ADVANCED RESOURCE MANAGEMENT IN CLOUD

ADVANCED RESOURCE MANAGEMENT IN CLOUD

- Managing resources in modern cloud environments is increasingly complex due to:
  - Dynamic and unpredictable workloads
  - Multi-tenant infrastructures
  - Energy efficiency and cost constraints
- Traditional approaches:
  - Threshold-based or heuristic rules
  - Hard to adapt to changing conditions
  - Often lead to over- or under-provisioning
- Need:
  - A self-adaptive, data-driven strategy that can continuously learn how to make optimal allocation decisions.
- Solution:
  - Reinforcement Learning (RL) — a framework to learn optimal resource management policies from real data and experience.

WHY REINFORCEMENT LEARNING?

- Reinforcement Learning (RL) is a framework for learning by interaction: the agent learns the best actions through trial, feedback, and adaptation.
- Key reasons why RL fits cloud resource management:
  - Sequential decision-making:
    RL optimizes a *sequence* of actions with delayed effects (e.g., scaling now affects future performance).
  - Experience-based learning:
    The agent improves directly from *real operational data*, without requiring a perfect model of the environment.
  - Self-adjusting behavior:
    RL continuously adapts to workload changes, failures, and new conditions.
  - Goal-oriented control:
    The reward function encodes system goals — e.g., minimize cost, maximize performance, preserve SLOs.

LEARNING OPTIMAL ACTIONS: THE TIC TAC TOE EXAMPLE

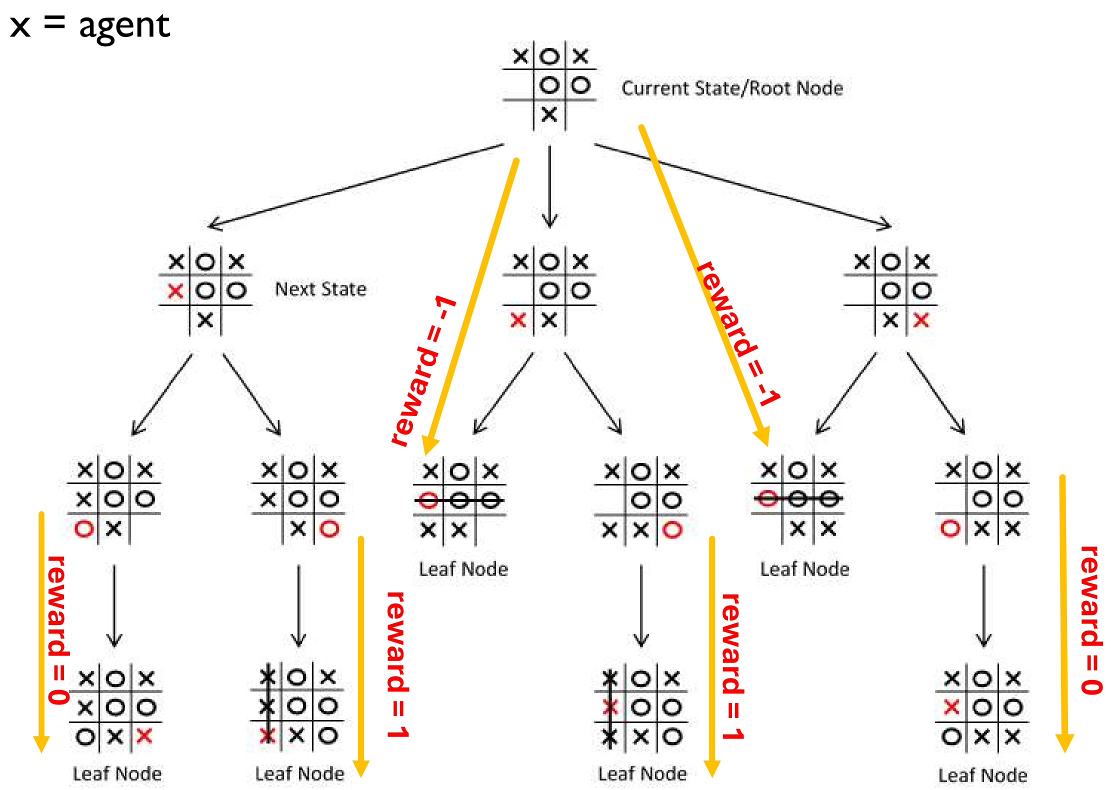
- Let's start with a simple game scenario.
- Goal: Find the *best move* (optimal action) at each turn.
- Concepts:
  - State (s): the current configuration of the game board
  - Action (a): placing a symbol (X or O) in a cell
  - Reward of an action (r): +1 if win, 0 if draw, -1 if loss
  - Total reward (G): sum of the rewards until the end of the game
  - Policy (π): rule that decides which move to take in each state
- Each move changes the future outcome — this makes it a sequential decision problem, just like cloud control.

POLICY EVALUATION

- A policy defines how the agent acts in every state.
- We can measure how good a policy is using the value function:
- $v_\pi(s) = \textit{Expected total reward from state } s \textit{ following policy } \pi$ $\quad v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$
  - A high value → good position for the agent
  - A low → risky or losing position

POLICY EVALUATION

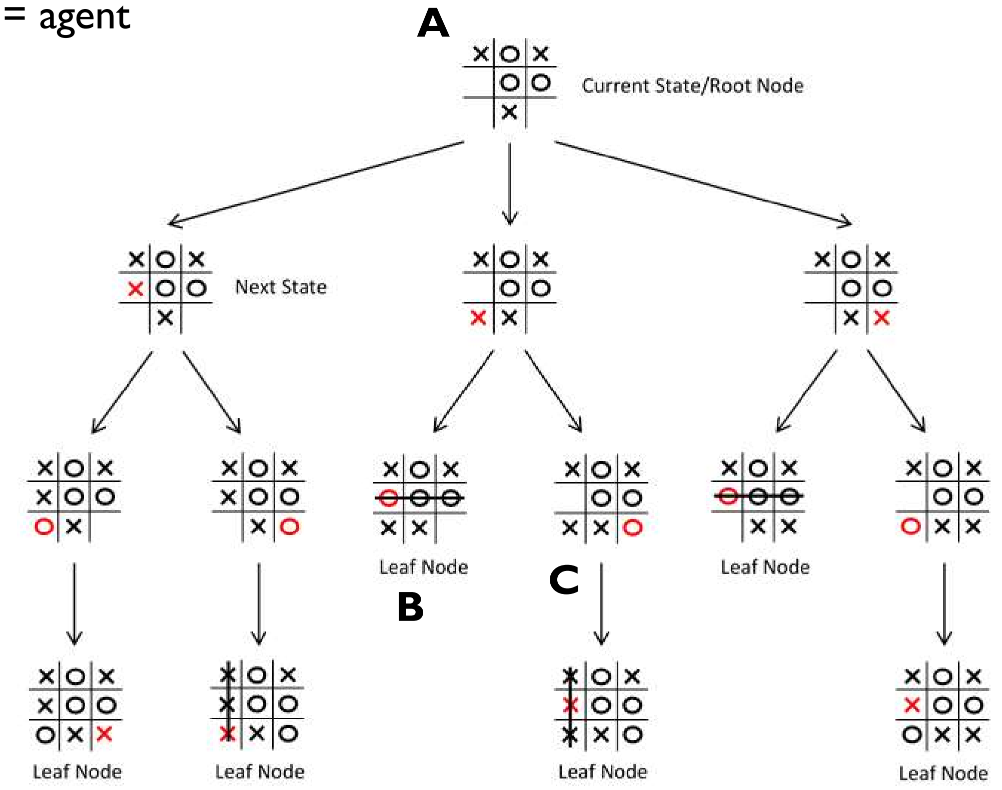
- The immediate reward of intermediate actions is 0



POLICY EVALUATION

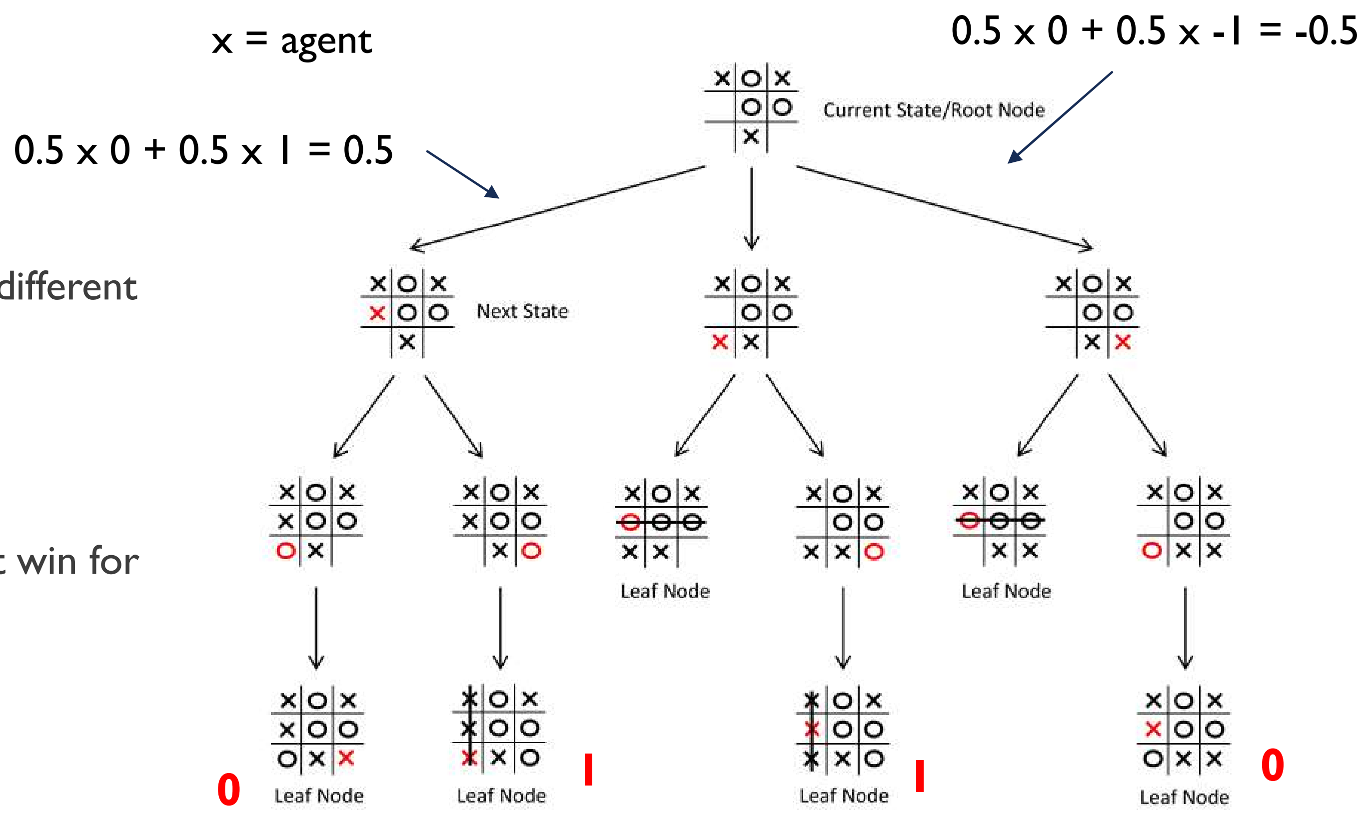
- For example, assume that the policy says that when in state A the agent makes the action a=(0,2) (middle)
- The next state can be B (with probability 0.5) or C with probability 0.5 - depending on the counter move.
- V(A) = 0.5 x -1 + 0.5 x V(C)
- V(C) = 1 (there is one move the agent can do)
- So V(A)=0
- The value of a state represents the average outcome of the game
- V(A)=0 means that on the average (playing many times) the game is a draw

x = agent



POLICY EVALUATION

- Under a different policy the state A has a different value
- If the agent moves in (0,1) then V(A)=0.5
- If the agent moves in (2,2) then V(A)=-0.5
- Q:Which value v(s) would make the agent win for sure once it is in state s?



POLICY EVALUATION: BELLMAN EQUATION

- The value function v(s) must satisfy a consistency relationship because the value of state s is related to the value of the neighboring states s', i.e. the states that can be reached after an action
- $v(s) = \sum_{s'} p(s'|s,a)\ [r + v(s')] = \sum_{s'} p(s'|s,a)\ r + \sum_{s'} p(s'|s,a)\ v(s')$
- Let p(s'|s,a) be the probability that due to the action taken in s, the next state is s' (consider only non-terminal states)
- Since the actions are fixed (given), this defines a Markov Reward Process (MRP)

Reward when moving to terminal states

$$\mathbf{v} = \mathbf{P}\mathbf{v} + \mathbf{r} + \mathbf{t}$$

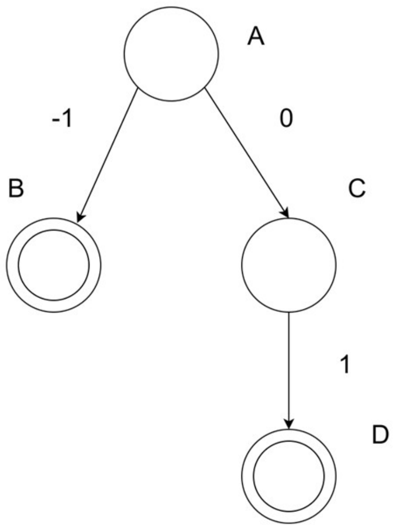
State transition matrix among non-terminal states

Reward vector when moving to other non-terminal states

$$\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{b}$$

EXAMPLE

- Not terminal states = {A,C}
- Terminal states={B,D}
- In our example there no immediate rewards when moving to not terminal state
- Prob{B|A}=0.5, Prob{D|A}=0→ t_A=-0.5
- Prob{B|C}=0,Prob{D|C}=1→t_C=1



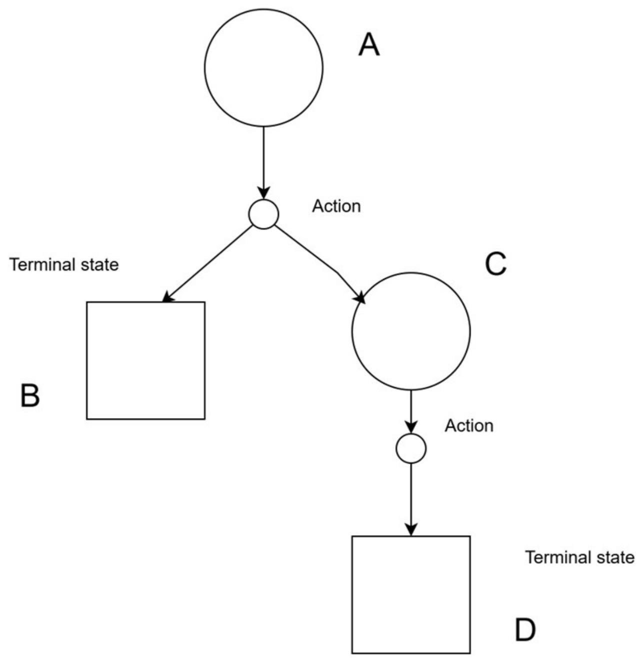
$$\begin{bmatrix} v_A \\ v_C \end{bmatrix} = \begin{bmatrix} 0 & 0.5 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} v_A \\ v_C \end{bmatrix} + \begin{bmatrix} -0.5 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} v_A \\ v_C \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0.5 \\ 0 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} -0.5 \\ 1 \end{bmatrix}$$

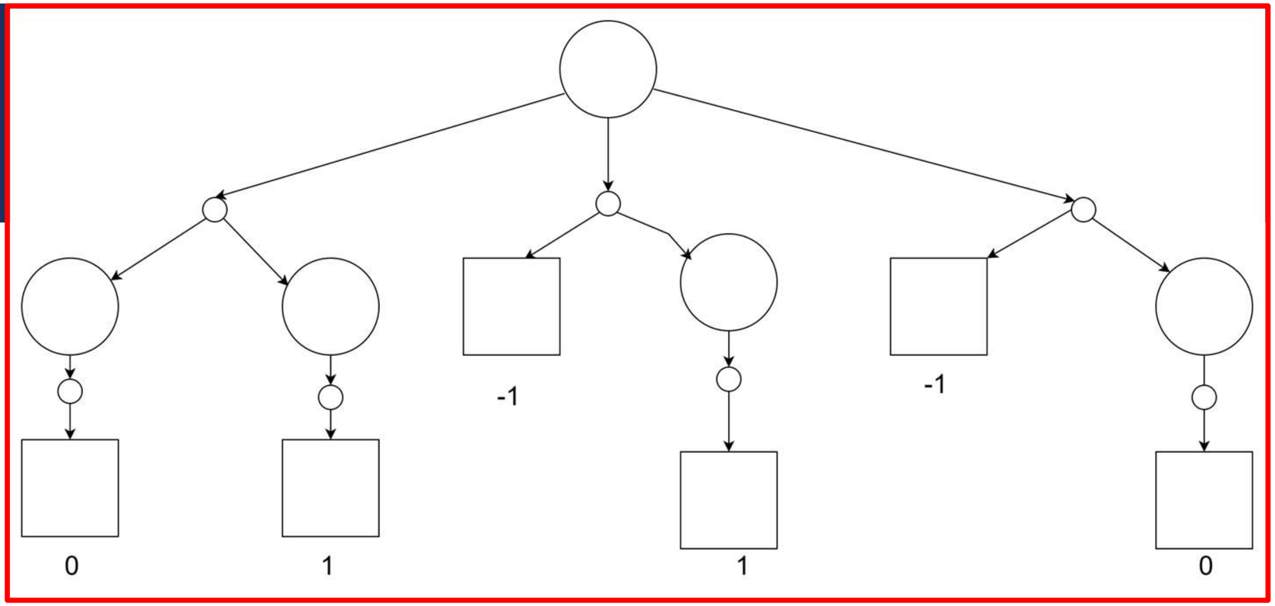
$$v_A = 0, \quad v_C = 1$$

OPTIMAL POLICY

- best sequence of actions that maximizes the expected total reward.
- A key result is that the optimal sequence of actions can be obtained through a greedy evaluation:
  at each step, the agent selects the action that leads to the next state with the highest expected value, according to the current estimate of the value function.
- It is useful to use a graph where actions and terminal states have different symbols



OPTIMAL POLICY



- The maximum value that a state can have, $v_*(s)$, i.e. the maximum expected reward G the agent can obtain by being in that state, obeys the following consistency rule
- $\mathrm{v}_*(\mathrm{s}) = \mathrm{maxa}(\sum_{s'} p(s'|s,a)\, r + \sum_{s'} p(s'|s,a)\, v_*(s'))$

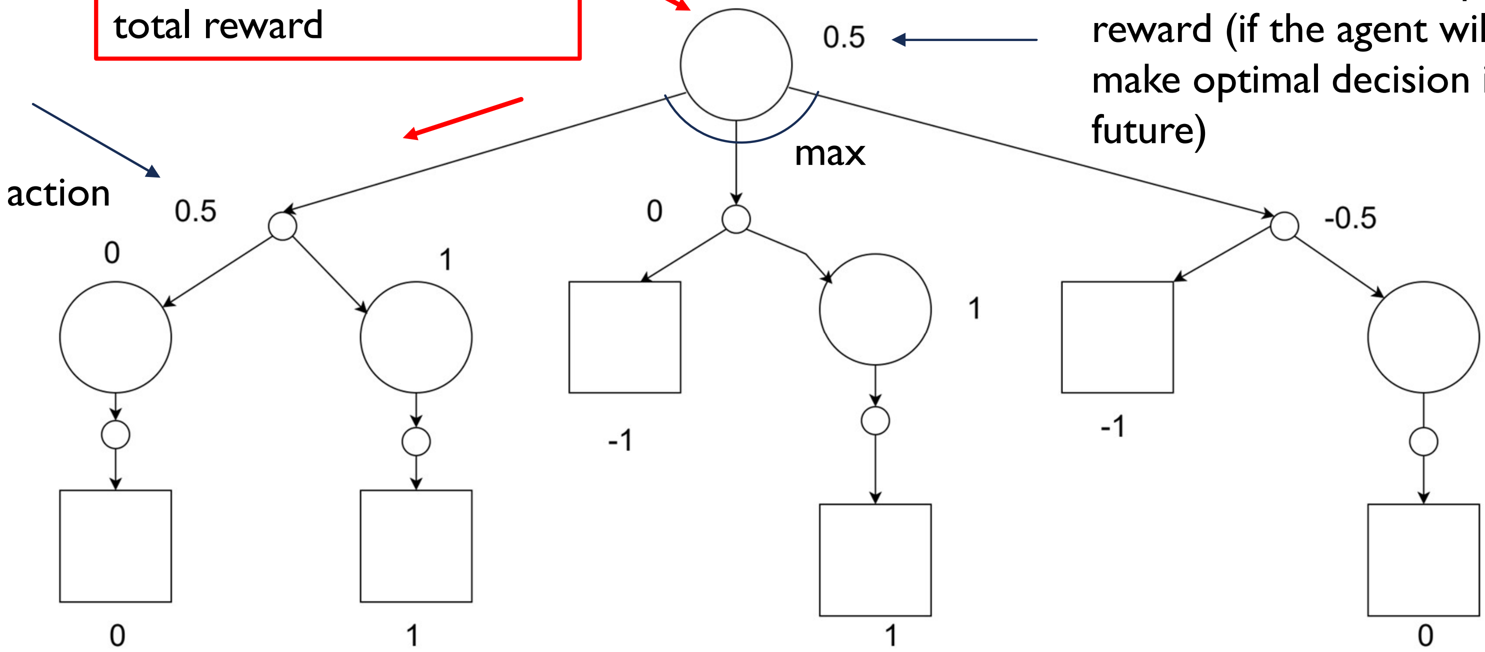
- p(s'|s,a) is the probability that the next state is s', given the current state is s <u>*and*</u> the action taken is a

OPTIMAL POLICY

The best action the agent can make is the one towards the maximum total reward

The value of the state is 0.5 because this the max possible reward (if the agent will always make optimal decision in the future)

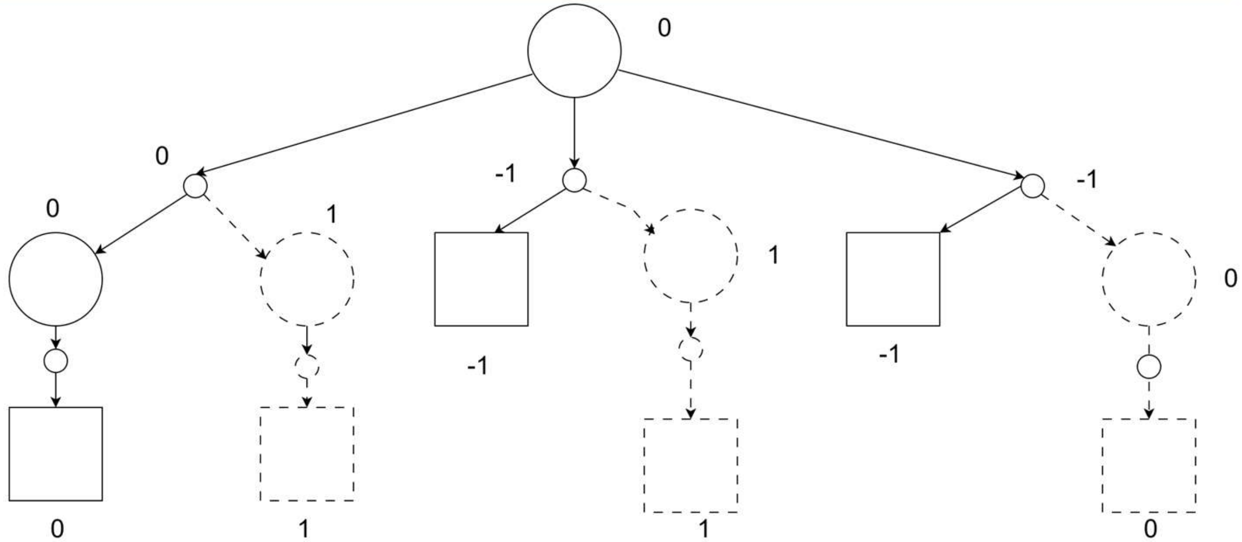
This action can at most provide total average reward 0.5 (in a game, the actual reward will depend on the counter moves)



Given v*(s) the best action to perform is the action that allows the reach states with the highest expected rewards

HOW TO BEAT THE WORST ENEMY? MINIMAX ALGORITHM

- What if the adversary always responds with **its** best possible move?
- The value of the next state should be lowest possible (ideally -1)



Best countermove (worst for the agent)

THE ACTION-VALUE FUNCTION Q(S,A)

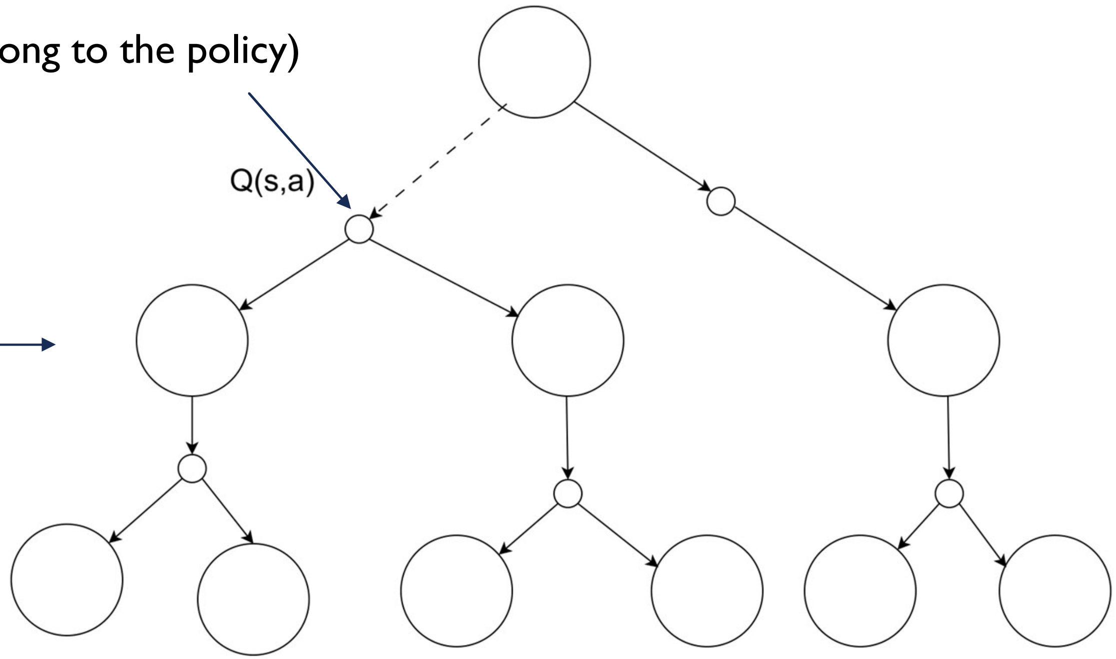
- An alternative way to measure a policy is the state-action value function Q(s,a).
- Q_{π}(s,a) is the expected total return of being in state s, taking action a, and thereafter following the policy π for all subsequent actions.
- The state-action value function is the expected total return of:

1. Being in state s,
2. Taking action a(which does not have to follow the policy π),
3. And then following the policy π for all subsequent actions.

THE ACTION-VALUE FUNCTION Q(S,A)

This action a (dotted lines) is ‘off-policy’ (doesn’t belong to the policy)

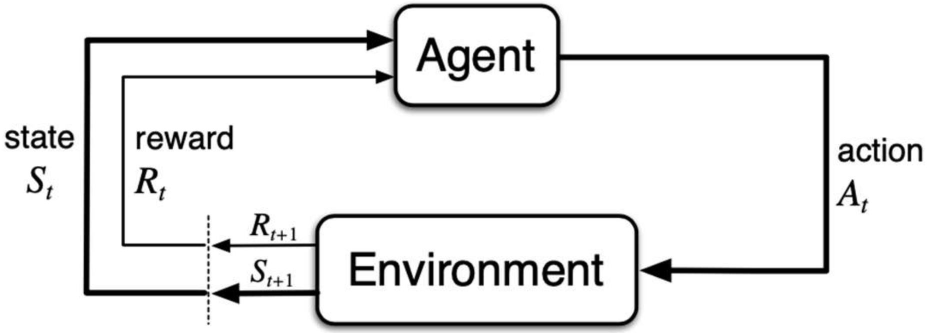
- …from now on, the agent follows the policy (i.e., whatever state is reached, it knows which action to take according to the policy).



Q VALUE COMPUTATION

- Given a policy and a model, Q(s,a) can be computed following similar algorithms to those used to evaluate v(s)
- Similarly, Q*(s,a) can be computed in the same way as v*(s)"

THE AGENT-ENV INTERFACE



- Agent and environment interact at each of a sequence of discrete time steps, t = 0, 1, 2, 3, . . .
- At each time step t, the agent receives some representation of the environment's state, St, and on that basis selects an action, At.
- One time step later, in part as a consequence of its action, the agent receives a numerical reward, Rt+1 and finds itself in a new state, St+1

MARKOV DECISION PROCESS (MDP)

- A MDP=(S,A,P,R, γ), where
  - S: finite set of states
  - A: finite set of actions
  - p: probability transition function p(s,s',a)=Pr{S_{t+1}=s'|A_t=a_t,S_t=s)
  - r: reward function r(s,s',a): reward receceived when moving from s to s' due to action a
  - γ: discount factor γ (0,1)
- In general, there is not a final state, and the system runs 'forever'
- For this reason the sum of rewards is discounted by the factor γ

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$$

$$G_t = R_{t+1} + \gamma G_{t+1}$$

OPTIMAL POLICY

- Optimal policy π^*:
- $V_{\pi*}(s) \geq V_{\pi}(s), \forall s \in S, \forall \pi$
- For finite MDP there is always at least a deterministic optimal policy.

REINFORCEMENT LEARNING

- In reinforcement learning the optimal policy is learned from the experience, from real data
- The model of the environment is unknown .

- Prediction: Estimate a given policy
- Control: Find the optimal policy

MONTECARLO PREDICTION

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$\quad V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$\quad Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

$\quad$ Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$\quad G \leftarrow 0$

$\quad$ Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$\quad\quad G \leftarrow \gamma G + R_{t+1}$

$\quad\quad$ Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

$\quad\quad\quad$ Append G to $Returns(S_t)$

$\quad\quad\quad V(S_t) \leftarrow \text{average}(Returns(S_t))$

- In Monte Carlo (MC)**,** the value of a state V(s) is estimated as the average (over many episodes) of all observed returns following visits to s (a state is at most visited once per episode)
- **Q:** In TTT, what does the value represent?

TEMPORAL DIFFERENCE (TD) PREDICTION

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

- The breakthrough in TD is to improve an estimation using other estimations (bootstrap)
- $V(S) \leftarrow V(S)+\alpha[R+\gamma V(S')-V(S)]$
- $V(S) \leftarrow V(S) + \alpha\ [R+\ \gamma V(S')]- \alpha\ V(S)$
- $V(S) \leftarrow (1- \alpha)V(S) + \alpha\ [R+\ \gamma V(S')]$

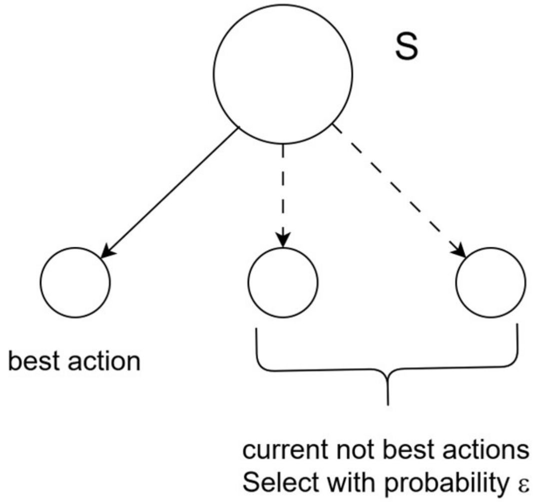
The expected value V(S) based on the (estimated) value of the state S’ reached from S
Called the target value

V(S) is updated as a weighted average of the old estimate and the target.

Q-LEARNING

- Q learning is considered of the most important breakthroughs in RL
- If Q* is known, then best action in state s is $\text{argmax}_a(Q(s,a)^*)$
- Q-learning estimates Q* directly from episodes and provides the optimal policy based on Q*
- The optimal policy is called the **target policy**
- The Q-learning algorithm follows a **behavior policy** which sometimes takes an action different from the target policy
- The behavior policy is used to explore other actions

Behavior policy = ε-greedy policy:
The agent performs an off-policy action with probability ε
Target policy = greedy (take argmax)



Q-LEARNING ALGORITHM

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

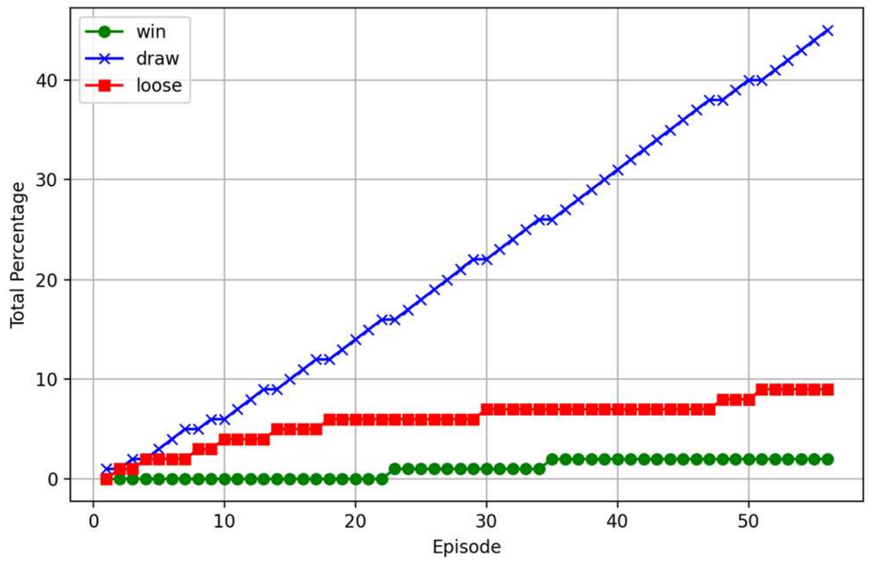
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

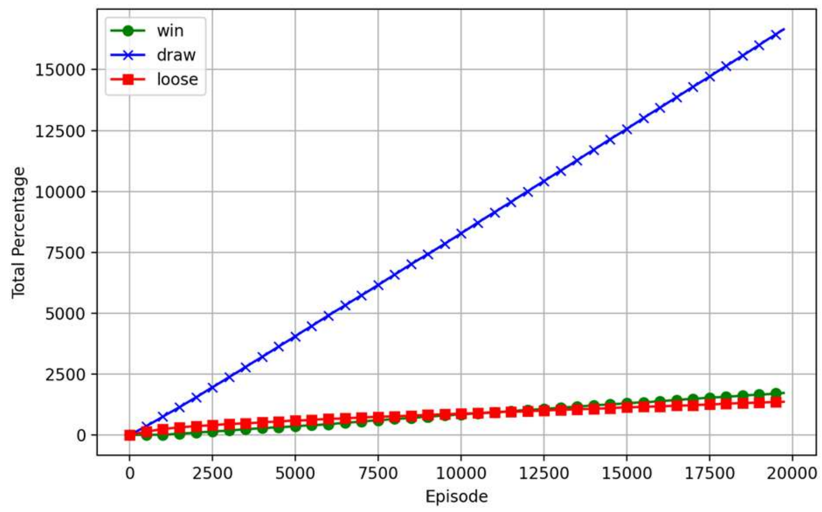
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

- Exploitation: select an action according to Q*
- Exploration: select a random action
- The tradeoff between exploration and exploitation is regulated by ε
- Q-Learning is a tabular method
- For convergence to the optimal policy, every state–action pair must be visited infinitely often, ensuring proper learning of all Q-values.
- The best policy is deterministic, it's just a lookup table

EXAMPLE

Agents makes not optimal choice with some probability



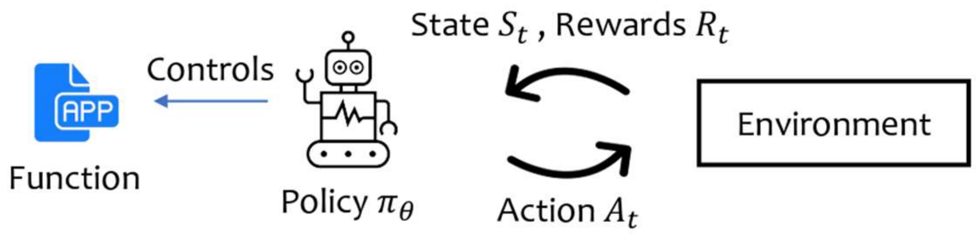


Original code from: https://github.com/rfeinman/tictactoe-reinforcement-learning/tree/master

FROM GAMES TO CLOUD CONTROL

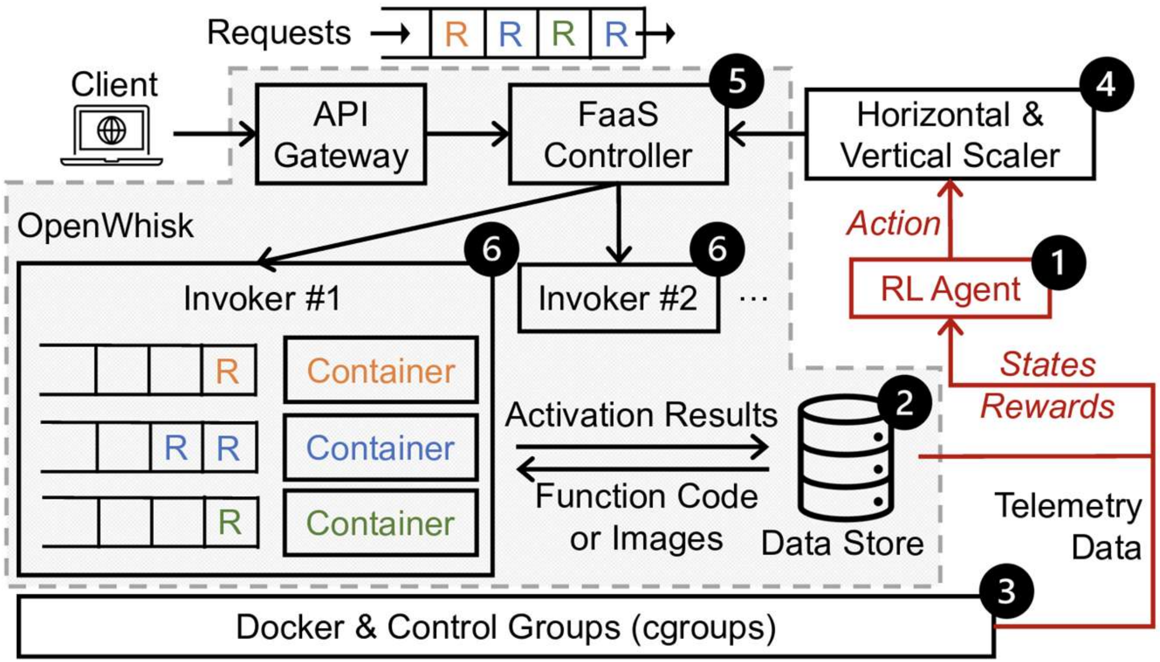
- The same RL principles apply to resource management:
- State: resource usage (CPU, memory), request rate, SLOs
- Actions: adjust resource limits or number of replicas
- Reward: combination of performance, cost, and energy efficiency
- The definition of f the reward function is critical as it should reflect the need to maximize performance and SLO satisfaction, while minimizing energy and cost.

EXAMPLE: RL-BASED AUTO-SCALER



- In most systems, autoscaling is threshold-based (e.g., scale-out if CPU > 70%).
- However, thresholds are static and suboptimal.
- RL-based auto-scaler:
- The state encodes current utilization, latency, and workload trend.
- Actions:
  - Vertical scaling → adjust CPU/memory limits
  - Horizontal scaling → add/remove containers
- Reward: penalizes SLO violations and excessive resource usage.
- The agent learns the thresholds dynamically to balance performance and cost.

SINGLE AGENT RL



the RL agent (labeled as 1) monitors system and application conditions from both the OpenWhisk data store (labeled as 2) and linux cgroup (labeled as 3).

The decision made by the RL agent is then passed by the horizontal and vertical scaler (labeled as 4) to the FaaS controller (labeled as 5) and finally changes the system state and function performance (6).

SINGLE RL DESIGN

- States: SLO Preservation Ratio (SP), Resource Utilization ($RU(CPU,mem)$), Arrival Rate Changes (AC), Resource Limits ($RLT(CPU,mem)$), Horizontal Concurrency (NC)
- Actions:
  - Vertical scaling: +/- step size of resource limits $av = \Delta RLT\ CPU,mem$
  - Horizontal scaling: +/- step size of number of function containers ($ah = \Delta NC$)
- Reward function

$$R_t = \underbrace{\alpha \cdot RU_t}_{\text{Resource Utilization}} + \underbrace{\beta \cdot SP_t}_{\text{SLO Preservation}} + \underbrace{penalty}_{\text{Penalize illegal or undesired actions}}$$

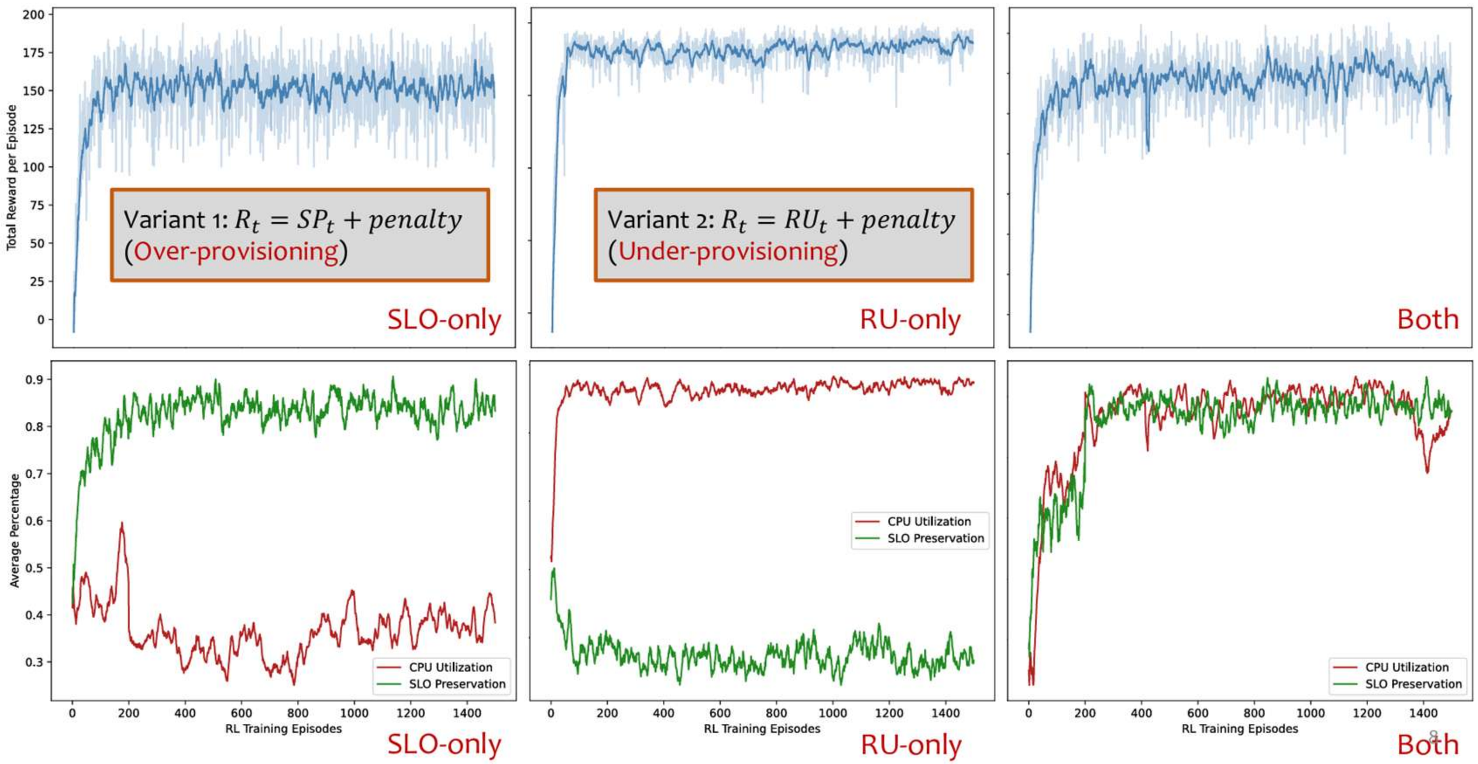
Resource Utilization

SLO Preservation

$$SP_t = \min\left(\frac{SLO\ Latency}{Actual\ Latency}, 1\right)$$

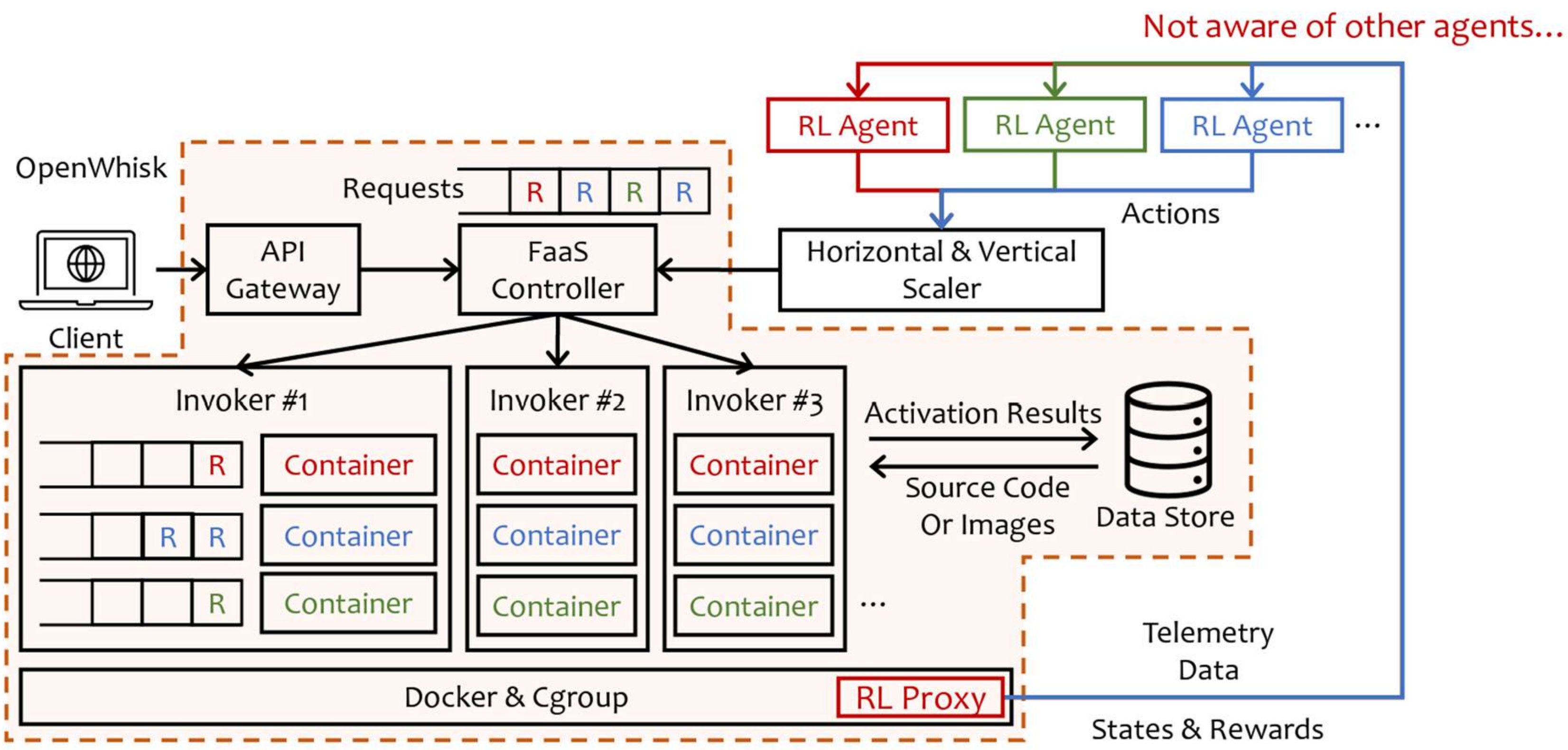
Penalize illegal or undesired actions
- Frequent dangling decisions
- Scale in/up/down when $NC_t = 0$

REWARD FUNCTION SENSITIVITY STUDY (SINGLE-AGENT RL)

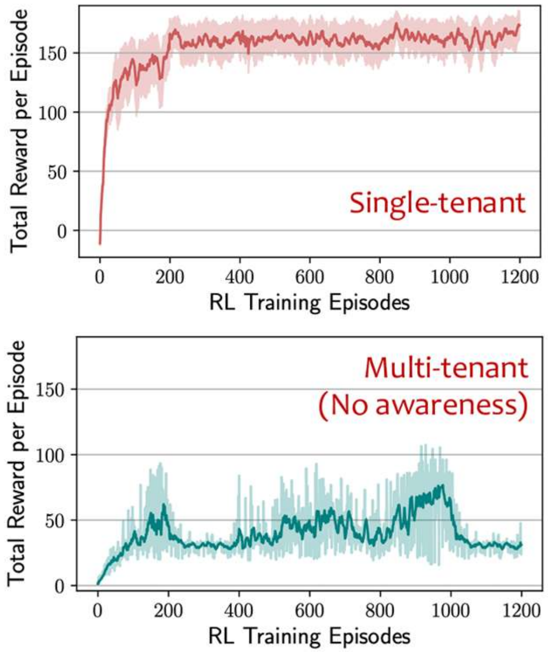


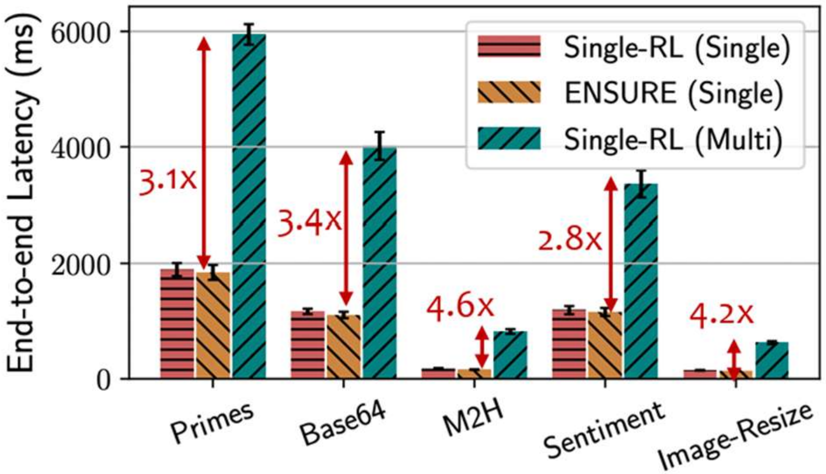
Optimal policy: as few SLO violations as possible while keeping the resource utilization as high as possible

MULTI-TENANT RL PIPELINE IN OPENWHISK



PERFORMANCE UNDER MULTI-TENANCY



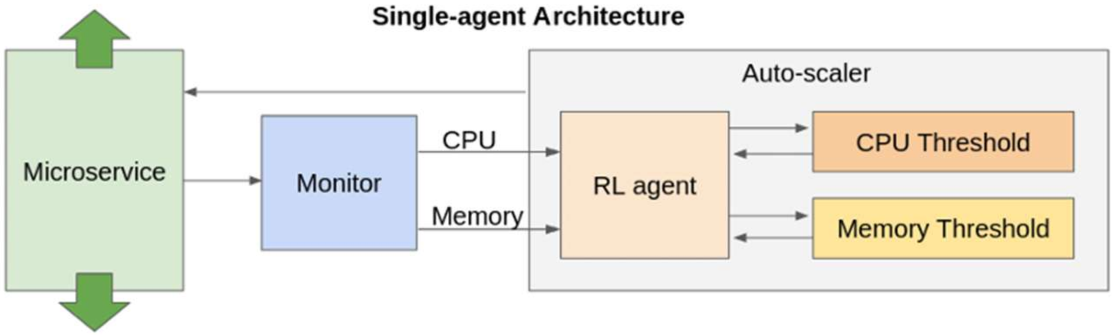


ENSURE = Comparison baseline

RL FOR AUTOSCALING

F. Rossi, V. Cardellini, F. L. Presti and M. Nardelli, "Dynamic Multi-Metric Thresholds for Scaling Applications Using Reinforcement Learning," in IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1807-1821, 1 April-June 2023

- An agent modifies the thresholds the auto-scaler uses to scale-out (increase) or scale in (decrease) the memory and the CPU
- For example, if u> θu (r>θr) the scaler adds CPU by spawning a new replica (horizontal scaling) or to increase the current allocated memory
- The thresholds represent the state of the agent, and the actions increase or decrease the current value of a quanta (δu, δr,), or leave unchanged



state

$$s_i = (\theta_u, u, \theta_r, r)$$

$$u \in \{0, \bar{u}, ..., L_u\bar{u}\}$$

$$r \in \{0, \bar{r}, ..., L_r\bar{r}\}$$

the actual CPU utilization u is discretized into Lu+1 levels

the actual memory allocation r is discretized into Lr+1 levels

Action space

$$\mathcal{A} = \{-\delta_u, -\delta_r, 0, \delta_u, \delta_r\}$$

RL FOR AUTOSCALING

- The reward function is critical as it determines the goal of the agent. By focusing on cost, the reward is interpreted as a penalty, and the agent will try to minimize the penalty

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

$$\begin{aligned} c_{\text{res},u}(s,a,s') &= e^{\xi \frac{\Theta_{\min} - \theta'_u}{\Theta_{\min}}} \\ c_{\text{res},r}(s,a,s') &= e^{\xi \frac{\Theta_{\min} - \theta'_r}{\Theta_{\min}}} \end{aligned}$$

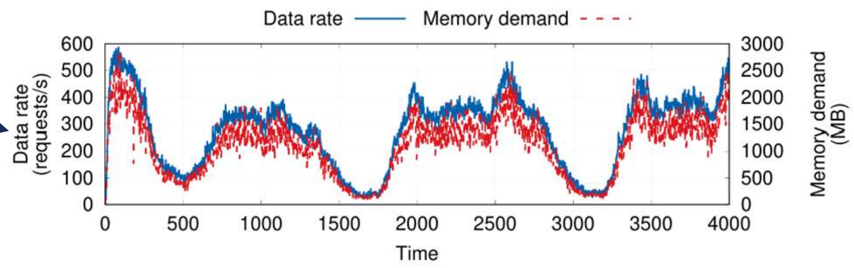
- The cost is higher when the threshold is low because this will likely mean to increase the number of replicas (recall that the state doesn't include the current number of replicas)

$$\begin{aligned} c_{\text{perf},u}(s,a,s') &= \begin{cases} e^{\Gamma \frac{t' - T_{\max}}{T_{\max}}} & t' \le T_{\max} \\ 1 & \text{otherwise} \end{cases} \\ c_{\text{perf},r}(s,a,s') &= \begin{cases} e^{\Gamma \frac{d' - m'}{m'}} & d' \le m' \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

- The cost is higher when the response time is closer to the maximum allowed one (SLO violation occurs when t>Tmax)

AN EXAMPLE OF ESPERIMENTAL RESULTS

- Experiments are performed under realistic workload

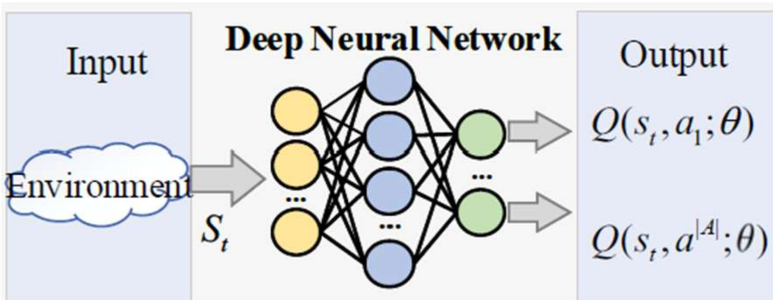


Configuration $\langle w_{\mathrm{perf}}, w_{\mathrm{res}} \rangle$	Average CPU threshold (%)	Average CPU utilization (%)	Average Memory threshold (%)	Average Memory utilization (%)	Median response time (ms)	$T_{\max}$ violations (%)	Memory violations (%)	Average number of replicas
$\langle 1, 0 \rangle$	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

Q-LEARNING AND DQL

- If the state space is not big then the implementation of the Q-learining is almost straightforward, as a matrix can be used to represent Q[s,a]
- However, traditional RL methods struggle with high-dimensional state
- The simplest approach is DQN that extends Q-learning by utilizing a deep neural network to approximate the Q-function Q(s, a; θ) where θ represents the parameters of the network. Another strategy is to directly approximate the best policy. Here the NN provides the probabilities of the best action to perform in a state
- In addition, the number of states can be high, and not all state must be visited (the value of unvisited state can be learned generalizing values of other states)

Value based



Policy based