6] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Real world performance of adaptive bitrate algorithms. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 15–29. ACM, 2017.

[7] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. In *Proceedings of the VLDB Endowment*, volume 12, pages 1705–1718, 2019.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[9] PyPy Project. Garbage collector documentation and configuration. `https://doc.pypy.org/en/latest/gc_info.html`, 2024. PyPy Documentation.

[10] Python Software Foundation. gc — garbage collector interface. `https://docs.python.org/3/library/gc.html`, 2024. Python 3 Standard Library Documentation.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[12] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[13] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM SIGSOFT Software Engineering Notes*, volume 9, pages 157–167. ACM, 1984.

[14] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3–4):279–292, 1992.