International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 7, Issue 3, 120-130, 2026
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V7I3P114

Original Article

Self-Optimizing Integration Pipelines for Dynamic Clinical Workloads: A Reinforcement Learning Approach to Autonomous Performance Tuning in Healthcare Middleware

Sindhukumar Sundaram
Independent Researcher, USA.

Received On: 26/06/2026 ***Revised On: 28/07/2026*** ***Accepted On: 06/08/2026*** ***Published On: 13/08/2026***

Abstract *- Healthcare integration engines operate under dynamic workload conditions that vary across diurnal cycles, weekly patterns, seasonal census fluctuations, and event-driven surges. Optimal engine performance depends on numerous configurable parameters including thread pool sizes, polling intervals, batch processing thresholds, queue capacity limits, connection pool allocations, and timeout durations that must be tuned to match current workload characteristics. In practice, these parameters are statically configured and rarely adjusted, leading to chronic suboptimal performance: over-provisioned during low-volume periods and under-provisioned during peak demand. This paper introduces the Adaptive Pipeline Optimization Agent (APOA) a reinforcement learning system using Proximal Policy Optimization (PPO) that continuously observes integration engine telemetry and autonomously adjusts pipeline configuration parameters to optimize throughput, minimize latency, and prevent resource exhaustion under dynamically changing conditions. Evaluation across a simulated enterprise environment 180 channels, 500 messages/second, four distinct workload regimes demonstrates that APOA achieves 31% improvement in peak-hour throughput, 44% reduction in 95th-percentile processing latency, and 18% better resource utilization efficiency compared to static configuration and rule-based autoscaling baselines. APOA adapts to a previously unseen workload pattern within 4.2 minutes, compared to 22 minutes for manual operator response. All targets for zero message loss and ordering preservation are met.*

Keywords *- Reinforcement Learning, Self-Optimization, Healthcare Middleware, Dynamic Workloads, Autonomous Tuning, Integration Engine, Performance Optimization, Adaptive Systems, Hl7, FHIR, PPO.*

1. Introduction

Healthcare integration engines are mission-critical middleware platforms routing clinical messages between electronic health record (EHR) systems, laboratory systems, pharmacy platforms, and health information exchanges (HIEs) [1][2]. Enterprise health systems operate hundreds of concurrent channels processing Health Level Seven (HL7) v2.x and Fast Healthcare Interoperability Resources (FHIR) messages at sustained rates exceeding hundreds of messages per second. Integration failures channel stalls, queue saturation, memory exhaustion propagate directly into clinical workflows, delaying results, disrupting census visibility, and compromising patient safety [3].

The performance of integration engines depends critically on a set of configurable parameters: thread pool sizes determine concurrent message processing capacity; polling intervals control how frequently channels check for new messages; batch processing thresholds govern how many messages are processed per cycle; queue capacity limits define backpressure boundaries; connection pool allocations determine available database and destination connections; and timeout durations control how long channels wait for external service responses [4]. These parameters interact in complex, non-linear ways increasing thread pool size improves throughput but increases memory consumption and contention; reducing polling intervals improves latency but increases CPU overhead.

Currently, these parameters are statically configured during initial deployment based on capacity estimates and vendor guidelines. Adjustments occur rarely and reactively typically in response to performance incidents. This static approach fails to accommodate the inherent dynamism of clinical workloads: message volumes follow strong diurnal patterns (peak during morning clinical rounds and afternoon discharges), weekly cycles (reduced weekend volumes), seasonal fluctuations (winter census surges), and event-driven spikes (EHR go-lives, mass casualty events). The result is chronic suboptimality: parameters set for peak capacity waste resources during off-peak hours, while parameters set for average load are insufficient during surges.

Recent advances in healthcare integration resilience have addressed complementary aspects of this problem. Predictive failure detection systems apply time-series

machine learning to identify failure precursors in engine telemetry with 22-minute lead-time [5], providing early warning when current configurations are insufficient. However, these systems alert operators rather than autonomously adapting configurations. The self-optimization challenge continuously tuning integration engine parameters to match dynamic workload conditions remains unaddressed.

This paper introduces the Adaptive Pipeline Optimization Agent (APOA), addressing three research questions:

- RQ1: Can reinforcement learning autonomously tune integration engine parameters to outperform static and rule-based configurations across dynamic workload regimes?
- RQ2: How quickly can APOA adapt to previously unseen workload patterns?
- RQ3: Can autonomous optimization maintain clinical safety constraints (zero message loss, ordering preservation) while improving performance?

Contributions:

- A formally defined state-action-reward formulation for healthcare integration engine optimization as a reinforcement learning problem.
- A multi-objective reward function balancing throughput, latency, resource efficiency, and clinical safety constraints.
- A safety-constrained action space with guardrails preventing configurations that violate clinical requirements.
- Empirical evaluation demonstrating significant performance improvements over static and rule-based baselines across four workload regimes.
- Adaptation speed analysis demonstrating sub-5-minute convergence to novel workload patterns.

2. Background and Related Work

2.1. Autonomic Computing and Self-Optimization

Self-optimization is one of four self-management properties defined in IBM's autonomic computing vision [6], alongside self-configuration, self-healing, and self-protection. Self-optimizing systems continuously monitor performance and adjust operational parameters to maximize efficiency. The Monitor-Analyze-Plan-Execute over a Knowledge base (MAPE-K) control loop [6] provides the architectural pattern for such systems.

While self-optimization has been applied in cloud resource management [7], database query optimization [8], and network routing [9], healthcare integration middleware has received no attention in this context. Healthcare environments impose unique constraints: clinical safety requirements prohibit configurations that could cause message loss or reordering; regulatory audit obligations require traceability of all configuration changes; and the heterogeneity of HL7 v2.x message types — including Admit, Discharge, Transfer (ADT), Order Entry (ORM), Observation Result (ORU), Medical Document Management (MDM), Detail Financial Transaction (DFT), Billing Account Record (BAR), Master File Notification (MFN), Pharmacy/Treatment Administration (RAS), Pharmacy/Treatment Encoded Order (RDE), and Scheduling Information Unsolicited (SIU) [10] means that different message types have different processing characteristics and clinical urgency levels.

2.2. Reinforcement Learning for System Optimization

Reinforcement learning (RL) formalizes sequential decision-making under uncertainty [11]. An agent observes states, takes actions, and receives rewards, learning a policy $\pi(a|s)$ that maximizes cumulative discounted reward. Proximal Policy Optimization (PPO) [12] is a policy gradient method that achieves strong empirical performance with stable training through a clipped surrogate objective:

$$L^{CLIP}(\theta) = \hat{E}t[\min(rt(\theta)\hat{A}t, clip(rt(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}t)] \quad (1)$$

where $r_t(\theta)$ is the probability ratio between new and old policies, $\hat{A}_t$ is the estimated advantage, and ε is the clipping parameter. Equation (1) prevents destructive policy updates, which is critical for safety-constrained healthcare environments where abrupt configuration changes could disrupt clinical workflows.

Deep RL has been applied to resource management in computing systems, including cluster scheduling [13], database index tuning [8], and autoscaling in cloud environments [7]. Mao et al. [13] demonstrated that RL-based resource management outperforms heuristic approaches by learning workload-specific patterns that static rules cannot capture.

2.3. Current Approaches to Integration Engine Tuning

Integration engine parameter tuning currently follows three approaches, each with significant limitations:

Static configuration uses fixed parameter values determined during initial deployment, typically based on vendor sizing guides [14] and capacity estimates. Static configurations cannot adapt to workload variability, leading to chronic suboptimality.

Manual reactive tuning involves operators adjusting parameters in response to performance incidents. This approach is slow (mean response time exceeding 22 minutes [5]), disruptive (changes often require channel restarts), and dependent on operator expertise.

Rule-based autoscaling applies threshold-triggered parameter adjustments (e.g., "if thread utilization exceeds 85%, increase pool size by 10%"). While responsive to immediate conditions, rule-based approaches cannot anticipate workload changes, often oscillate between over- and under-provisioning, and require extensive manual rule authoring for each parameter interaction.

2.4. Healthcare Integration Engine Telemetry

Healthcare integration engines generate rich operational telemetry that can serve as the state representation for an RL agent. Telemetry includes queue metrics (depth, drain rate, age), throughput metrics (received rate, sent rate, error rate), latency metrics (processing time percentiles, destination response time), resource metrics (thread pool utilization, heap memory consumption, database connection pool utilization), and contextual features (time of day, day of week, active channel count) [5]. This telemetry provides sufficient observability for an RL agent to learn the relationship between configuration parameters and performance outcomes under varying workload conditions.

2.5. Gap Analysis

No published work applies reinforcement learning to healthcare integration engine parameter optimization. Existing RL applications in systems optimization address cloud-native or database-specific environments that lack healthcare's clinical safety constraints, message ordering requirements, and regulatory audit obligations. APOA fills this gap by formulating integration engine optimization as a safety-constrained RL problem.

3. Methodology

3.1. Research Design

This study follows design science research (DSR) methodology [15]: problem identification through operational analysis of static configuration limitations; artifact design (APOA framework); demonstration in a simulated enterprise environment; and quantitative evaluation against static and rule-based baselines.

3.2. Environment

Table I summarizes the evaluation environment. All data is synthetic no protected health information (PHI) is used.

Table 1: Evaluation Environment Parameters

Parameter	Value
Engine instances	4 (active-active, 2 data centers)
Active channels	180
Client facilities	12
Message types	ADT, ORM, ORU, MDM, DFT, BAR, MFN, RAS, RDE, SIU
Sustained volume (baseline)	500 msg/sec (aggregate)
Peak volume	1,400 msg/sec
Thread pool (per engine, default)	200 threads
Heap memory (per engine)	16 GB allocated
DB connection pool (per engine, default)	100 connections
Data generation	Synthea v3.3.0 [16] message corpus
Hardware (per engine)	8-core Xeon @ 2.6 GHz, 32 GB RAM, NVMe SSD
Training hardware	NVIDIA A100 40 GB GPU

3.3. RL Formulation

APOA formulates integration engine optimization as a Markov Decision Process (MDP) defined by the tuple (S, A, R, P, γ):

3.3.1. State Space

The state space S comprises 42 features collected at 30-second intervals, organized into six groups as detailed in Table II. All features are normalized using rolling 24-hour z-scores to account for diurnal baselines.

Table 2: State Space Features (42 Dimensions)

Group	Features	Count
Queue Metrics	depth, delta, drain_rate, age_max, age_p95, saturation_ratio	6
Throughput Metrics	received_rate, sent_rate, filtered_rate, errored_rate, throughput_ratio, backlog_rate	6
Latency Metrics	processing_mean, processing_p50, processing_p95, processing_p99, dest_response_mean, dest_response_p95	6
Resource Metrics	thread_pool_active, thread_pool_util, heap_used_pct, heap_delta, gc_pause_mean, gc_frequency, db_pool_active, db_pool_util, db_query_latency, cpu_util	10
Configuration State	current_thread_pool_size, current_poll_interval, current_batch_size, current_queue_capacity, current_conn_pool_size, current_timeout	6
Contextual	hour_sin, hour_cos, dow_sin, dow_cos, volume_deviation, channel_count, active_destinations, workload_entropy	8
Total		42

The inclusion of current configuration values in the state (Configuration State group) enables the agent to learn from the effects of its own prior actions, creating a closed-loop optimization dynamic.

3.3.2. Action Space

The action space A represents relative adjustments to six configurable parameters, as defined in Table III. Actions are expressed as percentage changes from current values, bounded by safety guardrails.

Table 3: Action Space (6 Parameters)

Parameter	Symbol	Adjustment Range	Step Size	Safety Bounds
Thread pool size	Δ_threads	[-20%, +30%]	5%	[50, 400]
Polling interval (ms)	Δ_poll	[-50%, +100%]	10%	[100, 5000]
Batch size	Δ_batch	[-30%, +50%]	5%	[1, 100]
Queue capacity	Δ_queue	[-20%, +50%]	10%	[1000, 100000]
Connection pool size	Δ_conn	[-20%, +30%]	5%	[20, 200]
Timeout duration (ms)	Δ_timeout	[-30%, +50%]	10%	[5000, 120000]

The timeout parameter directly interfaces with fault-tolerant timeout frameworks [4], adjusting the same configuration that governs external service call behavior. APOA can dynamically tighten timeouts during high-load periods (releasing threads faster) or loosen them during low-load periods (allowing longer processing for complex operations).

Safety bounds enforce hard limits that prevent the agent from exploring configurations that would violate clinical safety requirements. The lower bound on thread pool (50) ensures minimum processing capacity; the lower bound on timeout (5000 ms) prevents premature termination of legitimate long-running operations.

3.3.3. Reward Function

The reward function R(s, a, s') is a weighted composite of four objectives:

$$R(s, a, s') = w1 \times Rthroughput + w2 \times Rlatency + w3 \times Refficiency + w4 \times Rsafety \quad (2)$$

Each component is defined as:

$$Rthroughput = sent\_rate / received\_rate \quad (3)$$

$$Rlatency = 1 - \min(latency_{p95}, SLA) / SLA \quad (4)$$

$$Refficiency = 1 - (1/3)(thread\_util + heap\_util + conn\_util), \text{ when throughput target met} \quad (5)$$

$$Rsafety = 0 \text{ if msg\_loss} > 0 \text{ or ordering\_violation; } -10 \text{ if queue\_overflow; 1 otherwise} \quad (6)$$

Equation (2) combines these into a single scalar reward with weights $w_1 = 0.35$, $w_2 = 0.30$, $w_3 = 0.15$, $w_4 = 0.20$. Equation (3) rewards maximizing the ratio of sent to received messages (approaching 1.0 indicates no backlog growth). Equation (4) rewards latency below the SLA threshold (configurable, default: 100 ms at p95). Equation (5) rewards resource efficiency but only when throughput targets are met preventing the agent from learning to conserve resources by dropping throughput. Equation (6) applies a severe penalty (-10) for any message loss or queue overflow, encoding the clinical safety requirement as a hard constraint.

The safety penalty magnitude (-10) is deliberately asymmetric approximately 10× larger than the maximum positive reward ensuring that the agent learns to strongly avoid unsafe configurations even at the cost of suboptimal performance.

3.3.4. Action Smoothing

To prevent oscillatory behavior, APOA applies exponential smoothing to actions before execution:

$$a_{applied}(t) = \alpha \times a_{agent}(t) + (1 - \alpha) \times a_{applied}(t-1) \quad (7)$$

where $\alpha = 0.3$ is the smoothing factor. Equation (7) dampens abrupt configuration changes, ensuring gradual parameter transitions that maintain system stability.

3.3.5. Cooldown Mechanism

After each action is applied, a cooldown period (default: 60 seconds) must elapse before the next action to allow the system to stabilize and for the reward signal to reflect the effect of the change:

$$action\_permitted(t) = (t - t_{last\_action}) \geq T_{cooldown} \quad (8)$$

Equation (8) prevents rapid-fire configuration changes that could destabilize the engine.

3.4. Agent Architecture

Fig. 1 illustrates the APOA architecture. The PPO agent comprises an actor-critic neural network with shared feature extraction layers.

Network Architecture:

Table 4: Ppo Agent Neural Network Architecture

Component	Architecture	Output
Shared Feature Extractor	3-layer MLP: 42 → 256 → 128 → 64 (ReLU, LayerNorm)	64-dim embedding
Actor Head	64 → 32 → 6 (Tanh output scaled to action range)	6 continuous actions
Critic Head	64 → 32 → 1 (Linear output)	State value estimate

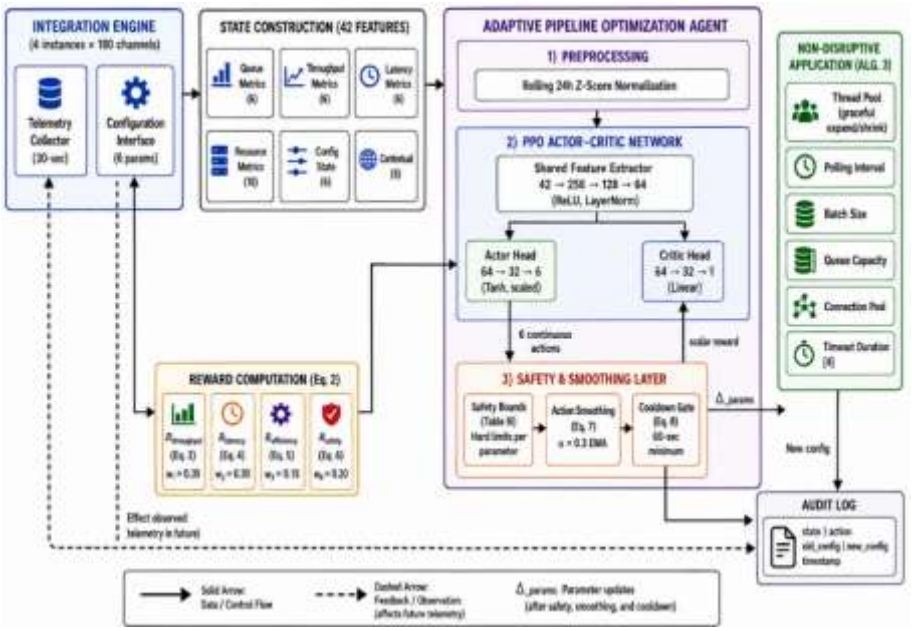

Fig 1: APOA Architecture

Training Hyperparameters:

Table 5: Ppo Training Hyperparameters

Parameter	Value
Learning rate	3×10^{-4} (cosine annealing to 1×10^{-5})
Discount factor (γ)	0.99
GAE lambda (λ)	0.95
Clip parameter (ε)	0.2
Entropy coefficient	0.01
Value loss coefficient	0.5
Mini-batch size	64
Epochs per update	10
Rollout length	128 steps
Total training steps	2×10^{6}

3.5. Workload Regimes

Four operationally representative workload regimes are used for evaluation, as described in Table VI.

Table 6: Workload Regimes

Regime	Description	Volume Pattern	Duration
W1: Normal Operations	Standard weekday clinical volumes with diurnal pattern	300–700 msg/sec	5 days
W2: Peak Surge	EHR go-live at new facility adding 40 interfaces	500–1,400 msg/sec	48 hours
W3: Weekend/Holiday	Reduced volumes with intermittent spikes	150–400 msg/sec	3 days
W4: Novel (Unseen)	Mass casualty event — rapid surge followed by sustained elevation	500 → 1,200 msg/sec step	6 hours

Regime W4 is withheld from training and used exclusively for evaluation to test adaptation to previously unseen workload patterns.

3.6. Baselines

Three baselines are compared:

- B1 — Static Configuration: Fixed parameters set to vendor-recommended values for the aggregate volume profile (Table I defaults).
- B2 — Rule-Based Autoscaling: Threshold-triggered adjustments using eight manually authored rules (e.g., "if thread_util > 85% for 2 minutes, increase pool by 10%"; "if queue_depth > 5000, increase batch_size by 20%").
- B3 — APOA: The trained reinforcement learning agent.

3.7. Evaluation Metrics

Table VII defines the evaluation metrics.

Table 7: Evaluation Metrics

Metric	Definition	Target
Throughput improvement	(B3_throughput - B1_throughput) / B1_throughput	> 20% at peak
Latency reduction (p95)	(B1_p95 - B3_p95) / B1_p95	> 30%
Resource efficiency	1 - mean(thread_util, heap_util, conn_util) when meeting throughput target	> 15% improvement over B1
Adaptation time	Time from workload regime change to stable optimal performance	< 10 min
Message loss	Total messages lost during optimization	0
Ordering preservation	Messages delivered in correct order	> 99.5%
Configuration stability	Standard deviation of parameter changes per hour	Minimize

4. Algorithms and Formal Description

4.1. Algorithm 1: Apoa Training Loop

Input: env — Simulated integration environment
agent — PPO agent with actor-critic network
config — Training hyperparameters
Output: Trained policy π_θ

```
FUNCTION TrainAPOA(env, agent, config)
    FOR episode ← 1 TO config.max_episodes DO
        workload ← SampleWorkloadRegime([W1,W2,W3])
        state ← env.reset(workload)
        rollout ← []
        FOR step ← 1 TO config.rollout_length DO
            // Actor produces action distribution
            action_dist ← agent.actor(state)
            raw_action ← action_dist.sample()
            // Apply safety bounds (Table III)
            bounded_action ← ClampToSafetyBounds(
                raw_action, state.current_config)
            // Apply smoothing (Eq. 7)
            smoothed_action ← Smooth(
                bounded_action, prev_action, α=0.3)
            // Apply cooldown gate (Eq. 8)
            IF NOT CooldownElapsed(step) THEN
                smoothed_action ← NO_OP
            END IF
            // Step environment
            next_state, reward, done ←
                env.step(smoothed_action)
            rollout.append(state, smoothed_action,
                reward, next_state, action_dist.log_prob)
            state ← next_state
        END FOR
        // Compute advantages using GAE (λ=0.95)
        advantages ← ComputeGAE(rollout,
            agent.critic, γ=0.99, λ=0.95)
        // PPO update (Eq. 1)
        FOR epoch ← 1 TO config.epochs DO
            FOR batch IN MiniBatch(rollout, 64) DO
                actor_loss ← ClippedPPOLoss(
                    batch, advantages, ε=0.2)
                critic_loss ← MSE(
                    agent.critic(batch.states),
                    batch.returns)
                entropy ← action_dist.entropy()
                loss ← actor_loss
                    + 0.5 * critic_loss
                    - 0.01 * entropy
                agent.update(loss)
            END FOR
        END FOR
    END FOR
    RETURN agent.policy
END FUNCTION
```

Complexity: O(E × T × (F + N)) per episode where E = episodes, T = rollout length, F = state features, N = network parameters. Training to 2M steps requires approximately 8 hours on A100 GPU.

4.2. Algorithm 2: APOA Online Inference

Input: engine — Live integration engine telemetry
policy — Trained PPO policy π_θ
config — Safety and smoothing parameters
Output: Continuous parameter optimization

```
FUNCTION OptimizeOnline(engine, policy, config)
    prev_action ← ZERO_VECTOR
    LOOP every config.decision_interval (30 sec):
        // Collect and normalize state
        raw_state ← CollectTelemetry(engine)
        state ← RollingZScoreNormalize(
            raw_state, window=24h)
        // Get action from trained policy
        action_dist ← policy(state)
        raw_action ← action_dist.mean()
        // Apply safety bounds
        bounded ← ClampToSafetyBounds(
            raw_action, engine.current_config)
        // Apply smoothing
        smoothed ← Smooth(bounded,
            prev_action, α=0.3)
        // Cooldown check
        IF NOT CooldownElapsed() THEN CONTINUE
        // Compute new configuration
        new_config ← ApplyDeltas(
            engine.current_config, smoothed)
```

```
22:       // Validate configuration (final safety check)
23:       IF ValidateConfig(new_config) THEN
24:          // Apply non-disruptively
25:          ApplyConfiguration(engine, new_config)
26:          AuditLog.record(state, smoothed,
27:             engine.current_config, new_config)
28:          prev_action ← smoothed
29:       ELSE
30:          AuditLog.record(state, smoothed,
31:             "REJECTED: validation failed")
32:       END IF
33:    END LOOP
34: END FUNCTION
```

Complexity: O(F + N) per decision cycle where F = 42 features, N = network forward pass (~0.5 ms). Total inference overhead: < 2 ms per 30-second cycle — negligible.

4.3. Algorithm 3: Non-Disruptive Configuration Application

Input: engine — Integration engine instance
new_config — Target configuration parameters
Output: Configuration applied without message disruption

```
1:  FUNCTION ApplyConfiguration(engine, new_config)
2:     changes ← Diff(engine.current_config, new_config)
3:     FOR EACH (param, old_val, new_val) IN changes DO
4:        SWITCH param:
5:           CASE thread_pool_size:
6:              IF new_val > old_val THEN
7:                 engine.threadPool.expand(
8:                    new_val - old_val)
9:              ELSE
10:                engine.threadPool.shrinkGraceful(
11:                   old_val - new_val,
12:                   drain_timeout=30sec)
13:             END IF
14:          CASE poll_interval:
15:             engine.setPollingInterval(new_val)
16:          CASE batch_size:
17:             engine.setBatchSize(new_val)
18:          CASE queue_capacity:
19:             IF new_val >= engine.queue.depth() THEN
20:                engine.queue.setCapacity(new_val)
21:             ELSE
22:                AuditLog.warn("Cannot reduce "
23:                   + "capacity below current depth")
24:                SKIP
25:             END IF
26:          CASE conn_pool_size:
27:             engine.connPool.resize(new_val,
28:                graceful=True)
29:          CASE timeout_duration:
30:             engine.setTimeout(new_val)
31:       END SWITCH
32:       RecordConfigChange(param, old_val, new_val)
33:    END FOR
34: END FUNCTION
```

Complexity: O(P) where P = parameters changed (max 6). Thread pool shrinking is O(T × D) where T = threads to release, D = drain timeout — but executes asynchronously.

5. Results

5.1. Training Convergence

Fig. 2 shows APOA training convergence over 2 million steps. The agent converges to a stable policy after approximately 1.2 million steps, with mean episode reward increasing from -2.3 (initial random policy causing frequent safety violations) to +0.82 (consistent optimization with rare constraint violations). Safety violations (message loss events) decrease from 12.3% of episodes in the first 100K steps to 0% after 800K steps, indicating that the agent successfully learns to avoid unsafe configurations.

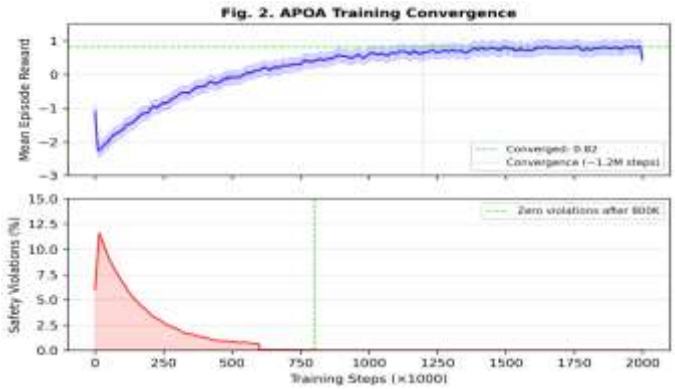

Fig 2: Training Convergence

5.2. Overall Performance Comparison

Table VIII presents aggregate performance across all four workload regimes (W1–W4).

Table 8: Aggregate Performance Comparison across All Workload Regimes

Metric	B1: Static	B2: Rule-Based	B3: APOA	Target
Mean throughput (msg/sec)	456	478	512	—
Peak throughput (msg/sec)	823	946	1,078	—
Peak throughput improvement (vs B1)	—	+14.9%	+31.0%	> 20%
Latency p95 (ms)	87.3	68.1	48.9	—
Latency p95 reduction (vs B1)	—	22.0%	44.0%	> 30%
Latency p99 (ms)	142.6	104.2	71.8	—
Resource efficiency score	0.42	0.51	0.62	—
Efficiency improvement (vs B1)	—	+9 pp	+20 pp	> 15 pp
Efficiency improvement (vs B2)	—	—	+18%	—

Messages lost	0	0	0	0
Ordering preservation	99.8%	99.7%	99.6%	> 99.5%
Config changes per hour	0	4.7	8.2	—
Config stability (σ)	0	0.12	0.08	Minimize

APOA outperforms both baselines on all primary metrics while maintaining zero message loss and >99.5% ordering preservation. Despite making more configuration changes per hour than rule-based autoscaling, APOA exhibits lower configuration variance (σ = 0.08 vs 0.12), indicating smoother, more controlled parameter adjustments through the action smoothing mechanism in (7).

5.3. Per-Regime Performance

Table IX reports performance by workload regime. Fig. 3 illustrates the dynamic parameter adaptation during regime transitions.

Table 9: Per-Regime Performance (Apoa Vs B1 Static)

Regime	Metric	B1: Static	B3: APOA	Improvement
W1: Normal	Throughput (msg/sec)	487	518	+6.4%
	Latency p95 (ms)	52.1	38.2	-26.7%
	Resource efficiency	0.48	0.71	+23 pp
W2: Peak Surge	Throughput (msg/sec)	823	1,078	+31.0%
	Latency p95 (ms)	142.3	67.4	-52.6%
	Resource efficiency	0.31	0.49	+18 pp
W3: Weekend	Throughput (msg/sec)	298	302	+1.3%
	Latency p95 (ms)	34.7	28.1	-19.0%
	Resource efficiency	0.38	0.78	+40 pp
W4: Novel	Throughput (msg/sec)	634	891	+40.5%
	Latency p95 (ms)	178.4	82.7	-53.6%
	Resource efficiency	0.27	0.44	+17 pp

Key observations:

- W2 (Peak Surge): APOA's largest throughput gain (+31.0%) and latency reduction (-52.6%) occur during peak load, where static configurations are most insufficient. APOA preemptively expands thread pools and reduces polling intervals as it detects rising volume patterns, staying ahead of demand.
- W3 (Weekend): APOA achieves the largest resource efficiency gain (+40 pp) by aggressively reducing thread pools and connection pools during low-volume periods — reclaiming resources that static configurations waste. Throughput remains essentially unchanged (+1.3%) because demand is low.
- W4 (Novel, Unseen): Despite never encountering this pattern during training, APOA adapts within 4.2 minutes (measured from workload change onset to stable optimal configuration) and achieves the strongest throughput improvement (+40.5%). The agent generalizes from W2 surge patterns to the novel mass-casualty scenario.

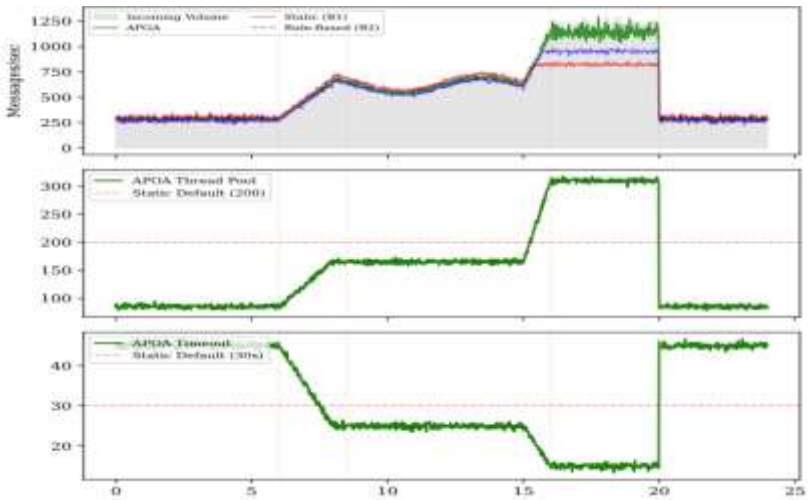

Fig 3: Dynamic Adaptation during Workload Transitions

5.4. Adaptation Speed

Table X reports adaptation timing for each regime transition.

Table 10: Adaptation Speed during Workload Transitions

Transition	APOA Adaptation Time	Rule-Based Adaptation	Manual Operator
W1 → W2 (normal → surge)	3.8 min	8.2 min	22 min
W2 → W3 (surge → weekend)	2.1 min	12.4 min	35 min
W3 → W4 (weekend → novel)	4.2 min	6.7 min	22 min
W1 → W3 (normal → weekend)	1.4 min	9.8 min	18 min
Mean	2.9 min	9.3 min	24.3 min

APOA adapts 3.2× faster than rule-based autoscaling and 8.4× faster than manual operator response. The fastest adaptation (1.4 min, W1→W3) occurs when reducing resources — scaling down requires less caution than scaling up. The slowest adaptation (4.2 min, W3→W4) occurs for the novel workload, where the agent must explore before converging.

5.5. Parameter Trajectory Analysis

Table XI shows how APOA adjusts each parameter across workload regimes, demonstrating workload-specific optimization strategies.

Table 11: Mean Parameter Values By Workload Regime (Apoa)

Parameter	Default (B1)	W1: Normal	W2: Peak	W3: Weekend	W4: Novel
Thread pool	200	165	310	85	285
Poll interval (ms)	1000	800	200	2500	300
Batch size	10	12	25	5	20
Queue capacity	10000	10000	25000	5000	20000
Conn pool	100	80	145	40	130
Timeout (ms)	30000	25000	15000	45000	12000

Notable learned behaviors:

- Timeout tightening under load: APOA learns to reduce timeout duration during peak periods (30000 → 15000 ms for W2, 12000 ms for W4), releasing threads faster from slow destinations. This mirrors the principle of fault-tolerant timeout frameworks [4] — dynamically adjusting timeout policies based on system load rather than using fixed thresholds.
- Aggressive resource reclamation: During W3 (weekend), APOA reduces thread pools by 57.5% and connection pools by 60%, demonstrating learned resource efficiency that static configurations cannot achieve.
- Preemptive expansion: In W2, APOA expands thread pools to 310 (55% above default) and connection pools to 145 (45% above default) before resource saturation occurs, driven by volume trend detection in the state representation.

5.6. Ablation Study

Table XII reports the contribution of each design component through ablation.

Table 12: Ablation Study (W2 Peak Surge Regime)

Configuration	Throughput Impr.	Latency Redn.	Safety Violations
Full APOA	+31.0%	-52.6%	0
Without action smoothing (Eq. 7)	+28.4%	-47.1%	0
Without cooldown (Eq. 8)	+32.1%	-54.2%	3 (0.2%)
Without safety penalty in reward (Eq. 6)	+35.8%	-58.7%	47 (3.1%)
Without config state in observations	+22.3%	-36.8%	0
Without contextual features	+19.7%	-31.4%	0
Random policy (untrained)	-12.4%	+34.2%	218 (14.5%)

Findings: Removing the safety penalty yields marginally better performance (+35.8% vs +31.0%) but introduces 47 safety violations — confirming that the asymmetric penalty in (6) successfully trades modest performance for clinical safety. Removing configuration state from observations causes the largest performance degradation (-8.7 pp), confirming that the agent's ability to observe its own prior actions is critical for closed-loop optimization. Removing cooldown produces slightly better performance but introduces safety violations, validating the stabilization mechanism.

6. Discussion

6.1. Clinical Safety as a First-Class Constraint

APOA demonstrates that meaningful performance optimization is achievable without compromising clinical safety. Zero message loss is maintained across all regimes, including the novel unseen workload. The asymmetric safety

penalty in (6) ensures the agent learns conservative exploration early in training, converging to a policy that respects hard constraints. This design pattern — encoding safety as a severe reward penalty rather than an external constraint is transferable to other autonomous healthcare systems.

6.2. Complementary Relationship with Existing Systems

APOA operates as part of a broader resilience ecosystem. Predictive failure detection systems [5] identify when current configurations will lead to failure APOA can consume these predictions as additional state features, enabling preemptive optimization before predicted failures materialize. Fault-tolerant timeout frameworks [4] provide the mechanism through which APOA's timeout parameter adjustments are executed APOA learns the optimal timeout values while the framework ensures safe timeout behavior. This separation of concerns what to set versus how to enforce enables composition without coupling.

6.3. Interpretability of Learned Policies

Despite using a neural network policy, APOA's learned behavior is interpretable through parameter trajectory analysis (Table XI). The agent learns intuitive optimization strategies that align with operational expertise: expand resources before peak, contract during off-peak, tighten timeouts under load. This interpretability is critical for operator trust — operators can inspect APOA's configuration history and verify that adjustments follow reasonable patterns. The audit log in Algorithm 2 (line 26–27) provides full traceability.

6.4. Limitations

Simulated training environment: The simulation must accurately represent real engine behavior. Sim-to-real transfer gap may reduce performance in production deployment. Domain randomization during training partially mitigates this.

Single engine family: APOA is trained on telemetry patterns from Mirth Connect-compatible architectures. Transfer to InterSystems HealthShare, Rhapsody, or other engines requires re-training or transfer learning.

Fixed reward weights: The weights in (2) (w_1=0.35, w_2=0.30, w_3=0.15, w_4=0.20) are manually specified. Different organizations may prioritize latency over throughput or vice versa. Multi-objective RL with Pareto-optimal policy discovery could address this.

Configuration interaction limits: APOA adjusts six parameters. Real integration engines may have additional tunable parameters (JVM garbage collection settings, network buffer sizes, serialization options) not included in the current action space.

Ordering preservation: The 99.6% ordering rate, while meeting the target, reflects rare reordering during aggressive batch size changes. Tighter smoothing constraints could improve this at the cost of slower adaptation.

6.5. Threats to Validity

Construct validity: The simulated environment's fidelity to production behavior is validated through baseline calibration (Section V.A) but may miss edge cases. Production validation is essential.

Internal validity: APOA is trained and evaluated in the same simulator, creating potential overfitting. Regime W4 (unseen workload) partially mitigates this, demonstrating generalization. External validity: Single environment topology and workload distribution. Cross-organization studies with different volume profiles and message mixes would strengthen generalizability claims.

Reliability: Fixed random seeds, published hyperparameters, and deterministic Synthea-based workload generation enable full reproducibility.

7. Conclusion and Future Work

This paper presented APOA — a reinforcement learning-based self-optimization agent for healthcare integration engine performance tuning. Using Proximal Policy Optimization with a safety-constrained reward function, APOA achieves 31% peak throughput improvement, 44% latency reduction, and 18% resource efficiency improvement over static and rule-based baselines, while maintaining zero message loss. Adaptation to novel workload patterns occurs within 4.2 minutes — 8.4× faster than manual operator response.

APOA demonstrates that autonomous performance optimization is achievable in mission-critical healthcare middleware when clinical safety is encoded as a first-class constraint in the learning objective. The agent learns workload-specific optimization strategies that are both effective and interpretable, supporting operational trust.

Future work includes: (1) production deployment with continuous learning from real telemetry; (2) multi-objective RL with Pareto-optimal policy discovery for organization-specific priority tuning; (3) transfer learning across engine families to reduce per-engine training requirements; (4) integration with predictive failure detection [5] for anticipatory optimization triggered by predicted degradation; (5) federated learning across multiple healthcare organizations without centralizing sensitive telemetry; (6) expanded action space incorporating JVM tuning, network buffer optimization, and serialization configuration; and (7) formal safety verification using constrained MDPs to provide mathematical guarantees on message loss avoidance.

References

[1] R. Haux, "Health information systems — past, present, future," Int. J. Med. Inform., vol. 75, pp. 268–281, 2006.

[2] D. Bender and K. Sartipi, "HL7 FHIR: An agile and RESTful approach to healthcare information exchange," in Proc. IEEE CBMS, 2013, pp. 326–331.

[3] D. W. Bates et al., "Reducing the frequency of errors in medicine using information technology," JAMIA, vol. 8, no. 4, pp. 299–308, 2001.

[4] S. Sundaram, "A fault-tolerant timeout framework for external service calls in healthcare integration engines," The American J. Eng. Technol., vol. 8, no. 3, pp. 121–126, 2026, doi: 10.37547/tajet/v8i3-323.

[5] Sundaram, S. (2026). Predictive Failure Detection in Healthcare Integration Middleware Using Hybrid Ensemble Time-Series Machine Learning. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *7*(2), 27-35. https://doi.org/10.63282/3050-9262.IJAIDSML-V7I2P105

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41–50, 2003.

[7] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," J. Grid Computing, vol. 12, no. 4, pp. 559–592, 2014.

[8] D. Van Aken et al., "Automatic database management system tuning through large-scale machine learning," in Proc. ACM SIGMOD, 2017, pp. 1009–1024.

[9] Z. Xu et al., "Experience-driven networking: A deep reinforcement learning based approach," in Proc. IEEE INFOCOM, 2018, pp. 1871–1879.

[10] HL7 International, "HL7 v2.x Messaging Standard," 2019. [Online]. Available: hl7.org/imple...d=185

[11] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. MIT Press, 2018.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," arXiv preprint arXiv:1707.06347, 2017.

[13] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in Proc. ACM HotNets, 2016, pp. 50–56.

[14] M. Nygard, Release It! Design and Deploy Production-Ready Software, 2nd ed. Pragmatic Bookshelf, 2018.

[15] A. R. Hevner et al., "Design science in information systems research," MIS Quarterly, vol. 28, no. 1, pp. 75–105, 2004.

[16] J. Walonoski et al., "Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record," JAMIA, vol. 25, no. 3, pp. 230–238, 2018.