Smarter Autoscaling with Reinforcement Learning

A simple project comparing RL, thresholds, and HPA

Anurag Baskota | anuragbaskotaa@gmail.com

August 26, 2025

Introduction

Autoscaling is one of the most important features in modern cloud systems. It makes application stay responsive under heavy load while also keeping infrastructure under control. Most setups still uses fixed rules like, "add replica when CPU exceeds 70%" or other fixed policies like scaling up when latency goes beyond a preset target.

But workloads are unpredictable. Spikes can arrive suddenly, and static thresholds may react late. I went through a research paper published by Prof. Ilkyeun Ra, Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications, where RL-inspired autoscaler significantly reduced the latency compared with fixed rules or CPU-based HPA.

Over a weekend, I ran a small experiment to test Reinforcement Learning (RL) as a smarter autoscaler. With bursty load, I tested it against a threshold policy and HPA. The RL approach cut slow-down spikes and reduced noisy scaling, leading to steadier speed at lower cost.

What I built (components)

Workload service (Spring Boot). A tiny HTTP app; under more traffic it replies slower.
Load generator. Four repeating 60s phases: 5, 40, 5, then 60 requests/sec.
Rolling metrics. Keeps recent latencies and reports p95.
RL controller. Probes latency, decides *down / hold / up*, and changes replicas.
Baseline controller. Simple thresholds with a short cooldown.
HPA. Kubernetes autoscaler using CPU, not latency.

How the RL autoscaler works

The RL autoscaler behaves like a careful manager that learns on the job. Every few seconds it looks at the current situation (how slow requests are and how many replicas are running), tries a small change, waits a moment for the system to react, measures what happened, and then updates its memory about whether that choice was good or bad. Over time, it repeats the changes that helped and avoids the ones that hurt. The goal is simple: keep latency steady for users while running no more replicas than needed.

Step-by-step

1. **See.** Check the latest p95 latency (is it good, okay, or bad?) and note the current number of replicas.
2. **Choose.** Decide one move: scale down, hold, or scale up. It sometimes picks a random move to learn, but mostly repeats what worked before.
3. **Change.** Apply at most one replica change, staying within the safe minimum and maximum.
4. **Wait.** Give the system a few seconds so the new replica count can take effect.
5. **Measure.** Measure p95 again to see if things improved or got worse.
6. **Learn.** If the change helped (lower latency with sensible cost), remember it as a good choice in this situation. If not, lower its score. Next time in a similar situation, prefer the higher-scoring choice.

How I ran the test

I ran the experiment on a local Kubernetes cluster created with kind (v1.33.1). All components were built and packaged with Java 17, and each autoscaler was evaluated in isolation under identical conditions for about five minutes. To create repeatable pressure, the load generator cycled through four one-minute phases—5, then 40, back to 5, and finally 60 requests per second—then looped this pattern to produce recurring bursts. Throughout each run I recorded three signals: the p95 response time (computed from recent requests), the current replica count reported by the Deployment's scale API, and the number of scaling actions issued. These measurements let me compare how well each approach controlled latency, how quickly it reacted, and how many pods it used on average.

Results

RL controller

- **Average p95:** 129 ms (best)
- **Average replicas:** 1.7 (fewest pods)
- **Peak p95:** 1,876 ms (early cold start spike), then steady

Explanation: The RL controller settled quickly after a short warm-up. Once it had a few cycles of feedback, it kept latency close to target while holding a low replica count. The early spike is expected during cold start and brief exploration; after that, decisions became stable and cost-efficient.

Baseline controller (thresholds)

- Average p95: 200–400 ms
- Average replicas: 3–4
- Reactive after thresholds; some up/down noise

Explanation: The baseline scaler responded only after the latency crossed fixed limits. This made it slower to protect user experience during fast bursts and prone to small oscillations around the thresholds. It also kept more replicas on average, which raises cost without clearly improving latency.

HPA (CPU-based)

- Scaled 1$\rightarrow$2 when CPU $>$ 70%
- Slower reaction (about 15–60s)
- Optimizes CPU, not user latency

Explanation: HPA behaved predictably and remained stable, but it followed CPU rather than direct user-facing metrics. Because CPU lags and does not always track perceived speed, scale-up came later than needed for sharp traffic spikes. As a result, HPA was reliable for resource control but less effective at keeping p95 low.

Side-by-side summary

Metric	RL Controller	Baseline	HPA
What it watches	Latency	Latency	CPU
Avg p95	**129 ms**	200–400 ms	(indirect)
Avg replicas	**1.7**	3–4	2.0
Reaction time	5–10 s	5–10 s	15–60 s
Adapts over time	Yes	No	No
Cost efficiency	High	Medium	Medium

The RL controller focuses on latency and learns over time, which is why it achieves the lowest average p95 with the fewest replicas. By contrast, the baseline is simple but static, and HPA is reliable for CPU control yet indirect for user experience—so both trail RL on responsiveness and efficiency under bursty load.

Conclusion

This test shows that a small RL controller can keep latency lower while using fewer pods than a threshold policy or CPU-based HPA. After a short warm-up, it reacted well to bursts, stayed steady, and avoided extra cost. The baseline was easy to understand but waited for limits to be crossed, and HPA followed CPU, not what users feel. If your service is sensitive to p95 and your traffic is spiky, an RL autoscaler is a practical next step to try.