A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

TESINA FINALE DI

VIRTUAL NETWORKS AND CLOUD COMPUTING

Corso di Laurea Magistrale in Ingegneria Informatica e Robotica
Anno Accademico 2024-2025

Dipartimento di Ingegneria

Docente
Prof. Gianluca Reali

Autonomous Scaling in Edge Computing Using Reinforcement Learning

Un Sistema di Autoscaling Intelligente su Kubernetes basato su Q-Learning

Studente

368204 **Daniele Nanni Cirulli** daniele.nannicirulli@studenti.unipg.it

Sommario

Questo progetto affronta il tema dell'autoscaling dinamico in ambienti di Edge Computing mediante un approccio basato su Reinforcement Learning. È stato progettato e testato un sistema su Kubernetes in cui un agente Q-Learning apprende in maniera autonoma strategie di scaling che bilanciano qualità del servizio e costi operativi. La metodologia sperimentale prevede una netta separazione tra fase di training e fase di evaluation: durante il training l'agente esplora lo spazio delle azioni con una strategia epsilon-greedy, aggiornando una Q-Table composta da 15 stati, ed è esposto a pattern di traffico randomizzati per favorire l'apprendimento di una policy robusta e ridurre il rischio di overfitting. Al termine dell'addestramento la Q-Table viene salvata e riutilizzata in evaluation in modalità greedy, senza esplorazione. La reward function adotta un approccio zone-based che adatta dinamicamente i pesi in base allo stato del sistema: in condizioni di violazione SLA il costo delle repliche viene attenuato per non scoraggiare lo scale-up necessario, mentre in condizioni di calma il peso del costo aumenta per incentivare il risparmio di risorse. Un ulteriore termine di action shaping introduce un bias euristico che accelera la convergenza dell'apprendimento. I risultati sperimentali mostrano che l'agente RL raggiunge il 95,1% di SLA Met, contro l'86,6% di un autoscaler baseline rule-based: un aumento di 8,5 punti percentuali di episodi che rispettano il vincolo di latenza. Il prezzo di questo miglioramento è un incremento del 9,4% nel numero medio di repliche attive, che tuttavia non compromette l'efficienza complessiva del sistema. L'architettura è completata da una dashboard interattiva sviluppata in Streamlit che permette di monitorare le prestazioni in tempo reale, modificare dinamicamente le soglie SLA e confrontare visivamente le performance dei due approcci.

Indice

1 Introduzione

1.1 Contesto e Motivazioni

L'Edge Computing rappresenta un paradigma emergente in cui l'elaborazione dei dati viene spostata dai datacenter centralizzati verso dispositivi periferici più vicini agli utenti finali [1]. Questa scelta architetturale riduce la latenza di comunicazione e migliora la reattività delle applicazioni, risultando particolarmente adatta per scenari come Internet of Things, streaming video ad alta definizione e sistemi di controllo real-time [2].
Tuttavia, l'Edge Computing introduce sfide specifiche rispetto al Cloud tradizionale. I nodi Edge dispongono di risorse computazionali limitate in termini di CPU, memoria ed energia, e devono gestire carichi di lavoro altamente variabili e difficilmente prevedibili. In questo contesto, la capacità di adattare dinamicamente le risorse allocate diventa cruciale per mantenere un equilibrio tra qualità del servizio ed efficienza economica.
Gli approcci tradizionali di autoscaling, come il Horizontal Pod Autoscaler di Kubernetes, si basano su regole predefinite che confrontano metriche con soglie fisse [3]. Quando una metrica supera un limite superiore, il sistema aumenta le repliche; quando scende sotto un limite inferiore, le riduce. Questa logica reattiva presenta però limitazioni significative. Non tiene conto della storia recente del sistema, reagisce solo agli eventi correnti e potrebbe andare incontro a oscillazioni continue note come flapping, in cui il sistema scala ripetutamente up e down degradando le performance complessive.
Il Reinforcement Learning offre un'alternativa interessante a questo approccio. Un agente RL può apprendere una strategia di scaling ottimale osservando come le proprie azioni influenzano lo stato del sistema nel tempo. Invece di seguire regole rigide, l'agente costruisce una rappresentazione del valore atteso di ogni azione in ogni stato, permettendogli di anticipare le conseguenze delle proprie decisioni e di adattarsi a pattern di traffico complessi senza richiedere regole esplicite.

1.2 Obiettivi del Progetto

Gli obiettivi specifici di questo lavoro sono i seguenti:

- Progettare e implementare un agente RL basato su Q-Learning capace di apprendere autonomamente policy di scaling efficaci in un ambiente Kubernetes reale;
- Sviluppare un protocollo sperimentale rigoroso che separi nettamente la fase di training dalla fase di valutazione, permettendo un confronto equo con approcci rule-based;
- Implementare un'infrastruttura completa che includa generazione di carico multi-scenario, logging strutturato e monitoraggio interattivo;

- Validare quantitativamente l'efficacia dell'approccio RL misurando latenza, violazioni SLA[1], costi e stabilità rispetto a una baseline tradizionale.

2 Fondamenti Teorici

2.1 Reinforcement Learning e Q-Learning

Il Reinforcement Learning è un framework di apprendimento automatico in cui un agente apprende a prendere decisioni ottimali interagendo con un ambiente. A differenza dell'apprendimento supervisionato, dove vengono forniti esempi etichettati di comportamento corretto, nel RL l'agente riceve solo feedback sotto forma di ricompense numeriche che indicano quanto sia stata buona o cattiva un'azione in uno specifico contesto.
Il problema è formalizzato come un Markov Decision Process, definito dalla tupla $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ dove $\mathcal{S}$ è l'insieme degli stati possibili, $\mathcal{A}$ l'insieme delle azioni disponibili, $P(s'|s,a)$ la probabilità di transizione da uno stato s a uno stato s' eseguendo l'azione a, $R(s,a)$ la funzione di ricompensa e $\gamma \in [0,1]$ il fattore di sconto che determina l'importanza delle ricompense future [4].
L'obiettivo dell'agente è apprendere una policy $\pi : \mathcal{S} \to \mathcal{A}$ che massimizza il valore atteso delle ricompense cumulative scontate nel tempo. Il valore di uno stato sotto una policy π è definito come:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right]$$

Q-Learning è un algoritmo off-policy che apprende direttamente la funzione action-value $Q(s,a)$, che rappresenta il valore atteso delle ricompense future partendo dallo stato s, eseguendo l'azione a e seguendo poi la policy ottimale. La regola di aggiornamento di Q-Learning è:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{1}$$

dove α è il learning rate che controlla la velocità di apprendimento, r è la ricompensa osservata, s' è il nuovo stato raggiunto dopo aver eseguito a in s, e il termine $\gamma \max_{a'} Q(s',a')$ stima il valore futuro ottimale [5].
Per bilanciare esplorazione e sfruttamento, Q-Learning utilizza tipicamente una strategia epsilon-greedy: con probabilità ϵ l'agente sceglie un'azione casuale per esplorare lo spazio delle azioni, mentre con probabilità $1-\epsilon$ sceglie l'azione con valore Q massimo. Nel nostro caso, utilizziamo un epsilon decay esponenziale che riduce gradualmente l'esplorazione man mano che l'agente acquisisce esperienza.

[1] **SLA (Service Level Agreement)**: Accordo sul livello di servizio che definisce gli standard qualitativi minimi che il fornitore si impegna a garantire al cliente. Nel contesto di questo lavoro, lo SLA si riferisce specificamente al vincolo sulla latenza massima di risposta (fissato a 0,35s).

2.2 Kubernetes e Autoscaling

Kubernetes è una piattaforma open-source per l'orchestrazione di container che automatizza il deployment, lo scaling e la gestione di applicazioni containerizzate. I concetti fondamentali rilevanti per questo progetto sono i Pod, che rappresentano l'unità minima di deployment contenente uno o più container, i Deployment, che gestiscono la creazione e l'aggiornamento di insiemi di Pod identici, e i Service, che espongono i Pod attraverso endpoint di rete stabili.
Kubernetes offre meccanismi nativi di autoscaling a diversi livelli. L'Horizontal Pod Autoscaler scala il numero di repliche basandosi su metriche come utilizzo CPU o memoria. Il Vertical Pod Autoscaler modifica le risorse allocate ai singoli Pod. Il Cluster Autoscaler aggiunge o rimuove nodi al cluster. Il nostro sistema implementa un autoscaler orizzontale custom che sostituisce la logica rule-based dell'HPA con un agente RL [3].

2.3 Modellazione del Problema come MDP

Nel contesto dell'autoscaling, il problema viene modellato come un *Markov Decision Process* (MDP). Lo stato del sistema è descritto dalla coppia $s = (l, n)$, dove $l \in \{0, 1, 2\}$ rappresenta il *bucket* di latenza e $n \in \{1, 2, 3, 4, 5\}$ indica il numero corrente di repliche attive.
Il bucket di latenza viene determinato confrontando la latenza misurata con due soglie configurabili l_{low} e l_{high}. In particolare:

- se la latenza è inferiore a l_{low}, il sistema si trova nella zona ottimale ($l = 0$);
- se la latenza è compresa tra l_{low} e l_{high}, il sistema rientra nella zona target ($l = 1$);
- se la latenza supera l_{high}, si ha una violazione degli SLA ($l = 2$).

Con questa discretizzazione, lo spazio degli stati risulta composto da $3 \times 5 = 15$ stati discreti.
Lo spazio delle azioni è definito come

$$A = \{-1, 0, +1\},$$

dove -1 corrisponde a uno *scale down* (rimozione di una replica), 0 indica *hold* (configurazione invariata) e $+1$ rappresenta uno *scale up* (aggiunta di una replica). Le azioni sono vincolate dai limiti del sistema: il numero di repliche deve rimanere nell'intervallo $[n_{\min}, n_{\max}]$, con $n_{\min} = 1$ e $n_{\max} = 5$.
Infine, la **reward function** è stata progettata per bilanciare tre obiettivi: (i) mantenere la latenza bassa, rispettando gli SLA; (ii) minimizzare il numero di repliche per contenere i costi; (iii) favorire stabilità, scoraggiando variazioni troppo frequenti della configurazione.

2.4 Metriche di Valutazione

Per valutare in modo quantitativo le prestazioni degli autoscaler ho considerato diverse metriche. La **latenza media** misura il tempo di risposta medio del servizio, espresso in secondi. Il **numero medio di repliche** fornisce invece una stima del costo operativo, in quanto riflette le risorse mediamente allocate.
Infine, la metrica **SLA Met**, introdotta specificamente per questo progetto, misura la percentuale di episodi in cui la latenza rimane al di sotto della soglia di violazione critica l_{high}:

$$\text{SLA Met} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}\big(\bar{l}_i < l_{\text{high}}\big) \cdot 100$$

dove N è il numero totale di episodi, $\bar{l}_i$ è la latenza media osservata nell'episodio i, l_{high} rappresenta il limite superiore dello SLA (fissato a 0.35s nel nostro setup) e $\mathbb{I}(\cdot)$ è la funzione indicatrice, pari a 1 se la condizione è verificata e 0 altrimenti.
Definiamo inoltre l'SLA Efficiency come il rapporto tra qualità del servizio e costi:

$$\text{Eff} = \frac{\text{SLA Met}}{\text{Repliche Medie}}$$

Questo indice sintetizza il trade-off fondamentale dell'autoscaling: un sistema efficiente mantiene alta qualità con poche risorse.

3 Architettura del Sistema

3.1 Panoramica dell'Architettura

Il sistema implementato adotta un'architettura cloud-native basata su Kubernetes e segue il ciclo MAPE (Monitor-Analyze-Plan-Execute). L'infrastruttura è divisa in due ambiti principali: la macchina host (su WSL2), che ospita il sistema di controllo, e il cluster Kubernetes (gestito tramite Minikube), dove gira l'applicazione edge containerizzata.
Sulla macchina host troviamo quattro componenti fondamentali: la dashboard di monitoraggio e configurazione in Streamlit, l'autoscaler RL basato su Q-Learning, l'autoscaler baseline rule-based e il generatore di carico HTTP multithread.
Lato cluster, invece, sono presenti il server API di Kubernetes (che funge da control plane), un servizio NodePort per il bilanciamento del carico e il deployment dei Pod Flask, configurati per simulare il comportamento di un nodo edge.
Il controllo avviene a ciclo chiuso. Un load generator sulla macchina host invia richieste HTTP concorrenti al servizio NodePort, che le smista ai Pod. Ogni Pod Flask, oltre a elaborare la richiesta, introduce un ritardo di circa 200 ms per emulare i tempi di servizio

reali. A intervalli regolari, l'autoscaler attivo (RL o baseline, mai entrambi contemporaneamente) esegue il monitoraggio: misura la latenza media su un campione di richieste e legge lo stato del deployment (numero di repliche) tramite le API di Kubernetes.
Nella fase di analisi, la latenza osservata viene discretizzata in tre classi (bassa, target, alta); questa, insieme al numero di repliche, definisce lo stato del sistema. L'autoscaler baseline applica una politica di scaling deterministica basata su soglie fisse, mentre l'autoscaler RL sceglie l'azione (scale up, hold, scale down) consultando la Q-Table addestrata in precedenza su vari scenari di traffico. L'azione scelta viene tradotta in un comando di scaling (equivalente a un kubectl scale), che porta alla creazione o terminazione dei Pod. Parallelamente, tutte le metriche (timestamp, episodio, latenza, repliche, reward ed epsilon per il caso RL) vengono salvate su file CSV. Questi log sono utilizzati sia dalla dashboard Streamlit per mostrare grafici e KPI (come violazioni SLA o efficienza), sia per le analisi discusse nei capitoli successivi.

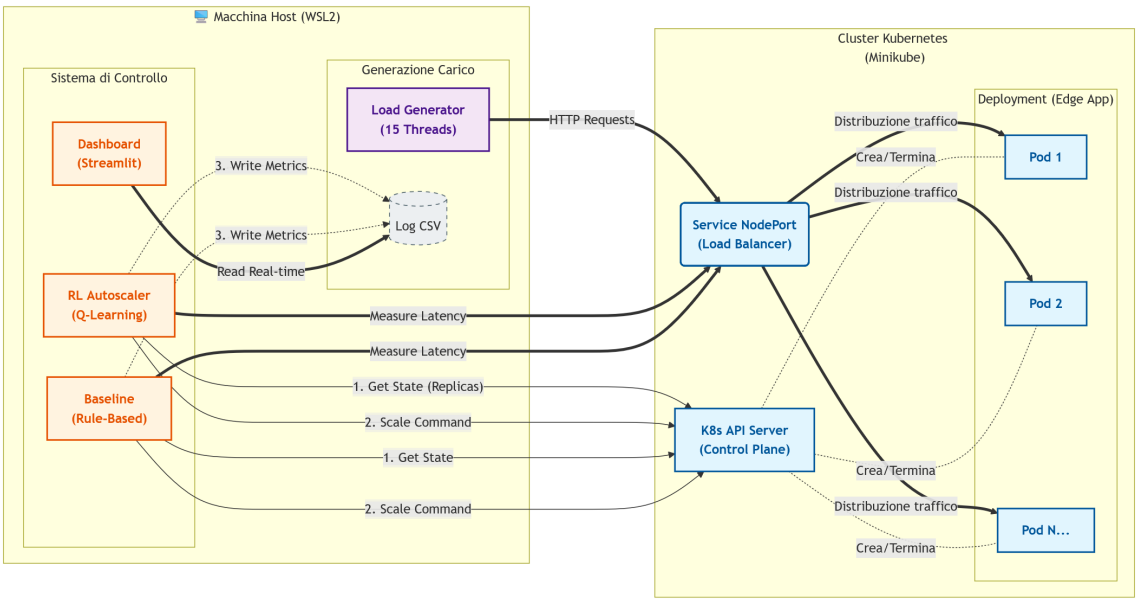

Figura 1: Architettura del sistema. Il loop di controllo MAPE vede gli Autoscaler monitorare la latenza tramite probing attivo sul Service NodePort e attuare le decisioni di scaling tramite l'API Server Kubernetes. I dati vengono persistiti su CSV per analisi e visualizzazione real-time tramite Dashboard.

3.2 Ambiente Kubernetes su Minikube

L'ambiente sperimentale è basato su Minikube [6], configurato per emulare un nodo Edge attraverso l'imposizione di stretti vincoli sulle risorse (Resource Quotas). Sebbene l'host fisico disponga di capacità computazionale superiore, ogni singola replica dell'applicazione (Pod) è stata vincolata a livello di container runtime mediante le direttive requests e limits di Kubernetes.
Nello specifico, a ciascun Pod è garantita una quota minima di 100m CPU (0,1 core) e 128MiB di RAM, con un limite superiore fissato a 1 CPU e 512MiB di RAM. Questa configurazione costringe il sistema a operare con risorse limitate, riproducendo le limitazioni hardware tipiche dei dispositivi IoT/Edge e rendendo necessario l'intervento dinamico dell'autoscaler al crescere del carico.

3.3 Microservizio Edge Application

L'applicazione target è un semplice server Flask [7] che risponde su una singola route, restituendo "OK" dopo un delay fisso di 200 millisecondi. Questo delay simula un carico computazionale costante e prevedibile che consente di isolare l'effetto del load balancing dalle fluttuazioni intrinseche del tempo di elaborazione. La scelta di 200ms rappresenta un compromesso: è sufficientemente lungo da rendere visibili gli effetti dello scaling, ma abbastanza breve da permettere test rapidi.

3.4 Load Generator Multi-Thread

Il generatore di traffico è un componente critico per validare l'autoscaler sotto condizioni controllate ma realistiche. Il sistema utilizza 15 thread worker concorrenti che inviano richieste HTTP in parallelo. La frequenza delle richieste è controllata da un thread centrale che legge ciclicamente un file di configurazione condiviso specificando lo scenario attivo.
Sono supportati quattro scenari principali:

- scenario **calma**: ogni thread invia circa 2 richieste al secondo, generando un carico aggregato di 30 req/s facilmente gestibile da un singolo Pod;
- scenario **spike**: la frequenza sale a 20 req/s per thread, saturando i Pod disponibili e forzando violazioni SLA se le repliche sono insufficienti;
- scenario **onda**: la frequenza varia sinusoidalmente tra 2 e 20 req/s, simulando pattern di traffico fluttuanti tipici di applicazioni reali;
- scenario **stop**: interrompe completamente il traffico.

La configurazione a 15 worker non è arbitraria, ma calibrata per stressare il sistema oltre i limiti di un singolo Pod. Mentre un carico sequenziale verrebbe smaltito mantenendo la latenza base di 200ms, la concorrenza di 15 thread genera un collo di bottiglia intenzionale. Nello scenario spike, questo si traduce in un carico di 300 req/s che satura la capacità computazionale disponibile, innalzando la latenza fino a 1.0s in assenza di scaling orizzontale.

4 Implementazione degli Algoritmi

4.1 Separazione Training ed Evaluation

Una scelta metodologica fondamentale di questo progetto è stata la netta separazione tra fase di addestramento e fase di valutazione dell'agente RL. Questa distinzione elimina un bias comune nei confronti con approcci rule-based: includere episodi di esplorazione nel confronto penalizzerebbe ingiustamente l'RL poiché durante l'esplorazione l'agente compie intenzionalmente azioni subottimali per scoprire nuove strategie.
L'agente opera in due modalità distinte controllate da una variabile d'ambiente. In modalità train, l'agente inizializza una Q-Table vuota di dimensione 15×3, esegue Q-Learning standard con aggiornamento ad ogni episodio secondo l'Equazione 1, utilizza una strategia epsilon-greedy con decay esponenziale partendo da $\epsilon_0 = 0,9$ e decadendo fino a $\epsilon_{\min} = 0,1$, salva periodicamente la Q-Table su disco e logga metriche dettagliate includendo il valore corrente di epsilon.
In modalità eval, l'agente carica la Q-Table pre-addestrata da disco, imposta $\epsilon = 0$ eliminando completamente l'esplorazione casuale, disabilita l'aggiornamento della Q-Table congelando la policy appresa, esegue azioni puramente greedy scegliendo sempre $a = \arg\max_{a'} Q(s, a')$ e logga risultati su file separato per distinguerli dai dati di training.
Il confronto tra RL e baseline viene effettuato esclusivamente sui dati di evaluation, dove entrambe le policy sono stazionarie e deterministiche. La fase di training serve unicamente ad apprendere la Q-Table ottimale, che viene poi validata separatamente.

4.2 Fase di Training con Pattern Randomizzati

Per evitare che l'agente memorizzi una sequenza specifica di eventi invece di apprendere una policy generalizzabile, l'agente non viene esposto a un singolo scenario fisso ma a una combinazione randomizzata di pattern diversi.
Il training è orchestrato da uno script dedicato (`training_benchmark.py`) che scrive ciclicamente nel file condiviso letto dal load generator. Sono definiti tre blocchi base corrispondenti agli scenari calma, onda e spike, ciascuno con durata variabile estratta casualmente in un intervallo specificato. Per ogni super-ciclo di training, i blocchi vengono

mescolati in ordine casuale, viene estratta una durata casuale per ciascuno nell'intervallo definito e lo scenario viene scritto nel file di configurazione. Il sistema attende la fine del blocco prima di passare al successivo.
Durante la fase di addestramento sono stati completati 6 super-cicli per un totale di circa 200 episodi. Questa durata si è rivelata sufficiente per ottenere la convergenza dell'algoritmo, assicurando la copertura di tutti i 15 stati, il decay di ϵ fino al target di 0,1 e un periodo finale di oltre 50 episodi per il consolidamento della policy appresa.
Per la fase di evaluation, viene utilizzato `benchmark.py` che esegue una sequenza deterministica e fissa, garantendo che il confronto tra RL e Baseline avvenga su condizioni assolutamente identiche. Entrambi gli autoscaler vengono testati sullo stesso pattern di scenari con durate e caratteristiche identiche, eliminando qualsiasi variabile legata al traffico. Questa scelta metodologica è stata molto importante per ottenere un confronto equo tra i due sistemi.

4.3 Reward Function Zone-Based

La progettazione della reward function ha rappresentato uno degli aspetti più critici e complessi, dato il ruolo fondamentale che ricopre all'interno di un sistema di Reinforcement Learning: essa codifica in modo implicito gli obiettivi del controllo. La struttura adottata utilizza un approccio zone-based che adatta dinamicamente i pesi delle diverse componenti in base allo stato corrente del sistema.
La reward complessiva è composta da tre termini additivi:

- Lo **SLA term** che premia latenze basse e penalizza violazioni. Nella zona high, quando la latenza supera la soglia critica l_{high}, viene assegnata una penalità severa di -10. Nella zona target, tra l_{low} e l_{high}, il reward non è fisso ma varia linearmente in base a quanto la latenza è vicina al limite inferiore. Viene calcolata una frazione $f = (l_{\text{high}} - l)/(l_{\text{high}} - l_{\text{low}})$ che vale 1 quando si è appena sopra la soglia bassa e 0 quando si è al limite della violazione. Il reward SLA è allora $2 + 4f$, variando quindi tra 2 e 6. Questo incentiva l'agente a mantenersi nella parte bassa della zona target invece di operare al limite della violazione. Infine, nella zona low, quando la latenza è sotto l_{low}, viene assegnato un premio di $+7$;

- Il **cost term** che penalizza le repliche extra. Il numero di repliche in eccesso rispetto al minimo è $e = \max(0, n-1)$ e il costo è $-w \cdot e$ dove il peso w varia con la zona. Nella zona high, quando il sistema sta violando lo SLA, il peso è basso ($w = 0,5$) per non scoraggiare lo scale-up necessario a rientrare nel target. Nella zona target, il peso è standard ($w = 1,0$). Nella zona low, quando la latenza è ottimale, il peso è alto (w = 2,0) per incentivare fortemente la riduzione di repliche non necessarie. Questo meccanismo ha lo scopo di evitare comportamenti eccessivamente conserva-

tivi da parte dell'agente RL, che tenderebbero a mantenere risorse superflue anche in condizioni di carico ridotto.

- Lo **shaping term** che fornisce un piccolo bias euristico per accelerare l'apprendimento. Se l'agente si trova in zona high e sceglie di scalare up, riceve un bonus di +2 per incoraggiare la reazione corretta. Se si trova in zona low con più di una replica attiva e sceglie di scalare down, riceve un bonus di +1 per incentivare il risparmio. In zona target non viene assegnato alcun bias, lasciando che l'agente decida basandosi sul bilanciamento naturale tra SLA e costi.

La reward totale è quindi $R(s,a) = R_{\text{SLA}}(l) + R_{\text{cost}}(n,l) + R_{\text{shape}}(a,l,n)$. Questa formulazione permette all'agente di apprendere comportamenti adattivi: spendere risorse aggressivamente quando necessario per mantenere la qualità del servizio, ma risparmiare quando le condizioni lo permettono. Il peso del costo varia di un fattore quattro tra zona low e zona high, creando una forte differenziazione contestuale.

4.4 Analisi Reward in Fase di Training

Per valutare il processo di apprendimento dell'agente è stata analizzata l'evoluzione della funzione di ricompensa durante la fase di training. La Figura 2 riporta l'andamento del reward medio in funzione degli episodi e alla curva di esplorazione ϵ della policy epsilon-greedy.

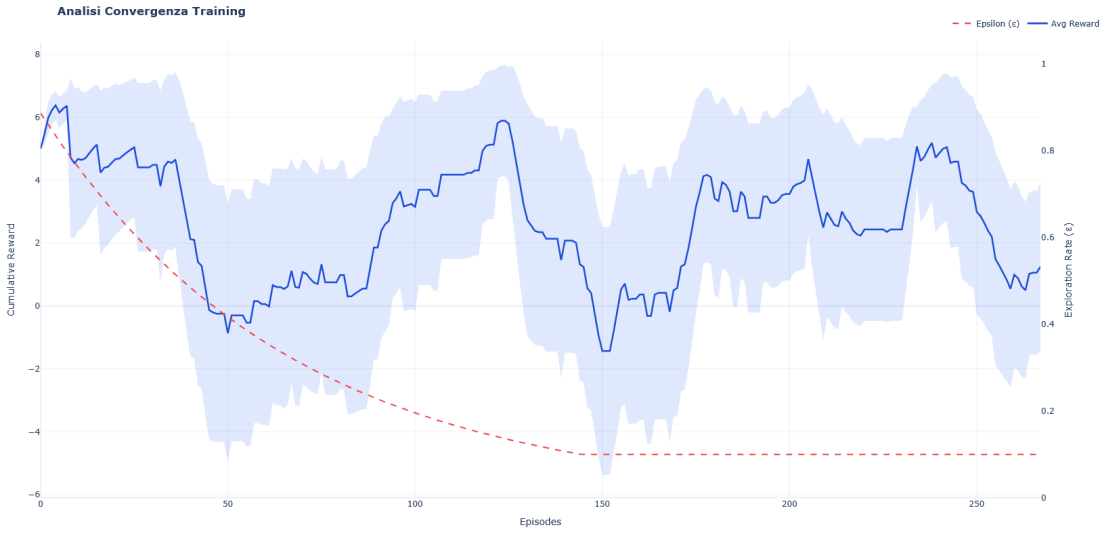

Figura 2: Andamento del reward medio e del tasso di esplorazione ϵ durante l'addestramento.

A differenza di un problema statico e deterministico, in cui ci si aspetterebbe una convergenza più stabile, il task di autoscaling considerato è intrinsecamente non stazionario: gli scenari di traffico (calma, onda e spike) si alternano nel tempo, modificando la difficoltà del controllo e il reward massimo ottenibile in ciascun episodio. Di conseguenza, fasi

di carico leggero producono reward più elevati e stabili, mentre durante gli spike anche politiche quasi ottimali subiscono penalità legate all'aumento della latenza.
Le oscillazioni osservate non indicano quindi una mancata convergenza, ma riflettono sia la dinamicità dell'ambiente sia la presenza di esplorazione residua dovuta a valori di ϵ non nulli. In questo contesto, la convergenza non va interpretata come una stabilizzazione puntuale del reward, bensì come la capacità dell'agente di mantenere una media positiva e relativamente stabile nel tempo.

4.5 Baseline Rule-Based Autoscaler

L'autoscaler baseline implementa una strategia tradizionale a soglie fisse senza alcuna memoria dello stato passato. La logica decisionale è puramente reattiva: se la latenza misurata supera la soglia alta e il numero di repliche è sotto il massimo consentito, incrementa le repliche di uno, mentre, se la latenza scende sotto la soglia bassa e le repliche sono sopra il minimo, decrementa di uno. In caso contrario, mantiene la configurazione corrente.
Per garantire un confronto equo, anche la baseline calcola un reward post-hoc utilizzando esattamente la stessa reward function dell'agente RL. Questo permette di confrontare i due approcci sulla stessa metrica unificata. Tuttavia, è importante sottolineare che la baseline non utilizza questo reward per decidere le proprie azioni, che rimangono determinate esclusivamente dalle regole IF-THEN. Il reward serve unicamente per valutazione comparativa.

5 Dashboard di Controllo

Per supportare le fasi di sperimentazione e garantire un monitoraggio efficace del sistema in tempo reale, è stata sviluppata una dashboard interattiva basata sul framework Streamlit [8]. L'architettura dell'interfaccia è stata progettata per offrire tre modalità di utilizzo distinte: la visualizzazione dedicata all'agente RL, quella per la Baseline e una specifica modalità di Confronto Diretto per l'analisi comparativa.
Il pannello laterale (sidebar) funge da centro di controllo operativo, da cui è possibile modificare dinamicamente le soglie SLA tramite slider. Quando questo accade le modifiche vengono salvate istantaneamente su un file di configurazione JSON condiviso letto periodicamente dagli autoscaler. Sempre dalla sidebar, l'utente può pilotare il generatore di traffico, attivando i diversi scenari di carico (Calma, Onda, Spike) attraverso un pulsante.
Nelle viste di monitoraggio singolo, l'area centrale presenta una serie di card riassuntive che mostrano la latenza media nell'i-esimo episodio, il numero di repliche correnti e il reward dell'ultimo episodio. I grafici temporali tracciano l'evoluzione, sovrapponendo le

curve di latenza alle soglie SLA per rendere immediatamente evidenti eventuali violazioni. Particolarmente utile in fase di training è il grafico del Reward Cumulativo, strumento essenziale per diagnosticare visivamente la velocità e la stabilità della convergenza dell'algoritmo di apprendimento.

La modalità Confronto Diretto, invece, è pensata per l'analisi post-evaluation. La dashboard carica e aggrega automaticamente i log di evaluation di entrambi gli approcci, calcolando metriche comparative come la latenza media, la percentuale di SLA Met, il costo computazionale e l'efficienza complessiva. Un elemento distintivo di questa vista è l'integrazione delle Contextual Zones: i grafici delle serie temporali vengono arricchiti con zone colorate che identificano semanticamente lo scenario di traffico attivo in ogni istante. Questa correlazione visiva è fondamentale per l'interpretabilità, in quanto permette di distinguere se un improvviso aumento delle repliche sia una reazione fisiologica a un picco di carico o un sintomo di instabilità del controllore. L'analisi è infine completata da Box Plot che confrontano le distribuzioni statistiche di latenza e repliche, evidenziando le differenze tra i due approcci in termini di varianza e robustezza.

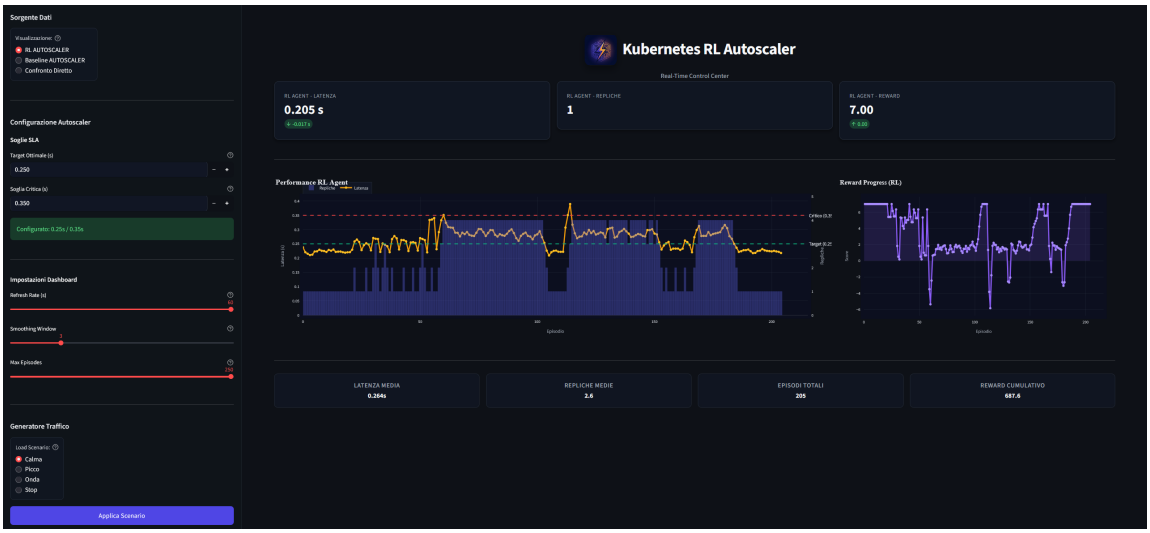

Figura 3: Dashboard di controllo. L'interfaccia permette di monitorare le metriche degli autoscaler, modificare soglie SLA dinamicamente e confrontare visivamente le performance attraverso grafici interattivi e statistiche aggregate.

6 Risultati Sperimentali

6.1 Setup Sperimentale

Gli esperimenti sono stati condotti su un laptop con AMD Ryzen 7 8845HS e 32 GB di RAM, eseguendo Windows 11 con WSL 2 Ubuntu 24.04. Il cluster Kubernetes è stato gestito tramite Minikube versione 1.37.0 con driver Docker [9]. Python 3.12.3 è stato utilizzato per tutti gli script con NumPy 2.3.4 per operazioni numeriche e Pandas 2.3.3

per manipolazione dati.

Prima Fase - Training RL In questa prima fase è stato effettuato il training dell'agente RL attraverso ***training_ benchmark.py***, esponendo l'agente a un curriculum di 6 super-cicli randomizzati comprendenti scenari calma, onda e spike con ordine e durata variabili. I parametri di apprendimento erano:

- learning rate $\alpha = 0,1$;
- discount factor $\gamma = 0,9$;
- epsilon iniziale $\epsilon_0 = 0,9$ decadente fino a $\epsilon_{\min} = 0,1$ con fattore decay 0,985;

La Q-Table appresa è stata salvata su disco al termine del training.

Seconda Fase - Evaluation RL Nella fase due è stato effettuato l'evaluation dell'agente RL. Il cluster è stato resettato a una replica e l'agente è stato riavviato in modalità eval caricando la Q-Table pre-addestrata. Con epsilon forzato a zero, la policy era completamente deterministica eseguendo solo azioni greedy. Il test è stato condotto tramite benchmark.py che simulava scenari differenti.

Terza Fase - Evaluation Baseline e Confronto Nella fase tre è stato effettuato l'evaluation della baseline. Il cluster è stato nuovamente resettato alla stessa configurazione iniziale e l'autoscaler baseline è stato testato con lo stesso ***benchmark.py*** dell'agente RL.

6.2 Risultati Quantitativi

La Tabella 1 riassume le performance aggregate dei due autoscaler nella fase di evaluation, misurate sulle serie temporali di latenza e sul numero di repliche. Per ciascun episodio è stata calcolata la latenza media del servizio e il numero medio di repliche attive; a partire da tali valori sono stati ricavati gli indicatori di sintesi riportati.

Metrica	**Baseline**	**RL (Eval)**	**Interpretazione**
Latenza Media [s]	0,2901	0,2644	RL 8,9% più veloce
Repliche Medie	2,34	2,56	RL 9,4% più costoso
SLA Met [%]	**86,6**	**95,1**	**RL +8,5 punti**
Violazioni SLA [n]	11	10	1 violazione evitata
SLA Efficiency	0,370	0,371	Sostanzialmente pari

Tabella 1: Confronto quantitativo tra Baseline e RL. La metrica SLA Met indica la percentuale di episodi con latenza sotto 0,35s. L'efficienza è calcolata come rapporto SLA Met su repliche medie.

Il risultato più significativo emerge dall'analisi della metrica SLA Met, che misura la percentuale di episodi in cui la latenza media rimane al di sotto della soglia critica di violazione. L'agente RL raggiunge il 95,1% di SLA Met rispetto all'86,6% della Baseline, con un guadagno di 8,5 punti percentuali di episodi "in regola" con il vincolo di QoS. In termini operativi, questo dato indica una maggiore robustezza dell'approccio RL nella gestione dei picchi: pur lavorando sullo stesso pattern di traffico, l'agente impara policy che riducono la frequenza (o la durata aggregata) delle violazioni.

Questo vantaggio qualitativo si riflette anche nella latenza media, che passa da 0,290 s della Baseline a 0,264 s con l'agente RL, con una riduzione relativa dell'8,9%. Il guadagno assoluto in secondi può apparire contenuto, ma in un contesto Edge – dove le richieste sono spesso interattive e time-critical – una riduzione sistematica di alcune decine di millisecondi per richiesta rappresenta un miglioramento non trascurabile, specialmente se considerato su orizzonti temporali lunghi o su volumi elevati di traffico.

Il prezzo di questa maggiore stabilità è un incremento del 9,4% nel numero medio di repliche (da 2,34 a 2,56). Questo overhead non è casuale, ma riflette la policy conservativa appresa dall'agente: la reward function, penalizzando severamente le violazioni SLA, incentiva il mantenimento di un margine di capacità che permetta di assorbire i picchi più aggressivi. In altre parole, l'agente preferisce utilizzare qualche replica in più pur di evitare lunghi periodi di degrado del servizio.

Infine, l'indicatore di SLA Efficiency – definito come rapporto tra SLA Met e numero medio di repliche – assume valori molto simili per i due approcci (circa 0,37). Questo suggerisce che il sistema scala in maniera quasi lineare: l'agente RL investe circa il 9,4%

di risorse in più ottenendo un incremento comparabile in termini di qualità del servizio. L'allocazione aggiuntiva non si configura quindi come puro spreco, ma come un investimento mirato per ridurre in modo consistente la frequenza delle violazioni (dal 13,4% al 4,9% degli episodi).

6.3 Analisi Qualitativa

Per interpretare i dati aggregati presentati in precedenza, è utile osservare il comportamento dei due autoscaler nel dominio del tempo. Le figure 4 e 5 riportano l'andamento rispettivamente della latenza e del numero di repliche durante l'evaluation, in cui sono stati alternati momenti di carico variabile (evidenziati dagli sfondi colorati).

Analisi della Latenza (Fig.4) Osservando il grafico della latenza, si nota come entrambi gli approcci subiscano l'impatto dei picchi di traffico. La differenza principale non risiede tanto nell'evitare completamente il superamento della soglia (linea rossa tratteggiata), quanto nella capacità di recupero. La Baseline (linea arancione) tende a mantenere valori di latenza elevati per un intervallo di tempo maggiore dopo l'inizio del picco e questo comportamento influisce sulla metrica SLA Met: sebbene il numero di eventi di violazione possa essere simile tra i due approcci, la durata della violazione è mediamente più lunga per la baseline, portando a una percentuale di rispetto dello SLA inferiore (86,6%). L'Agente RL (linea blu), pur mostrando anch'esso dei picchi istantanei, tende a riportare i valori al di sotto della soglia critica più rapidamente, limitando l'impatto complessivo sulla qualità del servizio.

Analisi delle Repliche (Fig.5) Dal grafico sul numero di repliche emerge chiaramente una differenza marcata nelle politiche di scaling dei due approcci. Durante lo scenario critico "Primo Spike", l'agente RL dimostra una reattività superiore, portando le repliche al limite massimo (4) quasi istantaneamente per contenere il picco, d'altro canto, la Baseline, vincolata alla sua logica incrementale (*step-by-step*), reagisce invece con maggiore inerzia, raggiungendo la capacità necessaria solo in ritardo. Tuttavia, nelle fasi di carico intermedio ("Onda"), emerge il costo di questa aggressività: mentre la Baseline mantiene un comportamento più stabile, l'RL manifesta oscillazioni più frequenti. Questo comportamento suggerisce che l'agente ottimizza le risorse istante per istante, accettando una minore stabilità strutturale pur di garantire la massima reattività ai cambiamenti improvvisi.

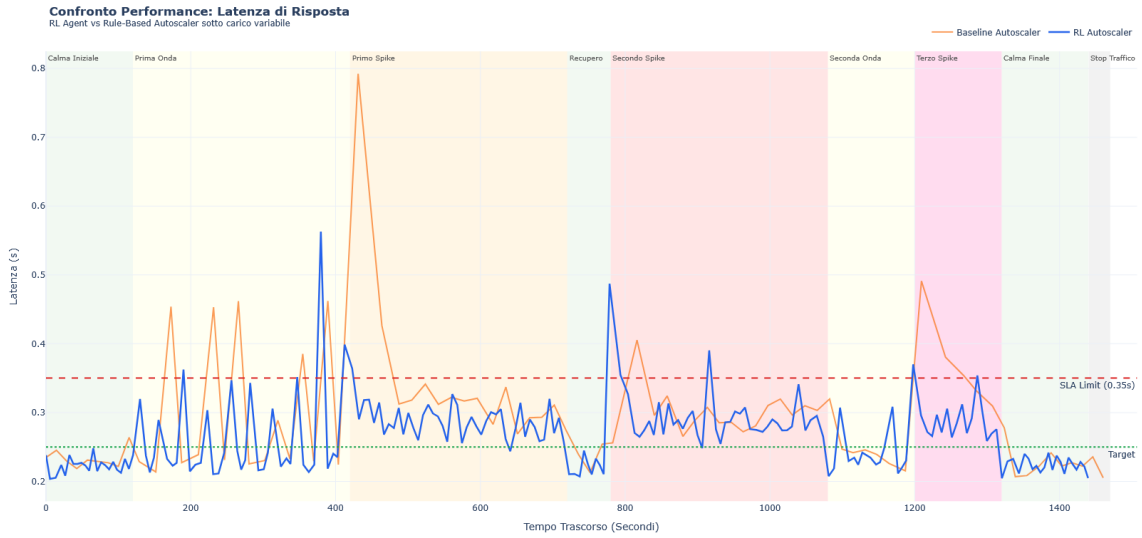

Figura 4: Confronto temporale della latenza. La linea rossa indica il limite SLA (0.35s).

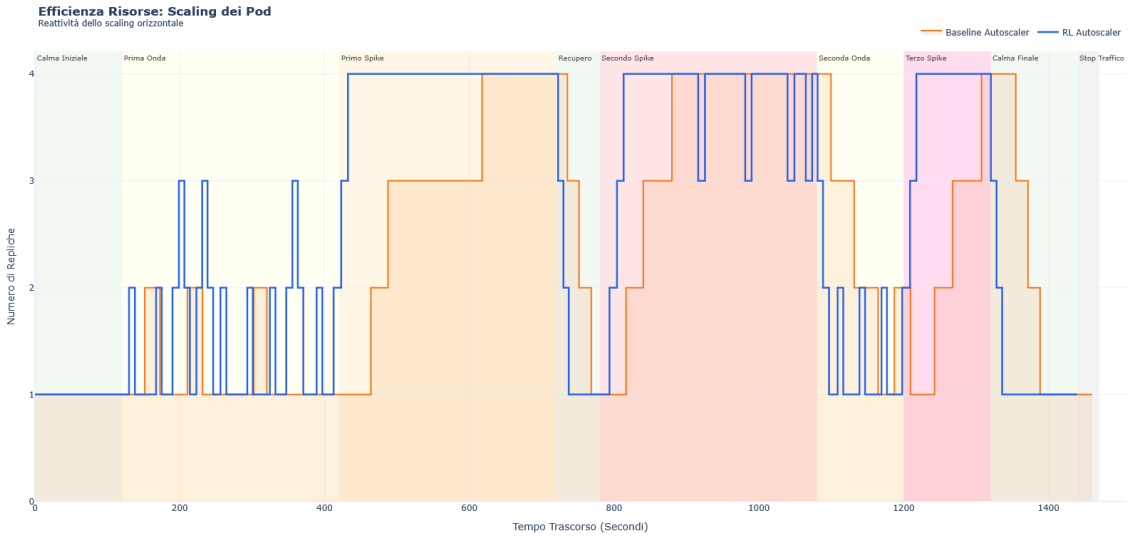

Figura 5: Confronto del numero di repliche attive. L'RL (blu) mostra una reazione più immediata ai picchi di carico.

6.4 Confronto Reward in Fase di Valutazione: RL vs Baseline

Per confrontare in modo oggettivo le prestazioni dell'autoscaler RL con la soluzione tradizionale è stato analizzato l'andamento del reward medio durante l'intera fase di valutazione. È importante sottolineare nuovamente che:

- l'agente RL utilizza la funzione di reward durante l'esecuzione per guidare direttamente le proprie decisioni;
- la Baseline rule-based, al contrario, applica semplici regole a soglia e non ha alcuna nozione interna di ricompensa.

Per poter confrontare i due approcci, alla Baseline è stato attribuito a posteriori un punteggio di reward calcolato sui log di valutazione, applicando la stessa funzione obiettivo usata per l'addestramento dell'agente RL.

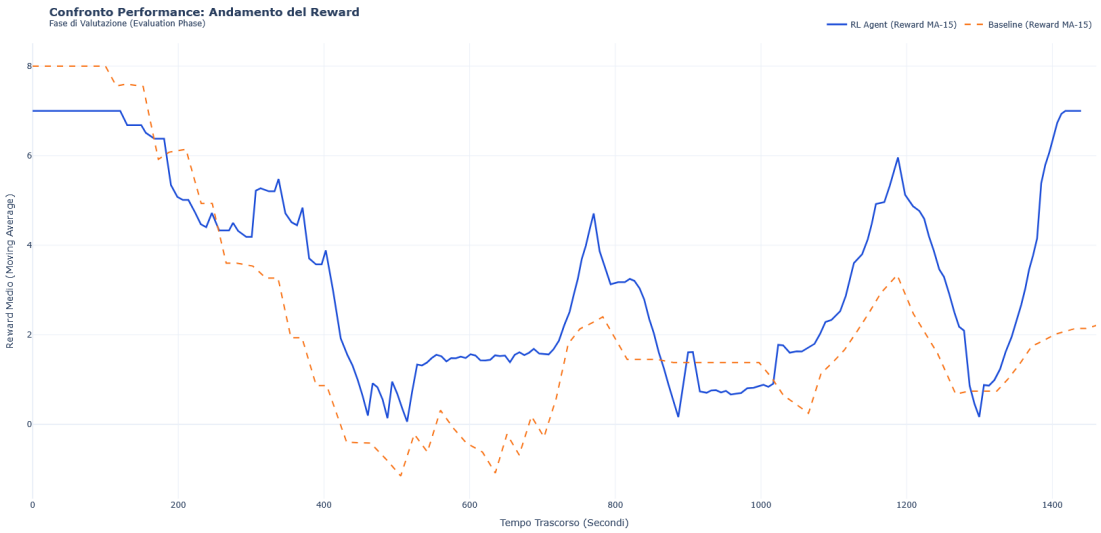

Figura 6: Andamento del reward medio in fase di valutazione: agente RL (blu) vs Baseline rule-based (arancione).

Dal grafico emerge una netta superiorità della politica appresa dall'agente RL rispetto alla logica reattiva della Baseline: la curva blu si mantiene per la maggior parte del tempo su valori di reward più elevati.
Analizzando più nel dettaglio le due traiettorie si osserva che:

- la Baseline presenta crolli profondi del reward in corrispondenza dei picchi improvvisi di carico. In queste fasi l'approccio a soglia fissa reagisce con ritardo allo stress del sistema, causando violazioni SLA marcate che vengono pesantemente penalizzate dalla funzione di reward;
- l'agente RL mostra un comportamento più proattivo: tendendo a mantenere un piccolo margine di sicurezza nel numero di repliche, riesce ad assorbire meglio le variazioni del traffico.

Nel complesso, l'analisi del reward in valutazione conferma quantitativamente quanto osservato sulle metriche di latenza e costo.

7 Discussione dei Risultati

L'analisi sperimentale conferma che l'approccio basato su Reinforcement Learning rappresenta un'alternativa solida e spesso superiore rispetto alla controparte tradizionale rule-based, in particolare nella gestione di pattern di carico variabili e picchi improvvisi, dove la capacità di anticipare e contenere le violazioni SLA risulta determinante. Il dato più rilevante è il miglioramento dell'**8,5%** nella metrica *SLA Met* (95,1% per RL contro 86,6% della Baseline). Questo non è dovuto a una maggiore potenza di calcolo, ma a una diversa gestione del rischio: mentre la Baseline attende il superamento della soglia per reagire, l'agente RL ha appreso una policy più reattiva.
L'agente accetta un leggero sovraccarico di risorse (+9,4% di repliche medie) per mantenere un buffer di sicurezza, prevenendo le violazioni invece di curarle. Tuttavia, questa reattività ha un costo in termini di stabilità: nelle fasi di carico medio-basso, l'RL mostra una tendenza a **oscillazioni** nel tentativo di ottimizzare continuamente le risorse, contrariamente alla maggiore inerzia della Baseline. In sintesi, l'RL si dimostra una buona scelta per scenari dove la latenza è prioritaria rispetto al risparmio energetico marginale.

8 Sviluppi Futuri

Per superare i limiti dell'attuale implementazione si propongono le seguenti direzioni di ricerca:

- **Deep Reinforcement Learning (DRL):** Sostituire la Q-Table tabellare con Reti Neurali (es. DQN o PPO) per gestire uno spazio degli stati continuo e includere metriche aggiuntive come l'uso di CPU, Memoria e Throughput di rete [10, 11].
- **Deployment su Traffico Reale:** Validare il sistema su un cluster cloud di produzione con traffico utente reale, caratterizzato da pattern meno prevedibili rispetto ai generatori sintetici.
- **Ottimizzazione Multi-Obiettivo:** Integrare il consumo energetico nella funzione di ricompensa, permettendo all'operatore di scegliere dinamicamente il punto di equilibrio tra performance (SLA) e sostenibilità.

9 Riferimenti bibliografici

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan. 2017. DOI: 10.1109/MC.2017.9

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, Oct. 2016. DOI: 10.1109/JIOT.2016.2579198

[3] Kubernetes Documentation, "Horizontal Pod Autoscaling," 2024. [Online]. Disponibile: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018. Disponibile: `http://incompleteideas.net/book/the-book-2nd.html`

[5] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. DOI: 10.1007/BF00992698

[6] Minikube Documentation, "Getting Started," 2024. [Online]. Disponibile: `https://minikube.sigs.k8s.io/docs/start/`

[7] Pallets Projects, "Flask Web Development," 2024. [Online]. Disponibile: `https://flask.palletsprojects.com/`

[8] Streamlit Inc., "Streamlit Documentation," 2024. [Online]. Disponibile: `https://docs.streamlit.io/`

[9] Docker Inc., "Docker Documentation," 2024. [Online]. Disponibile: `https://docs.docker.com/`

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. DOI: 10.1038/nature14236

[11] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016. DOI: 10.1038/nature16961