DYNAMIC MULTI-METRIC THRESHOLDS FOR SCALING APPLICATIONS USING REINFORCEMENT LEARNING

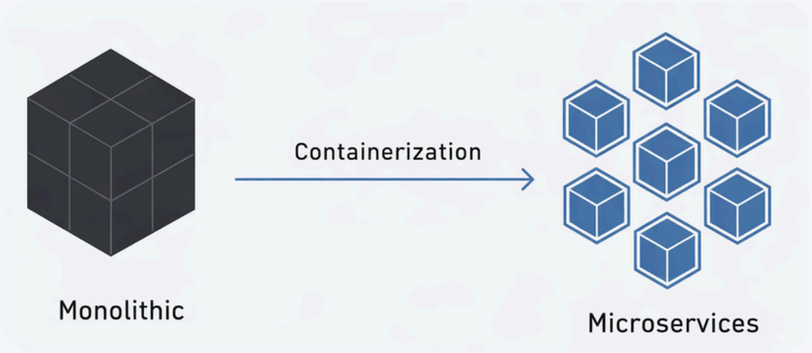
F. Rossi, V. Cardellini, F. Lo Presti, M. Nardelli

Pedicillo Marco 1983285

What's the problem?

Cloud computing has favored the development of elastic applications, capable of dynamically **adapting** their deployment to meet Quality of Service requirements.

In this context, many modern applications have adopted **microservice architectures**, where elasticity is managed through threshold-based auto-scaling policies, often manually defined on a single resource metric such as CPU utilization. However, modern applications are heterogeneous and may be CPU intensive, memory intensive, or mixed, making static single-metric thresholds often ineffective and prone to **Service Level Objective** (SLO) violations or resource waste.



This motivates the need for adaptive, multi-metric scaling solutions that can automatically adjust scaling thresholds based on observed application behavior.

And what's the solution?

The key idea of this work is to overcome these limitations by introducing **self adaptive, multi metric scaling thresholds learned through Reinforcement Learning**.

Instead of directly controlling the number of microservice replicas, the approach uses RL to **dynamically adjust scaling thresholds** for multiple resource metrics.

By learning from runtime feedback and optimizing a cost function, the system dynamically adapts scaling behavior. The cost function balances:

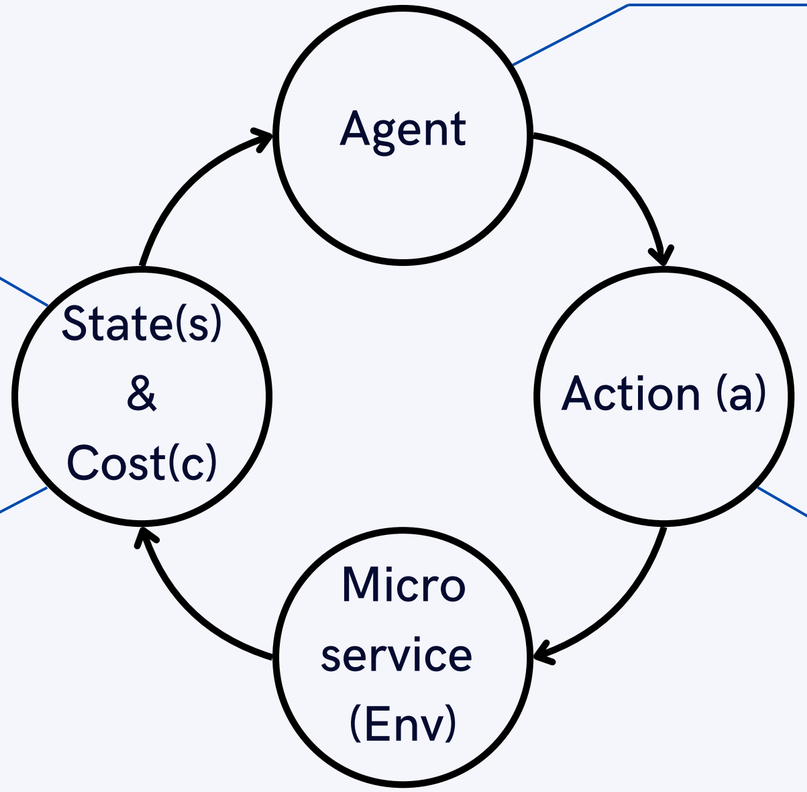
- Performance objectives (e.g., SLO satisfaction)
- Resource efficiency

As a result, the system:

- adapts to heterogeneous microservices and changing workloads
- avoids manual threshold tuning and explicit application modeling

How does it work?

Trial and error:
Learn the optimal strategy by interacting



State:
Current configuration (thresholds, resource usage)

Cost:
Immediate Feedback (SLO vs Resources)

Action:
Change scaling thresholds

- **Single-Agent**: It controls all scaling thresholds, resulting in higher state complexity and slower learning.
- **Multi-Agent**: It controls individual metrics, reducing the state space and improving scalability and execution time.

Cost Function

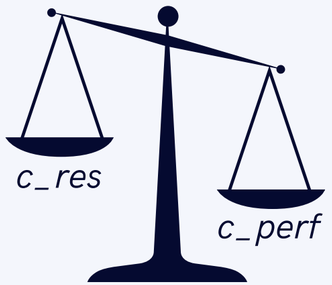
The RL Agent learns an adaptive scaling policy by minimizing an immediate cost function that captures the trade-off between performance and resource efficiency.

$$c(s, a, s') = w_{perf}\, c_{perf}(s, a, s') + w_{res} \cdot c_{res}(s, a, s')$$

$$\text{where } w_{perf} + w_{res} = 1$$

The cost function is composed of the **Performance Cost**, which penalizes SLO violations, such as increased response time, and the **Resource Cost**, which captures inefficient resource usage.

The weights **w_perf** and **w_res** define the relative importance of performance and resource efficiency. By tuning these weights, the system can prioritize either SLO satisfaction or resource cost reduction, depending on deployment objectives.



Q-FUNCTION UPDATE STRATEGIES

- **Model Free RL**: It does not require a priori knowledge of the dynamics of the system because the agent learns from experience alone through trial and error;
- **Model Based RL**: Uses a system estimate or model to predict future transitions, enabling more efficient exploration and accelerating learning.

✳ Q-learning Threshold

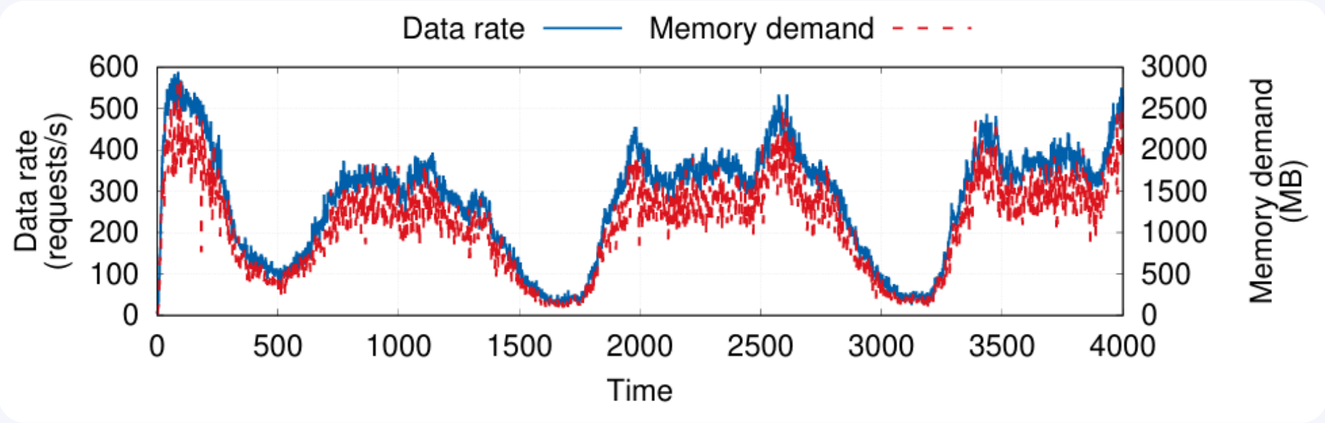
✳ Model-Based Threshold

✳ Deep Q-learning Threshold

	Q-learning	Model-Based	Deep Q-learning
Exploration strategy	ε-greedy	No exploration	ε-greedy
Scalability	Poor	Limited	High
Convergence speed	Slow	Fast	Medium-Fast
Handles large state spaces	No	No	Yes
Main advantage	Simple, no prior knowledge	Fast convergence	Scalable and stable
Main limitation	State-action explosion	Limited scalability	Higher complexity

Experimental Evaluation

The experiments evaluate the proposed strategies under a controlled and realistic workload. To this end, a microservice system is simulated using real **NYC Taxi Ride traces**, which generate realistic fluctuations in CPU and memory demand.
This setup allows analyzing how different RL approaches adapt scaling thresholds over time while optimizing a cost function that balances performance and resource efficiency. By varying the cost function weights, the ability of each strategy to adapt to different deployment objectives is evaluated.



Workload based on NYC Taxi Ride traces

Experimental Evaluation - Strategies Comparisons

Architecture	*Policy*	*Configuration* $\langle w_{\text{perf}}, w_{\text{res}} \rangle$	*Average CPU threshold (%)*	*Average CPU utilization (%)*	*Average Memory threshold (%)*	*Average Memory utilization (%)*	*Median response time (ms)*	$T_{\max}$ *violations (%)*	*Memory violations (%)*	*Average number of replicas*
SA	**MB Threshold**	$\langle 1, 0 \rangle$	50.52	33.56	50.18	32.02	8.35	5.42	0	6.85
		$\langle 0.5, 0.5 \rangle$	68.60	40.07	68.71	38.25	8.43	6.60	0	5.68
		$\langle 0, 1 \rangle$	89.99	50.32	89.98	48.02	9.00	14.07	0.07	4.48
	QL Threshold	$\langle 1, 0 \rangle$	63.75	34.85	67.43	33.27	8.36	6.22	0	6.63
		$\langle 0.5, 0.5 \rangle$	70.71	36.59	71.70	34.93	8.37	6.15	0	6.25
		$\langle 0, 1 \rangle$	75.63	38.81	78.58	37.05	8.42	6.65	0	5.84
	DQL Threshold	$\langle 1, 0 \rangle$	64.50	35.09	52.94	33.47	8.35	6.17	0.02	6.62
		$\langle 0.5, 0.5 \rangle$	86.36	41.49	86.30	39.59	8.41	10.70	0.05	5.64
		$\langle 0, 1 \rangle$	85.98	48.14	87.15	45.95	8.73	12.90	0.05	4.71
	DQL Threshold (pre-trained)	$\langle 1, 0 \rangle$	64.04	33.71	51.08	32.17	8.35	5.50	0	6.83
		$\langle 0.5, 0.5 \rangle$	87.99	42.69	86.67	40.76	8.50	9.65	0.05	5.49
		$\langle 0, 1 \rangle$	88.71	49.75	87.45	47.47	9.00	13.32	0.07	4.55
MA	**MB Threshold**	$\langle 1, 0 \rangle$	50.10	33.58	50.01	32.04	8.35	5.67	0	6.85
		$\langle 0.5, 0.5 \rangle$	67.23	42.43	73.90	40.50	8.54	7.22	0	5.31
		$\langle 0, 1 \rangle$	89.99	50.32	89.99	48.02	9.00	14.07	0.07	4.48
	QL Threshold	$\langle 1, 0 \rangle$	70.38	35.43	68.52	33.82	8.36	6.05	0	6.50
		$\langle 0.5, 0.5 \rangle$	75.56	40.76	82.84	38.92	8.46	7.42	0	5.58
		$\langle 0, 1 \rangle$	86.82	48.03	87.48	45.85	8.81	10.72	0	4.70
	DQL Threshold	$\langle 1, 0 \rangle$	52.50	33.68	52.64	32.14	8.35	5.45	0	6.82
		$\langle 0.5, 0.5 \rangle$	67.24	39.18	66.56	37.39	8.42	6.70	0	5.80
		$\langle 0, 1 \rangle$	73.71	41.62	71.26	39.72	8.51	7.75	0	5.49
	DQL Threshold (pre-trained)	$\langle 1, 0 \rangle$	52.20	33.67	52.62	32.12	8.35	5.55	0	6.83
		$\langle 0.5, 0.5 \rangle$	65.07	38.00	80.03	36.29	8.41	6.95	0	6.01
		$\langle 0, 1 \rangle$	81.92	43.52	78.42	41.55	8.53	10.05	0.02	5.34

Q-Learning:

- Slow adaptation to workload and weight changes
- Less stable scaling behavior in large state spaces
- Sub-optimal thresholds with w_perf=1

Model-Based:

- Fast convergence
- Best control of response time and SLO violations

DQL:

- Performance close to Model-Based
- Better scalability across weight settings

DQL(Pre-Trained):

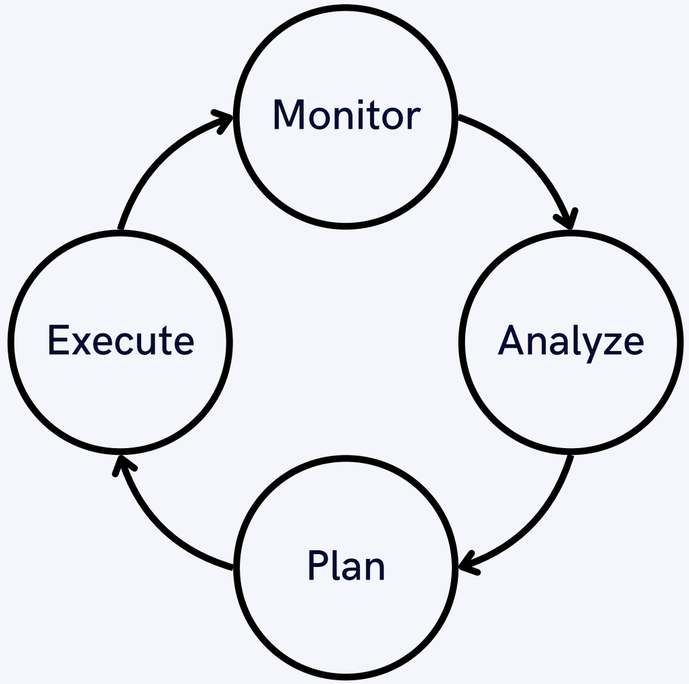
- Faster convergence compared to DQL
- Reduced exploration cost
- More stable thresholds in the early phase

Prototype-Based Experiments

The proposed RL scaling policies are evaluated in a real Kubernetes environment using a custom auto-scaler implemented within a **MAPE** control loop. The experiments are conducted on a Google Cloud cluster and involve two representative microservices: **Pi**, a CPU-intensive service, and **Word Count**, a memory-intensive service. The evaluation focuses on response time, SLO violations, and resource usage, highlighting the benefits of learning-based threshold adaptation.

Benchmarks:

- **Horizontal Pod Autoscaler** (HPA), which relies on static CPU thresholds;
- **HyScale**, which combines horizontal and vertical scaling;
- A model-free Q-learning approach by **Horovitz et al**, which adapts CPU thresholds at runtime.



Prototype-Based Experiments - Results

Application	Architecture	*Policy*	*Average CPU threshold (%)*	*Average CPU utilization (%)*	*Average Memory threshold (%)*	*Average Memory utilization (%)*	*Average Share CPU/Mem (MB)*	*Median response time (ms)*	T_{max} *violations (%)*	*Memory violations (%)*	*Average number of replicas*
Pi	SA	**MB Threshold**	75.96	43.35	76.00	15.59		90.38	6.92	0	2.26
		DQL Threshold	83.05	48.93	67.92	18.43		75.24	8.81	0	1.70
		DQL Threshold (pre-trained)	89.09	49.30	88.80	15.69		85.14	9.43	0	1.74
	MA	**MB Threshold**	79.27	47.66	89.94	15.25		73.66	8.99	0	1.98
		DQL Threshold	52.58	34.30	52.75	14.77		66.81	1.12	0	2.79
		DQL Threshold (pre-trained)	72.78	42.29	82.61	15.09		76.47	5.56	0	2.12
	-	**HPA 80/80**	80.00	49.94	80.00	15.48		82.12	9.59	0	1.56
		HPA 70/70	70.00	47.75	70.00	17.53		72.57	6.03	0	1.61
		HPA 60/60	60.00	43.37	60.00	14.87		70.05	5.46	0	1.70
	-	**HyScale 80/80**	80.00	63.09	80.00	37.27	0.388/100	115.28	47.56	0	1.00
		HyScale 70/70	70.00	58.46	70.00	35.81	0.440/100	82.86	35.00	0	1.02
		HyScale 60/60	60.00	53.49	60.00	36.15	0.511/100	72.42	26.76	0	1.11
	-	**Horovitz et al.**	57.20	34.21	-	15.99		12.19	1.51	0	2.68
Word-count	SA	**MB Threshold**	78.83	26.64	78.71	52.93		6.56	5.65	0	2.69
		DQL Threshold	76.35	33.88	84.29	50.83		8.02	14.18	0.71	2.52
		DQL Threshold (pre-trained)	88.25	42.13	88.59	59.20		17.11	21.25	0.63	1.64
	MA	**MB Threshold**	78.06	27.96	73.41	53.51		6.35	8.24	0	2.56
		DQL Threshold	52.38	21.94	52.86	44.54		6.29	4.76	0	3.40
		DQL Threshold (pre-trained)	68.68	31.08	71.67	53.13		7.68	11.49	0	2.31
	-	**HPA 80/80**	80.00	47.01	80.00	53.03		30.47	30.29	0	1.20
	-	**HPA 70/70**	70.00	28.04	70.00	55.58		6.81	4.29	0	2.23
	-	**HPA 60/60**	60.00	25.33	60.00	47.14		5.91	1.42	0	2.61
	-	**HyScale 80/80**	80.00	51.51	80.00	40.97	0.303/200	35.95	29.07	0	1.00
		HyScale 70/70	70.00	49.96	70.00	39.59	0.352/201	32.88	25.00	0	1.00
		HyScale 60/60	60.00	46.16	60.00	37.03	0.314/200	42.26	32.56	0	1.00
	-	**Horovitz et al.**	58.68	34.34	-	59.17		8.67	9.37	1.56	1.72

Conclusion

The experimental results demonstrate that RL enables dynamic and self-adaptive threshold tuning, eliminating manual configuration and allowing the system to optimize user-oriented objectives such as response time and SLO violations.

From the comparison of the proposed approaches, the following insights emerge:

- **Model-Based Threshold** achieves the best performance when an approximate system model is available, thanks to fast convergence and stable behavior.
- **Deep Q-Learning Threshold** offers a scalable, model-free alternative with performance close to the Model-Based approach, especially with pre-training.
- The **MA Architecture** improves scalability and reduces execution time compared to the SA solution.

Overall, the results confirm that this approach is effective for elastic microservice management, while static threshold-based policies are inadequate for modern cloud-native applications.