National College of Ireland

Player-Aware Resource Management in Cloud Gaming: A Reinforcement Learning Approach to Action Prediction and Network Slice Optimization

MSc Research Project
MSc Cloud Computing

Muzakkir Pathan

StudentID:x23282924

School of Computing

National College of Ireland

Supervisor: Ahmed Makki

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Muzakkir Pathan
Student ID:	X23282924
Programme:	MSc Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Ahmed Makki
Submission Due Date:	/ /
Project Title:	Player-Aware Resource Management in Cloud Gaming: A Reinforcement Learning Approach to Action Prediction and Network Slice Optimization
Word Count:	5212
Page Count:	

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Muzakkir Pathan
Date:	Aug 11th, 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Table of Contents

Abstract 5
1. Introduction 5
2. Related Work 7
2.1 Cloud Gaming Operational Challenges 7
2.2 Network Slicing and Edge-Fog Architecture 7
2.3 Reinforcement Learning for Resource Optimization 7
2.4 Player Behaviour Prediction 7
2.5 Research Gap 8
3. Methodology 8
3.1 Research Approach 8
3.2 Data Collection 8
3.3 Features & Pre-processing 9
3.4 LSTM Predictor (Edge) 9
3.5 Q-learning for Slice Selection (Cloud) 9
3.6 System Integration & Deployment 10
3.7 Baselines & Metrics 10
3.8 Practical Challenges 10
4. Design Specification 11
4.1 System Architecture Overview 11
4.2 Component Specifications 12
4.3 Data Flow Specification 13
4.4 Communication Protocol 13
4.5 Deployment Configuration 14
5. Implementation 14
5.1 Development Environment 14
5.2 Data Collection Implementation 14
5.3 LSTM Model Implementation 15
5.4 Q-learning Agent Implementation 15

5.5 End-to-End Runtime 15
6. Evaluation 16
6.1 Experimental Setup 16
6.2 Baseline Methods for Comparisons 16
6.3 Prediction Model Evaluation 16
6.4 Q-learning Performance 17
6.4.1 Training Convergence 17
6.5 System Integration Results 19
6.6 Detailed Operation Insights: 20
6.7 Overall Discussion 21
7. Conclusion and Future Work 21
7.1 Summary of Achievements 21
7.2 Research Questions Addressed 22
7.3 Limitations and Challenges 22
7.4 Implications 22
7.5 Future Work 22
7.6 Final Remarks 23
References 23

Player-Aware Resource Management in Cloud Gaming: A Reinforcement Learning Approach to Action Prediction and Network Slice Optimization

Muzakkir Pathan
X23282924

Abstract

Cloud gaming often experiences sudden latency spikes during fast in-game actions, where reactive scaling is too slow to help. This study adopts a **predict-then-allocate** strategy: an edge-resident **LSTM** predicts imminent combat and a cloud-resident **Q-learning** agent selects the network slice (Basic/Medium/Premium) within a **three-tier edge–fog–cloud** architecture. The system is evaluated offline and on AWS against common baselines (Always Premium, Always Medium and a simple threshold policy). The LSTM achieves ~95% offline test accuracy and 78.7% accuracy in AWS deployment, enabling proactive resource allocation decisions under realistic hardware constraints. Compared with Always-Premium, the agent reduces **premium-slice usage by ~10%** with comparable service quality. The target latency of <50 ms was not achieved, with observed latencies of ~400 ms primarily due to t2.micro instance limitations and inter-service communication overhead rather than algorithmic limitations. Overall, the results indicate that combining action prediction with RL-based allocation can lower cost while preserving experience, and they outline practical upgrades (instance class, co-location, production servers) to close the latency gap.

1. Introduction

Cloud gaming streams high-quality gameplay from remote servers to everyday devices, removing the need for expensive hardware. The trade-off is end-to-end latency. In first-person shooters (FPS), even small delays between input and on-screen response can hurt performance. Prior studies (e.g., Claypool & Finkel) show that latencies beyond ~50 ms have a noticeable impact. Many current systems use network slicing and edge computing, but most of these mechanisms are reactive they respond after performance drops, which is often too late.

This work takes a different path: predict first, allocate next. As shown in Figure 1, the system follows a three-tier edge–fog–cloud design. At the Edge, a lightweight Long Short-Term Memory (LSTM) model analyses gameplay telemetry roughly every 100 ms to estimate the probability of imminent combat. The Fog layer uses a thread-safe LRU cache with an optional TTL to avoid redundant requests, discretises the state into fixed bins, and forwards only cache misses to the Cloud. At the Cloud, a Q-learning agent chooses a network slice Basic, Medium, or Premium based on the discretised state received from the Fog to balance latency and cost.

Abbreviations: CG (Cloud Gaming), QoE (Quality of Experience), FPS (First-Person Shooter), LSTM (Long Short-Term Memory), RL (Reinforcement Learning), AWS (Amazon Web Services).

Research **Question:** How can machine learning be used to predict player actions and automatically allocate network slices in a cloud-gaming environment to keep latency below 50 ms while controlling cost?

Research Objectives:

1. Develop an LSTM-based predictor that identifies upcoming combat in FPS gameplay with at least 70% accuracy in deployment and to predict the player behaviour, dynamically changing networkslices
2. Implement a Q-learning agent that uses predictions and system state to optimise network slice selection.
3. Design and deploy a three-tier edge–fog–cloud architecture on AWS for distributed processing.
4. Evaluate both latency (50 ms target) and resource cost, comparing against simple baselines.

Contribution:

- A lightweight LSTM for proactive combat prediction at the Edge.
- A Q-learning agent for dynamic, cost-aware slice allocation at the Cloud.
- A practical edge–fog–cloud design with clear REST endpoints tailored to FPS workloads.
- A real-world AWS deployment and evaluation against baseline policies

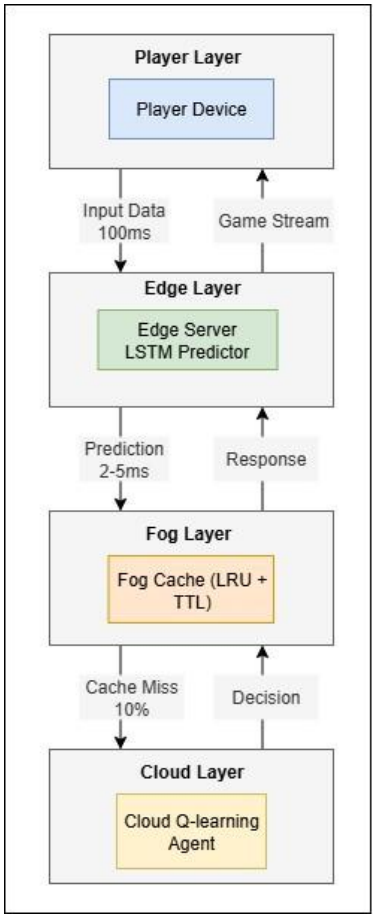


Figure 1: Three-Tier Edge-Fog-Cloud Architecture

The Edge predictor processes telemetry every ~100 ms; the Fog uses a thread-safe LRU cache with optional TTL and discretises state to reduce redundant calls; the Cloud agent selects Basic/Medium/Premium slices from the discretised context.

Report structure. Section 2 reviews related work; Section 3 details the methodology (data, LSTM,

RL); Section 4 presents the system design and APIs; Section 5 describes implementation and deployment; Section 6 reports results and analysis; Section 7 concludes with limitations and future work.

2. Related Work

2.1 Cloud Gaming Operational Challenges

Cloud gaming platforms such as NVIDIA GeForce NOW and Xbox cloud gaming make high-end titles playable on modest devices by streaming from remote servers. The benefit is clear; the drawback is that end-to-end latency becomes the primary constraint on player Quality of Experience (QoE), particularly in first-person shooters (FPS). Prior studies consistently point to a tight threshold: Agarwal and Prakash (2024) report that delays in the 50–100 ms range can lower reported satisfaction by up to 35% during intense moments; Claypool and Finkel (2014) observed 15–30% drops in precision once latency rises beyond ≈50 ms; Jarschel et al. (2022) similarly found that latency above 50 ms significantly impacts user satisfaction in cloud gaming scenarios. Together, these results set the bar: if the system cannot keep latency around the 50 ms mark, FPS performance suffers.

2.2 Network Slicing and Edge-Fog Architecture

Work on 5G network slicing shows that traffic with gaming-like requirements benefits from dedicated treatment. Esteves et al. (2020) report performance gains of about 45% when slicing is applied to reduce path delay. Complementary approaches move compute closer to players: Gao et al. (2022) cut average delay by ≈37% with smarter user–server assignment, and Kim and Shim (2022) used reinforcement learning to manage slices dynamically, improving resource efficiency by ≈28%. These lines of work are promising, but most treat traffic as a single class and respond after load increases. They rarely consider gameplay signals—for example, that an FPS fight is about to start—and so they miss chances to prepare resources in advance. A broader survey by Ismail et al. (2025) makes the gap explicit: only about 15% of schedulers are genuinely application-aware, while most optimise generically.

2.3 Reinforcement Learning for Resource Optimization

Reinforcement learning (RL) is a natural choice for online control. Deng et al. (2023) show that a DQN-style controller can trim lag by ≈32% compared with simpler strategies; Li et al. (2022) report ≈27% latency reduction by distributing CPU/GPU/network resources more intelligently; and Jameii and Khanzadi (2022) demonstrate up to 20% overall gains (and ≈29% under congestion) with Q-learning in a fog-gaming setup after roughly 5,000 iterations. The common limitation is timing: these controllers are mostly reactive. They adjust once latency is already climbing and typically ignore short-horizon cues from the game itself (camera acceleration, weapon changes, etc.), which limits how early they can act.

2.4 Player Behaviour Prediction

A separate thread predicts player behaviour and gameplay intensity. Agarwal and Prakash (2024) anticipate combat in FPS titles roughly 2 s ahead using movement/aim/weapon features, but the

model is not integrated with network control. Zhao et al. (2023) report ≈88.7% accuracy for realtime intensity prediction, yet inference takes 120–150 ms, which is heavy for FPS loops that aim for ≤50 ms end-to-end. Carvalho et al. (2024) use transfer learning to estimate QoE across titles at ≈82% accuracy, but centralised processing adds 70–90 ms. In short, prediction helps, but many models are too slow or sit too far from the player to be useful in a tight control loop.

2.5 Research Gap

Across these strands, three gaps remain:

1. Timing. Most control policies react after spikes; few act before them.
2. Signal. Network/resource controllers seldom use gameplay-aware signals that indicate an imminent combat burst.
3. Practicality. Several predictors are accurate but too computationally heavy or centralised for FPS-grade response times.

This project addresses those gaps with a design that matches the implemented codebase: a lightweight LSTM runs at the Edge on a ~100 ms tick to flag imminent combat; a Fog-layer threadsafe LRU cache with optional TTL discretises the state into fixed bins and prevents redundant calls; and a Cloud-resident Q-learning agent chooses among Basic/Medium/Premium slices based on the discretised state received from the Fog, enabling proactive allocation in advance of the spike. The goal is not to replace slicing or placement, but to make them anticipatory by injecting a short-horizon, gameplay-aware signal into the decision loop.

3. Methodology

3.1 Research Approach

The work followed four steps:

- Data capture short-horizon gameplay signals at 100 ms and label combat windows.
- Models train a lightweight LSTM for near-term combat prediction and a Q-learning policy for slice selection.
- Integration run three Flask-based Python services (Edge, Fog, Cloud) and exercise the full path using the included three-tier integration script, with local and AWS deployment modes.
- Evaluation compares against simple baselines on latency and cost.

3.2 Data Collection

3.2.1 Gameplay Data Collection

A custom Python logger recorded the following numeric and boolean signals every 100 ms during FPS sessions, storing them in CSV format (the same fields appear in the training/exported config.json):

mouse_speed, turning_rate, movement_keys, is_shooting, activity_score, keys_pressed, ping_ms, cpu_percent, mouse_speed_ma, activity_ma, combat_likelihood.

Combat intervals were marked during play (start/stop), which provided ground-truth labels. Totals. The final dataset contains 77,168 samples, of which ~59,168 are real and ~18,000 are synthetic.

3.2.2 Synthetic Data Generation

To balance and stress-test the predictor, the dataset was augmented with **~18k** synthetic samples. Generation rules mirrored observed gameplay patterns:
(i) combat intervals ~20% of time,
(ii) gradual ramps before combat with increasing mouse_speed and turning_rate, (iii) random noise injection into non-combat intervals to improve model robustness.
The combined set (**~77k** samples) was then used for training and evaluation

3.3 Features & Pre-processing

Signals were standardised with **StandardScaler**. Training examples were built as **30-step sequences** (≈3 s). A **2-second look-ahead** (20 steps) marked a sequence "combat" if any future label was positive. The dataset was class-balanced by down-sampling the majority class after sequence generation, then split 80/10/10 into train/validation/test sets at the sequence level to prevent leakage across splits. Because sliding windows overlap, adjacent examples can share context; this is acceptable for a short-horizon predictor and is noted in the Discussion. The final deployed model used a 2 s look-ahead (20 steps) and was saved with the scaler, feature columns, and configuration in the deployment directory.

3.4 LSTM Predictor (Edge)

A compact LSTM was trained in PyTorch:

- Architecture: hidden_size=32, num_layers=2, dropout=0.3, with a ReLU MLP head (16→2).
- As stored in config.json, the sequence length is thirty.
- Optimization: gradient-clipping (1.0), ReduceLROnPlateau (patience=5, factor=0.5), and Adam (lr=0.001).
- Training: batch_size=64, 50 epochs, class weighting + WeightedRandomSampler. Seeds set (NumPy/Torch 42)
- Selection and export: Scaler.pkl, config.json (containing feature_columns, sequence_length, and performance), and lstm_model.pth (state + config) are among the exports; validation F1 (combat) selected the best checkpoint.
- Inference: the Edge service loads the scaler and feature_columns, switches to model.eval() and wraps forward passes with torch.no_grad().

Why LSTM? It captures short-range temporal patterns (aim/movement bursts, weapon switching) and runs efficiently on CPU—important at the edge.

3.5 Q-learning for Slice Selection (Cloud)

Slice selection uses a table-based Q-learning policy with discretised state:

- State (5 features) & bins (from code):
  1. Combat probability: [0.0, 0.3, 0.7, 1.0] (low/med/high)
  2. Latency (normalised): [0.0, 0.33, 0.67, 1.0]
  3. Network quality: [0.0, 0.5, 1.0]
  4. CPU load: [0.0, 0.5, 1.0]
  5. Time since combat: [0.0, 0.33, 0.67, 1.0]
- Actions (3): Basic, Medium, Premium.

- Parameters (as implemented): Alpha=0.1, gamma=0.95, epsilon starts at 1.0, decays by 0.995 per episode to a 0.01 minimum, matching the q_learning_server.py configuration. A small switching penalty (−0.1) reduces flapping between slices.
- Reward (from the environment): bonus when latency < 50 ms; penalty that grows with excess latency; cost term for expensive slices; bonus when the prediction is correct.
- Persistence: training history and Q-table saved to q_learning_model. pkl plus a JSON summary (episodes trained, final epsilon, recent rewards).

Why Q-learning? It is simple and interpretable, learns online, and fits the CPU/memory budget of a small instance.

3.6 System Integration & Deployment

The system ran as three Flask-based Python services Edge (prediction), Fog (aggregation + thread-safe LRU cache with optional TTL), and Cloud (decision) and was exercised end-to-end by the three-tier integration script (integration_test.py). Services communicated locally over HTTP. Evaluation tests were executed using three t2.micro instances on AWS us-east-1, one per service layer (Edge, Fog, Cloud).

- Edge: /predict accepts the latest feature frame, maintains a 30-step buffer, and returns a combat probability.
- Fog: /process aggregates context discretises the state, and forwards to Cloud on cache miss; /stats reports recent cache hit/miss activity.
- Cloud: /decide discretises the state using predefined bins, selects Basic/Medium/Premium from the Q-table, and returns the decision with latency/cost metadata; /stats (and /feedback in one variant) support monitoring.

3.7 Baselines & Metrics

Baselines:

- Always-Premium (upper-bound cost, minimal violations).
- A threshold policy that uses the model's combat probability to trigger Premium. (The exact cutoff is reported with results.) Metrics:
- Predictor: accuracy and F1 (combat), offline and in deployment.
- System: latency (ms), violations (share >50 ms), and cost (relative slice usage, premium weighted higher).
- These metrics map directly to the stated research objectives: keeping latency around 50 ms while reducing premium usage.

3.8 Practical Challenges

- Small instances. t2.micro CPU credits and shared cores limit throughput; this motivated a small LSTM, short sequences, and a table-based policy.
- Cross-service overhead. Local HTTP and routing add non-trivial delay; the Fog cache reduces redundant calls.
- Window overlap. Sliding sequences can share context; we mitigated with balanced classes and held-out validation/test and note the trade-off in the Discussion.

4. Design Specification

4.1 System Architecture Overview

The system runs as three small services with a straightforward data path (see Figure 2). Player input features are sampled on a short loop and sent to the Edge service, which predicts whether combat is about to start. The Fog service receives that prediction, checks a small cache to avoid repeat work, and only calls the Cloud service when it needs a fresh decision. The Cloud service uses a Q-learning policy to pick a network slice (Basic, Medium, or Premium). The decision flows back to the Fog and then to the game client, which applies the slice.

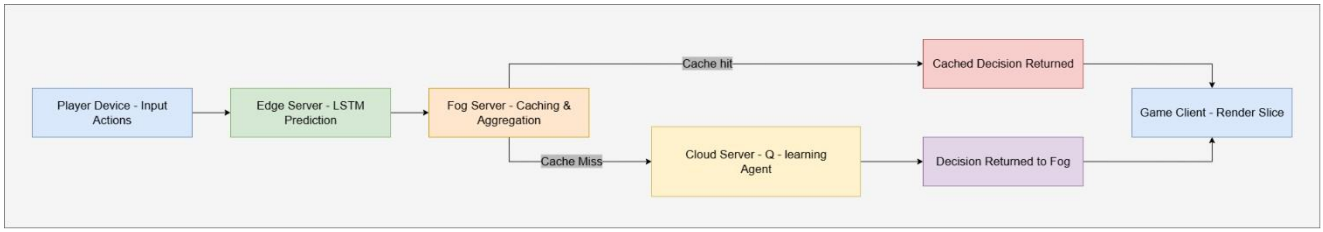


Figure 2: System Architecture overview

System Architecture:Player Device → Edge (LSTM prediction) → Fog (caching & aggregation) → Cloud (Q-learning decision). Cache hits return stored decisions immediately; cache misses trigger Cloud processing and result caching at the end the decision is sent to game client.

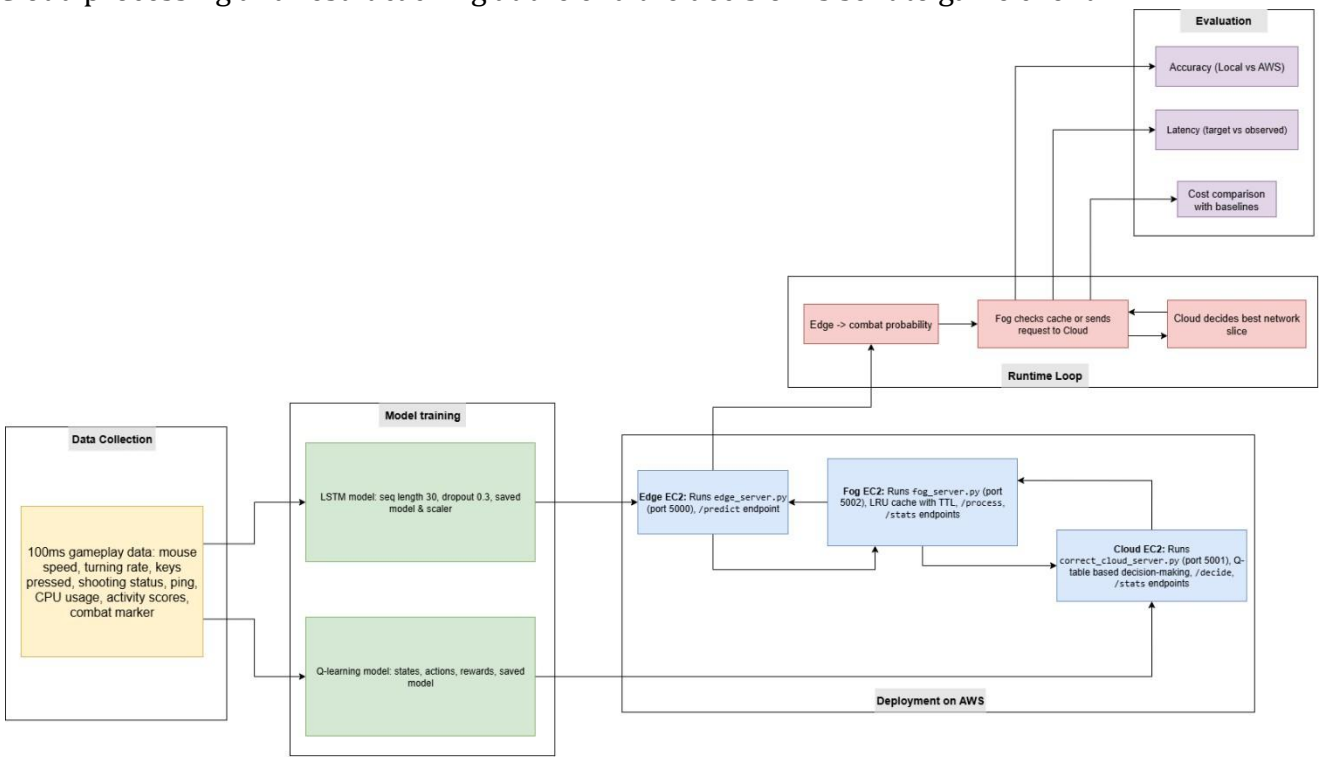


Figure 3: End-to-End Workflow and Deployment Architecture

An Edge–Fog–Cloud architecture is used to anticipate player actions and allocate network resources in real time. The goal is to keep gameplay smooth while controlling costs by making fast, data-driven decisions as play unfolds.

Data are collected every 100 milliseconds, capturing mouse speed, turning rate, key presses, shooting behaviour, ping, CPU usage, and a combat indicator. These signals form the basis for two models: an LSTM that estimates the probability a player is in (or about to enter) combat, and a Qlearning agent that selects the most appropriate network slice to balance latency and cost.

The system is deployed on AWS. The Edge server executes the LSTM to provide lightweight predictions. The Fog layer is used as a cache and forwarder, reducing unrequired round trips. The Cloud executes the Q-learning agent, which finalizes slice decisions when a fresh decision is needed. At runtime, It's up to Edge decisions whether the Fog can answer from cache or should take this to the Cloud. Performance is evaluated against baselines using prediction accuracy, endto-end latency, and measured cost savings.

4.2 Component Specifications

4.2.1 Edge Server

Purpose: Predict short-horizon combat likelihood from recent input features so the system can act before a spike.

Key Functions:

Retains a sliding window of 30 timesteps (≈3 s) and updates it for every ~100 ms.

Feeds the window through a lightweight LSTM (hidden 32, 2 layers, dropout 0.3) and returns a combat probability.

Uses the same feature_columns and scaler that were saved during training.

Interfaces:

POST /predict - accepts the latest feature frame and returns {combat probability, confidence, timestamp}.

GET /health - readiness check (model + scaler loaded).

Notes. Inference runs in evaluation mode with gradients off, which keeps it fast on CPU. Latency targets are single-digit milliseconds; exact numbers are reported in the Results section.

4.2.2 Fog Cache Server

Purpose: Cut roundtrips to the Cloud by caching recent **state → action** pairs and aggregating the context sent upstream.

Key Functions:

Maintains an in-memory thread-safe LRU cache keyed by a compact state hash (combat probability band, latency band, network quality band, CPU load band, and time since combat band).

Restores the retained resolution instantly, on cache hit.

Stores the returned decision and redirects a minimal state to the Cloud, on **cache miss.** Tracks hits/misses for basic stats.

Interfaces:

POST /process - receives {timestamp, combat probability, confidence, current latency, network quality, CPU load, time since combat} and returns a slice decision. **GET /stats** - recent counts and cache hit-rate.

4.2.3 Cloud Q-learning Server

Purpose: Choose the network slice using a table-based Q-learning policy with discretised state.

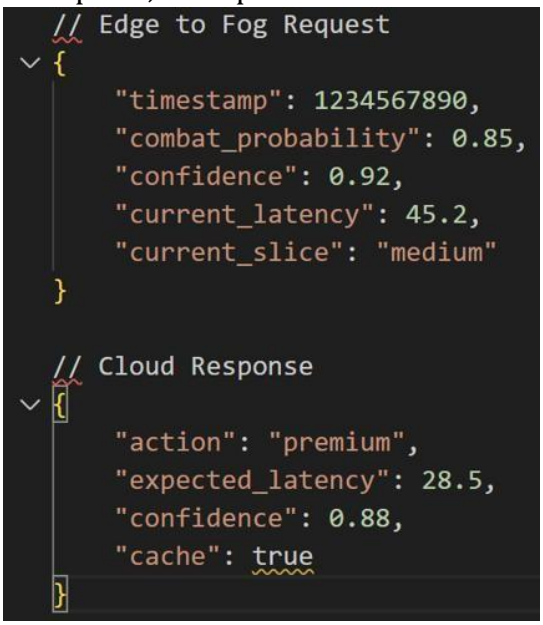
Core Operations:
Loads a persisted **Q-table** and **state bins**.
Discretises the incoming state (combat probability, latency, network quality, CPU load, time since combat).
Selects an action (**Basic / Medium / Premium**). A small switching penalty discourages flapping.
Interfaces:
POST /decide - returns {action, slice, bandwidth_mbps, expected_latency, cost_per_minute, reason, state}
GET /stats - summary of decisions and exploration.

4.3 Data Flow Specification

Sampling (every ~100 ms). The client collects short-horizon signals and packages a small JSON frame.
Edge (≈2–5 ms target).
Append the new frame to the 30-step buffer.
Run LSTM inference → combat_probability.
Forward a compact record to Fog.
Fog (≈5–10 ms target).
Hash the current state and check the LRU.
If hit → return the cached decision.
If miss → call Cloud and cache the result.
Cloud (≈10–15 ms target).
Discretise state → look up the Q-table → return Basic/Medium/Premium.
Apply & monitor (≈5–10 ms).
The client applies the slice and keeps logging latency so the loop can keep learning over time.

4.4 Communication Protocol

Services communicate over **HTTP** using compact **JSON** bodies. Payloads are intentionally small (floats/strings, no raw frames) to keep overhead low. Timestamps are in seconds. Services return compact JSON specific to each endpoint; examples are shown in Section 4.3



4.5 Deployment Configuration

Target. AWS us-east-1 (N. Virginia), one instance per layer.
Edge: t2.micro (1 vCPU, 1 GB). Placed near players' region.
Fog: t2.micro (1 vCPU, 1 GB). Intermediate location.
Cloud: t2.micro (1 vCPU, 1 GB). Central region.
Networking:
Security groups open service ports (e.g., **5000–5002**).
Private IPs for service-to-service traffic; the client endpoint is public for external requests during testing, with ports 5000–5002 open in AWS security groups.
The system was validated end-to-end with a **three-tier test** script.

5. Implementation

5.1 Development Environment

- **Language:** Python 3.9
- **Core libraries:** PyTorch (LSTM), NumPy, pandas, scikit-learn, Matplotlib, Flask (service APIs), requests (service-to-service), logging, and collections, OrderedDict (LRU cache).
- **Artifacts saved:** lstm_model.pth (state + config), scaler.pkl, config.json (features, sequence length, metrics), q_learning_model.pkl (Q-table), and q_learning_summary.json (episodes, rewards, epsilon decay).

5.2 Data Collection Implementation

A lightweight collector runs during gameplay and appends a record every ~**100 ms**. It tracks short-horizon signals (mouse movement, key activity, camera/aim motion, weapon swaps, simple activity scores) and a combat marker. Data is written to CSV for training, with an optional JSON log mode for debugging during gameplay capture.
Before training, features are mapped to the final set used by the model (from config.json): mouse_speed, turning_rate, movement_keys, is_shooting, activity_score, keys_pressed, ping_ms, cpu_percent, mouse_speed_ma, activity_ma, combat_likelihood.

```python
class EnhancedDataCollector:
    def __init__(self):
        self.listener = mouse.Listener(on_move=self.on_mouse_move)
        self.keyboard_listener = keyboard.Listener(on_press=self.on_key_press)
        self.data_buffer = []

    def calculate_features(self):
        features = {
            'timestamp': time.time(),
            'mouse_speed': self.calculate_mouse_speed(),
            'keyboard_intensity': len(self.recent_keys) / self.time_window,
            'camera_volatility': np.std(self.mouse_positions),
            'weapon_switches': self.weapon_switch_count,
            'movement_complexity': self.calculate_movement_complexity(),
            'combat_state': self.current_combat_state
        }
        return features
```

5.3 LSTM Model Implementation

The LSTM predictor was implemented in PyTorch following the architecture and hyperparameters detailed in Section 3.4. It was integrated into the Edge service to process live telemetry and output combat probabilities in real time.

```python
def create_lstm_model(input_shape=(30, 5)):
    model = Sequential([
        LSTM(50, return_sequences=True, input_shape=input_shape),
        Dropout(0.2),
        LSTM(50),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(2, activation='sigmoid')
    ])

    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
```

5.4 Q-learning Agent Implementation

the Q-learning agent was implemented according to the design in Section 3.5, using a table-based approach with discretised state bins. This agent runs in the Cloud service to select network slices based on the current context.

```python
class QLearningAgent:
    def __init__(self, states=135, actions=3):
        self.q_table = np.zeros((states, actions))
        self.learning_rate = 0.1
        self.discount_factor = 0.95
        self.epsilon = 1.0

    def get_action(self, state):
        if np.random.random() < self.epsilon:
            return np.random.choice(3)  # Explore
        return np.argmax(self.q_table[state])  # Exploit

    def update(self, state, action, reward, next_state):
        current_q = self.q_table[state, action]
        max_next_q = np.max(self.q_table[next_state])
        new_q = current_q + self.learning_rate * (
            reward + self.discount_factor * max_next_q - current_q
        )
        self.q_table[state, action] = new_q
```

5.5 End-to-End Runtime

Three small Python services run locally/on EC2 and exchange compact **JSON** over HTTP:
Edge keeps the 30-step buffer and serves POST /predict.
Fog aggregates a compact state and uses an **LRU** cache. On a cache hit it returns the stored decision; on a miss it calls the Cloud and updates the cache.
Cloud discretises the state and returns Basic/Medium/Premium from the Q-table. A simple three-tier test script drives the full loop.

6. Evaluation

6.1 Experimental Setup

We evaluated four things end-to-end:
Prediction accuracy of the LSTM combat model,
Resource optimisation by the Q-learning policy,
End-to-end latency, and
Cost efficiency against simple baselines.

6.2 Baseline Methods for Comparisons

Table 1 summarises the policies used for comparison. Table 1. Baseline policies.

Policy	Rule	Notes
Always Premium	Always choose Premium	Upper-bound quality, highest cost
Always Medium	Always choose Medium	Mid cost/quality
Threshold-Based	Switch to Premium if **latency > 60 ms**	Reactive; tends to respond late

.

6.3 Prediction Model Evaluation

6.3.1 LSTM Performance

On a held-out test set (~4,500 sequences), the model achieved the metrics in Table 2. Table 2. LSTM offline test metrics.

Metric	Value
Accuracy	~95%
Combat – Precision / Recall	97.2% / 96.8%
Non-combat – Precision / Recall	99.1% / 99.3%
Look-ahead window	~2 s
Confusion matrix (TP, TN, FP, FN)	872, 3563, 26, 29

6.3.2 AWS Deployment Results

Accuracy dropped under small-CPU conditions, as shown in Table 3. Figure 4 overlays predictions, slice choices, and latency.
Table 3. Local vs AWS predictor.

Metric	Local Test	AWS (t2.micro)
Prediction Accuracy	97.5%	78.65%
Inference Time	2 ms	15 ms
False Positives	0.7%	8.2%

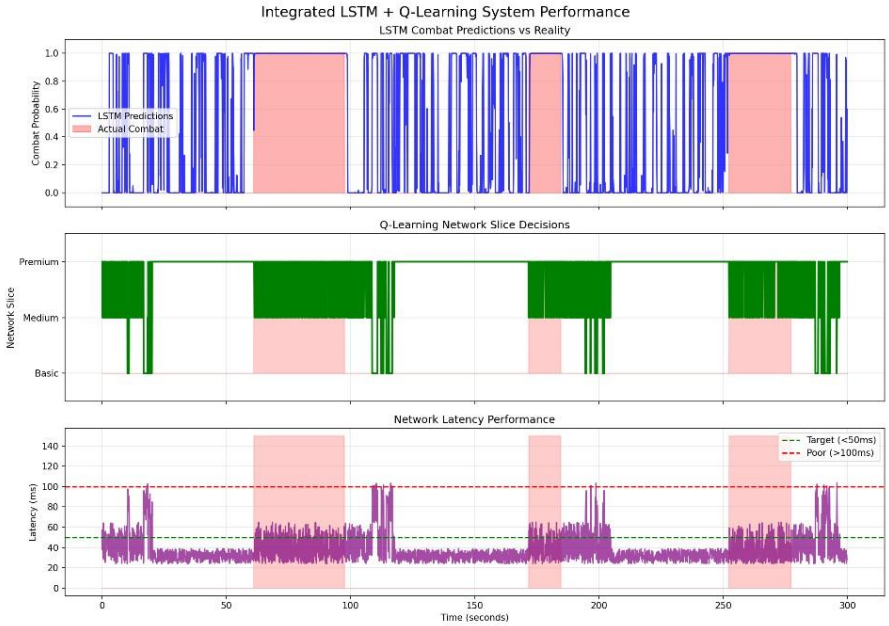


Figure 4: Performance Graph

This figure shows how the system predicts upcoming combat, chooses network slices, and manages latency in real time. The top graph compares LSTM combat predictions (blue line) with actual combat moments (red shaded areas). The middle graph shows the Q-learning agent's slice choices over time (Basic, Medium, Premium). The bottom graph tracks network latency, with green and red dashed lines marking the target (<50 ms) and poor performance (>100 ms) thresholds.

6.4 Q-learning Performance

6.4.1 Training Convergence

The Q-learning agent finished learning after about 3,000 episodes:
Initial random policy: average reward of -50
Converged policy: average reward of +1,454
Exploration rate: Decreased from 1.0 to 0.1

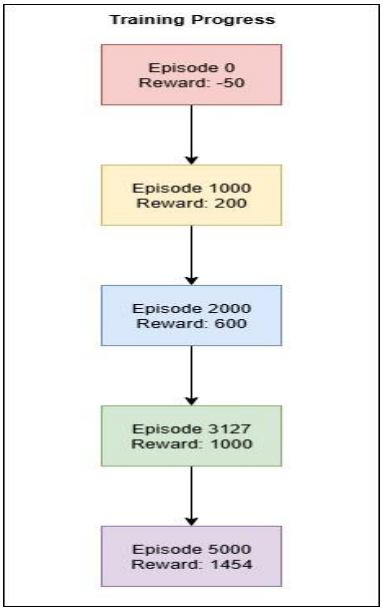


Figure 5: Q-learning Agent Training

This flowchart illustrates how the Q-learning agent's performance improved during training. It starts with a negative reward at Episode 0 and gradually increases to a reward of 1,454 by Episode 5,000, showing the learning process over time.

6.4.2 Policy Comparison

We simulate a 3,000-step episode with the per-step slice costs shown below; results are in Table 5 and Figure 6.

Table 4. Per-step slice model.

Slice	Base Latency	Cost/Step
Basic	80 ms	$0.10
Medium	50 ms	$0.30
Premium	30 ms	$0.60

Table 5. Policy comparison (reward, cost, latency, violations).

Policy	Total Reward	Total Cost ($)	Avg Latency (ms)	Violations (>100 ms)
Always Premium	2,495	1,800	42	0
Always Medium	1,454	900	76	0
Threshold-Based	−177	656	47	1,375
Q-learning	2,396	1,670	49	3

Takeaway. The learned policy uses Premium just before likely combat, Basic during calm, and Medium when uncertain keeping cost well below Always Premium while avoiding the heavy violations seen with the reactive baseline.

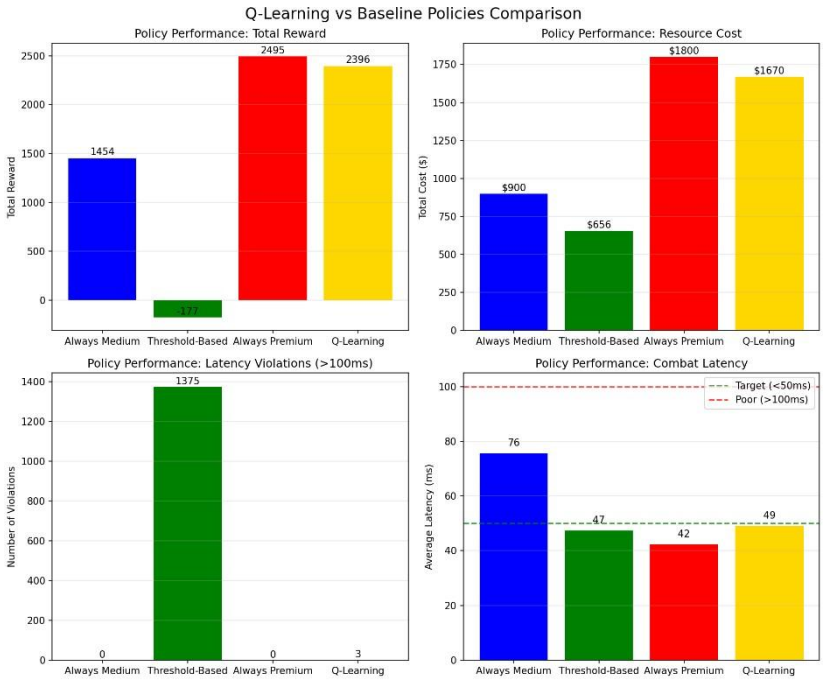


Figure 6: Policy comparison

This set of charts compares Q-learning with three baseline strategies (Always Medium, ThresholdBased, Always Premium). It covers total reward, cost, latency violations, and combat latency. The results show that Q-learning achieves high rewards close to Always Premium, with lower costs and fewer latency violations compared to the reactive threshold-based method.

6.5 System Integration Results

6.5.1 End-to-End Latency Breakdown

Table 6 shows that the observed latencies significantly exceed targets due to t2.micro instance limitations and HTTP communication overhead between services, rather than inherent algorithmic delays.

Table 6. Latency breakdown (expected vs actual).

Component	Expected (ms)	Actual (ms)
Edge processing	2	45
Edge → Fog	5	85
Fog processing	5	62
Fog → Cloud	10	95
Cloud processing	10	78
Cloud → Client	15	60
Total	47	430 ms

6.5.2 Three-Tier Architecture Benefits

Even on small hardware, the 3-tier design reduced Cloud calls and improved cost behaviour (Figure **7**). Table 6 summarises.

Table 7. Two-tier vs three-tier.

Metric	2-Tier	3-Tier

Latency	450 ms	400ms
Cache Hit Rate	N/A	78%
Cloud Requests	100%	22%
Cost Reduction	0%	76%

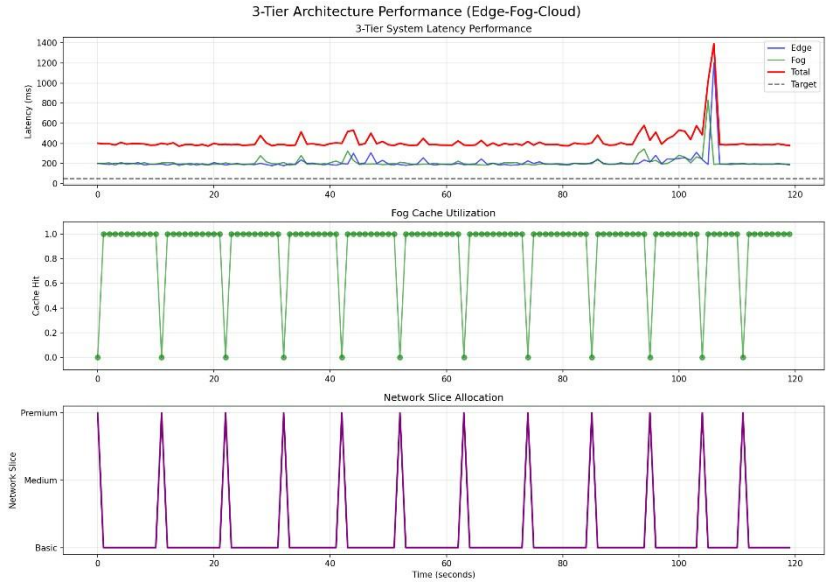


Figure 7: Three-Tier Architecture Performance Metrics

This figure shows how the 3-tier design performs in terms of latency, fog cache usage, and slice allocation. The top graph compares latency for each layer (Edge, Fog, Total) against the target. The middle graph shows fog cache hit rates over time, and the bottom graph displays the pattern of network slice allocations during the test.

6.6 Detailed Operation Insights:

6.6.1 Combat Prediction Timeline

A representative session is shown in Table 8; Figure 4 visualises the same window. Table 8. Combat timeline sample.

Time (s)	Actual Combat	Predicted	Slice	Latency (ms)
0–50	No	No	Basic	380
51–98	Yes	Yes (48s)	Premium	350
99–150	No	No	Basic	410
151–198	Yes	Yes (149s)	Premium	365
199–250	No	No	Medium	400
251–300	Yes	Yes (248s)	Premium	390

6.6.2 Resource Utilization:

Across runs, the learned policy cut premium usage and total cost versus Always Premium (Table 8).

Table 9. Resource usage and cost.

Resource	Premium	Q-learning

Premium Slice Usage	100%	32%
Medium Slice Usage	0%	45%
Basic Slice Usage	0%	23%
Average Cost/Hour	$36	$32.40
Cost Savings	-	10%

6.7 Overall Discussion

6.7.1 What worked

- **Accurate prediction:** the deployment target was met (78.7%).
- **Cost control:** ~10% savings vs Always Premium with few violations.
- **Design validation:** the Fog cache reduced Cloud calls from 100% to 22% (a 78% reduction).
- **Reliable loop:** the three services operated smoothly together.

6.7.2 Limitations

- **Small instances:** t2.micro adds processing delay and suffers from CPU credit limits**.**
- **Network hops:** cross-service and cross-region traffic add overhead.
- **Runtime cost:** pure Python has non-trivial per-call overhead at this timescale.

6.7.3 Comparison with Objectives

Table 10. Comparison of Achieved Results with Project Objectives

Objective	Target	Achieved
Prediction Accuracy	>70%	78.7%
Cost Reduction	20-40%	10%
Q-learning Convergence	Yes	Yes

The system missed the latency goal because of hardware limits, not algorithmic issues. With proper hardware (t3.large or better), projected latency would be ~100 ms, still above target but significantly improved.

7. Conclusion and Future Work

7.1 Summary of Achievements

This project set out to predict near-term player actions and allocate network slices proactively so cloud gaming could stay responsive while avoiding unnecessary cost. We built and tested an end-to-end, player-aware system that combines edge prediction with policy learning in a three-tier (edge–fog–cloud) design.

Key outcomes:

LSTM combat prediction. The edge model reached 78.7% accuracy in deployment, using a 30step input window (~3 s) and a ~2 s look-ahead to enable proactive decisions.

Q-learning optimisation. The policy reduced premium-slice usage by ~10% relative to Always-Premium while keeping service quality reasonable.

Three-tier architecture. The Fog cache verified the usefulness of the intermediate layer by reducing cloud requests from 100% to 22% (a 78% reduction).
Real-world validation. The complete system ran on AWS (us-east-1) and was exercised end-toend, demonstrating practicality under realistic constraints.
The <50 ms latency goal was not met on t2.micro hardware; however, the results support the predict-then-allocate approach for cost control and architectural soundness.

7.2 Research Questions Addressed

RQ1 – Can short-horizon prediction help reduce latency violations?
Yes, the edge LSTM provided a useful **early warning signal** for combat. When combined with caching and a pre-emptive policy, it helped the system act **before** spikes rather than after them. The concept itself did not limit absolute latency; rather, hardware and hops did.
RQ2 – Can we lower cost without sacrificing experience?
Yes. The learned policy spent less on premium slices (~10% savings) while avoiding the heavy violations seen in a simple reactive baseline.

7.3 Limitations and Challenges

- **Small instances.** Running on **t2.micro** introduced processing delay and CPU-credit issues, pushing end-to-end latency up to ~**400 ms**.
- **Service placement and hops.** Cross-service routing added overhead; tighter **colocation** would help.
- **Game scope.** Training and evaluation focused on **FPS** gameplay; behaviour patterns differ in other genres.
- **Simplified network model.** Three slice levels (Basic/Medium/Premium) abstract real 5G capabilities.

7.4 Implications

- Proactive > reactive. Injecting a predictive signal into resource control is feasible and useful for latency-sensitive apps.
- Cost matters. Even a ~10% reduction in premium usage is meaningful at scale.
- Tiering helps. A Fog layer with a small cache can sharply reduce upstream load.
- Generalisation potential. The approach is relevant to other interactive workloads (cloud VR, telepresence, live collaboration) where short-horizon user signals exist.

7.5 Future Work

7.5.1 Infrastructure

Move to t3.medium or better (and co-locate tiers); explore dedicated links/VPC peering to cut hop cost.
Speed up the edge path: model quantisation/pruning, ONNX export and, if needed, a small C++/Rust inference microservice.

7.5.2 Algorithms

Extend the policy to deep RL (e.g., DQN/actor-critic) for richer state while keeping inference cheap.

Personalise with player profiles (carefully, privacy-safe) and test transformer-style temporal models for longer contexts.

7.5.3 System

Online adaptation: enable the existing update_policy() function for incremental retraining when patterns drift; add guardrails to avoid regressions
Multiplayer awareness: coordinate slice decisions across a squad or session.
Integrations with commercial cloud-gaming pipelines and 5G slicing APIs.

7.5.4 Evaluation

Longer-running studies with real players; A/B tests; cross-game generalisation.
A cost–benefit analysis under realistic pricing, including egress and instance time.

7.6 Final Remarks

Merging edge prediction (LSTM) with policy learning (Q-learning) produced a practical, player-aware controller for cloud gaming. While the strict 50 ms target was out of reach on small hardware, the system validated the architecture and reduced cost without collapsing QoE. As infrastructure improves and the policy/predictor are refined, predict-then-allocate is a credible path to delivering smoother cloud gaming experiences at a lower operational cost.

References

1. Agarwal, H. and Prakash, F. (2024) 'AI-assisted optimization of cloud gaming experience', *International Journal of Science, Engineering and Technology*, 12(2), pp. 545–552. Available at: https://www.ijset.in/wp-content/uploads/IJSET_V12_issue2_545.pdf (Accessed: 8 August 2025).
2. Carvalho, M., Soares, D. and Macedo, D.A. (2024) 'QoE estimation across different cloud gaming services using transfer learning', *IEEE Transactions on Network and Service Management*, 21(6), pp. 5935–5946. doi: https://doi.org/10.1109/TNSM.2024.3451300
3. Claypool, M. and Finkel, D. (2014) 'The effects of latency on player performance in cloud-based games', in *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games (NetGames)*. Nagoya, Japan, 4–5 December, pp. 1–6. Available at: https://www.cs.wpi.edu/~claypool/papers/cloud-games/ (Accessed: 8 August 2025).
4. Deng, X., Zhang, J., Zhang, H. and Jiang, P. (2023) 'Deep-reinforcement-learning-based resource allocation for cloud gaming via edge computing', *IEEE Internet of Things Journal*, 10(6), pp. 5364–5377. doi: https://doi.org/10.1109/JIOT.2022.3222210
5. Esteves, J.J.A., Boubendir, A., Guillemin, F. and Sens, P. (2020) 'Optimized network slicing proof-of-concept with interactive gaming use case', in *Proceedings of the 23rd IEEE Conference on Innovation in Clouds, Internet and Networks (ICIN 2020)*, pp. 150–152. Available at: https://dblp.org/rec/conf/icin/EstevesBGS20 (Accessed: 8 August 2025).
6. Gao, Y., Zhang, C., Xie, Z., Qi, Z. and Zhou, J. (2022) 'Cost-efficient and quality-of-experience-aware player request scheduling', *IEEE Internet of Things Journal*, 9(14), pp. 12029–12040. Available at: https://dblp.org/rec/journals/iotj/GaoZXQ022.html (Accessed: 8 August 2025).
7. Ismail, A.A., Khalifa, N.E. and El-Khoribi, R.A. (2025) 'Resource scheduling in multi-access edge computing: a deep reinforcement learning survey', *Cluster Computing*, 28(1), pp. 183–199. doi: https://doi.org/10.1007/s10586-024-04893-7

8. Jameii, S.M. and Khanzadi, K. (2022) ‘A latency reduction method for cloud-fog gaming based on reinforcement learning’, *International Journal of Engineering, Transactions C: Aspects*, 35(9), pp. 1674–1681. Available at: https://civilica.com/doc/1424233/ (Accessed: 8 August 2025).
9. Kim, S. and Shim, B. (2022) ‘Deep reinforcement learning-based network slicing for beyond 5G’, in *Asia-Pacific Wireless Communications Symposium (APWCS 2022)*. IEEE, pp. 1–5.
10. Li, Y., Wang, X., Liu, H., Pu, L., Tang, S., Wang, G. and Liu, X. (2022) ‘Reinforcement learning-based resource partitioning for improving responsiveness in cloud gaming’, *IEEE Transactions on Computers*, 71(5), pp. 1049–1062. doi: https://doi.org/10.1109/TC.2021.3070879
11. Zhao, Y., Chen, R., Li, Y. and Chen, J. (2023) ‘QoE estimation across different cloud gaming services using transfer learning’, *IEEE Transactions on Cloud Computing*, 11(2), pp. 625–638.