smart cities

MDPI

Article

LLM-Driven Offloading Decisions for Edge Object Detection in Smart City Deployments

Xingyu Yuan and He Li *

Department of Sciences and Informatics, Muroran Institute of Technology, Muroran 050-8585, Japan; 23096502@muroran-it.ac.jp
* Correspondence: heli@mmm.muroran-it.ac.jp

Highlights

What are the main findings?

- We introduce an LLM-driven framework that auto-generates and refines DRL reward functions from natural language objectives for edge offloading in object detection.
- On a real-world dataset, policies trained with LLM-generated rewards achieve higher throughput and lower process latency than expert-designed rewards.
- DRL policies can be retargeted across objectives (e.g., latency, energy, and accuracy) using prompts only, eliminating manual reward redesign.

What is the implication of the main finding?

- Engineering overhead is reduced while accelerating the adaptation of edge AI services when goals or conditions change.
- The robustness of offloading and scheduling improves under heterogeneous workloads, fluctuating bandwidths, and dynamic device capabilities.

Abstract

Object detection is a critical technology for smart city development. As request volumes surge, inference is increasingly offloaded from centralized clouds to user-proximal edge sites to reduce latency and backhaul traffic. However, heterogeneous workloads, fluctuating bandwidth, and dynamic device capabilities make offloading and scheduling difficult to optimize in edge environments. Deep reinforcement learning (DRL) has proved effective for this problem, but in practice, it relies on manually engineered reward functions that must be redesigned whenever service objectives change. To address this limitation, we introduce an LLM-driven framework that retargets DRL policies for edge object detection directly through natural language instructions. By leveraging understanding of the text and encoding capabilities of large language models (LLMs), our system (i) interprets the current optimization objective; (ii) generates an executable, environment-compatible reward function code; and (iii) iteratively refines the reward via closed-loop simulation feedback. In simulations for a real-world dataset, policies trained with LLM-generated rewards adapt from prompts alone and outperform counterparts trained with expert-designed rewards, while eliminating manual reward engineering.

Keywords: large language model; edge computing; YOLO; object detection; deep reinforcement learning; offloading; smart city

Academic Editor: Pierluigi Siano

Received: 15 August 2025
Revised: 6 October 2025
Accepted: 7 October 2025
Published: 10 October 2025

Citation: Yuan, X.; Li, H. LLM-Driven Offloading Decisions for Edge Object Detection in Smart City Deployments. *Smart Cities* **2025**, *8*, 169. https://doi.org/10.3390/smartcities8050169

1. Introduction

In the contemporary era, the development of Internet of Things (IoT)-enabled smart cities has emerged as a global priority [1–3]. In this context, object detection, namely the

automatic identification and localization of objects such as vehicles, pedestrians, or road hazards from visual data, serves as a pivotal technology that underpins traffic monitoring, public safety, and urban planning [4–6]. Figure 1 illustrates a representative example, where dashcam footage is processed by nearby edge devices to detect events such as potholes, rock falls, or car crashes, enabling timely alerts for urban management and road safety. Precise detection and localization of objects in dynamic urban environments are indispensable for deriving actionable insights from high-volume video and sensor data, a requirement that often demands low-latency, real-time processing [7,8]. Due to their computationally demanding nature, state-of-the-art object detection models are increasingly deployed on edge devices (i.e., computation units positioned close to data sources, such as roadside servers or vehicle-mounted processors), thereby reducing inference latency and alleviating network backhaul congestion [9].

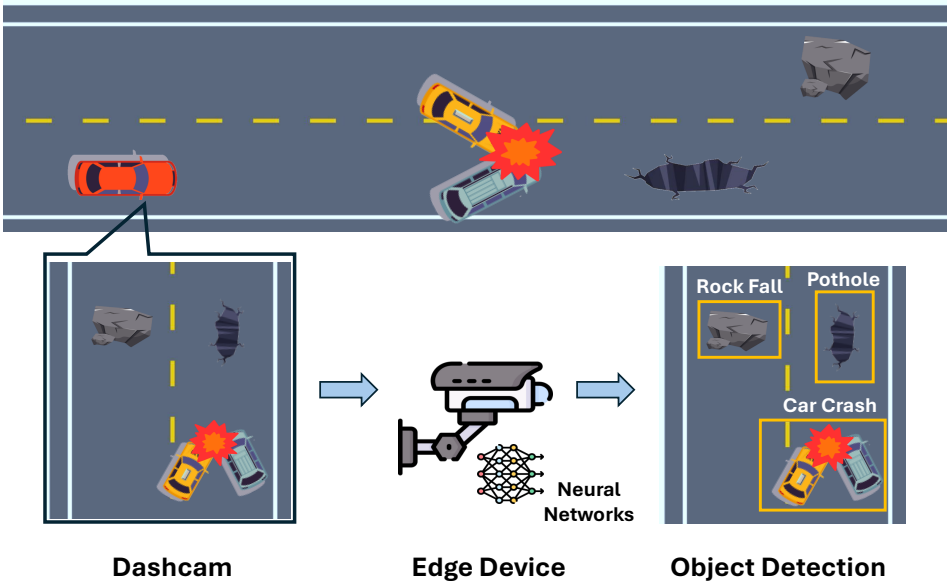

Figure 1. Illustration of edge-based object detection in a smart city scenario. Video frames are first captured by dashcams and transmitted to nearby edge devices, where neural network models perform real-time inference. Detected objects such as rock falls, potholes, and car crashes are identified and localized, enabling timely alerts for traffic safety and urban management.

Despite the advantages of proximity-based deployment, offloading object detection to edge devices presents a set of substantial challenges. Edge environments are inherently heterogeneous, with diverse hardware capabilities (e.g., CPU, GPU, and memory resources), energy constraints (e.g., limited battery supply for mobile or roadside units), and varying network conditions [10]. In addition, workloads fluctuate with the complexity of the scene and object density, creating a dynamic and resource-constrained environment in which static scheduling strategies struggle to maintain both accuracy and low-latency responsiveness [11]. Deep reinforcement learning (DRL) has emerged as a promising approach to address the complexities of edge-based object detection scheduling [12–14]. Unlike static heuristic methods, DRL learns adaptive policies through trial-and-error interactions with the environment, thereby enabling it to navigate complex trade-offs among inference accuracy, latency, and resource consumption under continuously fluctuating workloads and network conditions [15].

However, the application of DRL is constrained by its reliance on manually engineered reward functions. The reward function encodes optimization objectives (e.g., inference accuracy, response latency, and energy consumption) into a scalar feedback signal that guides policy learning [16]. In real-world object detection scenarios, optimization objectives change according to the operational context. One example is autonomous driving, where low latency is prioritized during peak traffic hours while accuracy is maximized during

off-peak periods, such as at night [17–19]. Each change in service objectives typically requires redesigning and fine-tuning the reward function, a process that is time-consuming, error-prone, and difficult to scale in dynamic smart city environments [20].

Recently, large language models (LLMs) have demonstrated outstanding planning capabilities in tasks such as adaptive resource scheduling [21,22] and multi-agent decision coordination [23,24]. Emerging studies further reveal that LLMs can generate reward function codes from natural language prompts, thereby automating reinforcement learning (RL) for complex control tasks [25]. Although the use of LLMs in smart city applications is gaining increasing attention [26,27], their potential to dynamically adapt offloading decisions through language prompts remains largely unexplored.

Leveraging the contextual reasoning and code generation capabilities of LLMs, our system introduces a new paradigm that enables DRL schedulers in edge-based object detection to adapt seamlessly to evolving optimization objectives. Unlike conventional DRL approaches, which require extensive manual reward engineering whenever service priorities change, our framework directly incorporates the transformed optimization objectives into natural language prompts, enabling the LLM to automatically redesign the reward function.

In detail, our method (i) interprets optimization objectives expressed in natural language, (ii) translates them into an executable, environment-compatible reward function code, and (iii) iteratively refines the reward through closed-loop simulation feedback. This design eliminates the need for manual reward engineering when service objectives change, allowing DRL policies to adapt rapidly to evolving operational contexts. To evaluate our method, we conducted simulations on real-world datasets, comparing DRL scheduling policies trained with LLM-generated reward functions against those trained with expert-designed rewards. The experiment results demonstrate that DRL policies trained with LLM-generated reward functions consistently achieve superior scheduling performance compared to those designed by human experts, including higher user allocation rates and lower latency. Moreover, the method exhibits robustness to changes in optimization priorities, retaining high performance when changing objectives (from maximizing throughput to minimizing latency) through simple re-prompting of the LLM without any manual reward redesign.

The main contributions of this work are summarized as follows:

- **LLM-driven DRL retargeting framework:** We present a novel framework that leverages LLMs to directly retarget DRL scheduling policies for edge-based object detection from natural language instructions, removing the need for manual reward engineering.
- **Automatic reward generation and refinement:** The proposed approach converts optimization objectives into an executable, environment-compliant reward function code and iteratively improves it using simulation feedback.
- **Adaptability to dynamic optimization goals:** Our method supports rapid adaptation of DRL policies to evolving service requirements, sustaining high performance under shifting priorities in latency, accuracy, and energy consumption.
- **Comprehensive experimental validation:** Extensive simulations on real-world datasets verify that LLM-generated rewards enable DRL schedulers to outperform human expert-crafted rewards, even when optimization objectives change dynamically.

The rest of this paper is organized as follows. Section 2 reviews related work on edge-based object detection scheduling, deep reinforcement learning, and LLM-assisted optimization. Section 3 presents our problem formulation, the preference-conditioned MDP model, and the proposed LLM-driven DRL scheduling framework. Experimental results and comparative evaluations are provided in Section 4. Finally, Section 5 concludes the work and discusses potential directions for future research.

2. Literature Review

This section reviews related work from four perspectives: (i) edge-based object detection in smart cities, (ii) intelligent offloading and scheduling strategies for edge object detection, and (iii) large language models for decision making.

2.1. Edge-Based Object Detection in Smart Cities

Object detection has emerged as a core enabling technology for the perception layer of smart cities, underpinning a wide range of IoT-driven applications. In the context of intelligent urban surveillance, for example, automated object detection facilitates critical tasks such as vehicle recognition, license plate identification, and traffic flow monitoring, which are essential for real-time road safety, restricted-area security, and vehicle tracking [28]. Beyond transportation, object detection also plays a pivotal role in environmental sustainability efforts. Deep learning-based small object detection models have been employed for intelligent waste management, enabling accurate detection and classification of garbage items to optimize recycling processes in rapidly urbanizing cities [29]. Similarly, in the domain of residential safety and automation, IoT-enabled object detection has been integrated with home monitoring and appliance control systems, enhancing both security and convenience through real-time recognition and remote actuation [30].

Early smart city object detection systems were primarily deployed in centralized cloud infrastructures, transmitting video and sensor data to remote servers for processing. Although this enabled access to ample computational resources, the rise of latency-sensitive applications (e.g., autonomous driving, real-time surveillance, and large-scale video analytics) has revealed inherent drawbacks, including high latency, heavy backhaul bandwidth usage, and reduced reliability under unstable networks. As a result, increasing research attention has shifted toward migrating inference closer to data sources via edge computing.

For example, ref. [9] proposed Edge YOLO, an object detection system based on edge–cloud collaboration and reconstructive convolutional neural networks. This lightweight framework combines pruning in the feature extraction network and compression in the feature fusion network, enabling real-time detection on embedded platforms with 26.6 frames per second, only 25.67 megabytes of parameters, and a mean average precision of 47.3% on COCO2017. Ref. [31] introduced $EC^2Detect$, a real-time online video object detection method based on edge–cloud collaboration. By performing accurate but computationally heavy detection in the cloud on adaptively selected keyframes and using lightweight tracking at the edge for other frames, $EC^2Detect$ achieves a more than 4.77× higher processing speed, up to 8.12× lower latency, and approximately 17× lower bandwidth usage compared to state-of-the-art methods. Similarly, edge–cloud collaborative video analytics for smart cities have been shown to deliver over 80% greater efficiency compared to cloud-only approaches while maintaining an object tracking accuracy exceeding 96% [32].

While edge deployment mitigates the latency and bandwidth issues of cloud-based object detection, it faces inherent challenges such as device heterogeneity, limited resources, unstable networks, and fluctuating workloads, making offloading and scheduling challenging.

2.2. Intelligent Offloading and Scheduling Strategies for Edge Object Detection

The heterogeneity of edge environments renders traditional static or heuristic scheduling algorithms insufficient for sustaining high-performance object detection. DRL offers a promising alternative by enabling agents to learn adaptive scheduling policies through continuous interaction with the environment.

For example, ref. [14] proposed a DRL-assisted task offloading and resource allocation approach for self-driving object detection, integrating a task prioritization mechanism

and a time utility function to optimize efficiency. The method, formulated as an NP-hard MINLP problem, employed a deep neural network to learn offloading strategies based on traffic and network conditions, achieving at least a 10% higher utility time compared to representative baselines. Similarly, ref. [33] developed a DRL-based intelligent driving task scheduling service for vehicle–edge collaborative networks, leveraging real-time factors such as channel conditions, image entropy, and detector confidence. By combining lightweight and heavyweight models within a collaborative framework, the system achieved stable convergence, enhanced detection precision, and efficient scheduling in complex driving scenarios on the SODA10M dataset.

In a broader MEC context, ref. [34] presented a DRL-based user allocation framework that predicts the resource utilization of incoming service requests, particularly in CPU–GPU co-execution environments where resource usage is significantly non-linear. The proposed approach estimates the number of users an edge server can accommodate under given latency constraints and determines an optimal allocation policy, outperforming deterministic methods by at least 10% in terms of user allocation efficiency. However, DRL also has a key limitation: it relies on manually engineered reward functions that must be redesigned whenever optimization objectives change.

Beyond DRL-based approaches, recent studies have explored complementary directions for enhancing MEC scheduling. For instance, ref. [35] proposed an energy-efficient task scheduling framework based on traffic mapping in heterogeneous MEC environments, which minimizes makespan under energy constraints and achieves substantial improvements in both energy consumption and task throughput. In parallel, ref. [36] introduced a fuzzy logic-guided behavior cloning method to accelerate the early-stage training of DRL agents for edge offloading, reducing random exploration and improving stability. These works highlight important advances in MEC task scheduling; however, DRL also has a key limitation. It relies on manually engineered reward functions that must be redesigned whenever optimization objectives change.

2.3. Large Language Models for Decision Making

The emergence of ChatGPT (OpenAI, GPT-3.5 version) has sparked a surge of research into LLMs, which are now being applied across diverse domains due to their remarkable generalization, contextual reasoning, and code generation capabilities.

In particular, LLMs have shown outstanding planning performance in adaptive resource scheduling and multi-agent decision coordination tasks. For instance, ref. [21] proposed the LLM-based resource allocation optimizer (LLM-RAO) to address complex and dynamic wireless network optimization problems. By using prompt-based tuning to flexibly convey changing task objectives and constraints, LLM-RAO achieved up to 40% higher performance than conventional deep learning methods and 2.9× the performance of traditional analytical approaches, without requiring retraining. Similarly, ref. [22] introduced NetLLM, a unified framework that adapts LLMs for networking tasks such as viewport prediction, adaptive bitrate streaming, and cluster job scheduling. Leveraging pre-trained knowledge and multimodal data processing, NetLLM significantly outperformed state-of-the-art algorithms while reducing the engineering overhead of domain-specific model design.

Recent studies also demonstrate that LLMs can serve as automated reward designers for RL, enabling dynamic policy adaptation in complex control environments. Ref. [25] proposed Eureka, which takes advantage of the zero-shot code generation capabilities of LLMs and the opportunities for improvement in this context to evolve reward functions without task-specific templates. Tested across 29 RL environments, Eureka-generated rewards outperformed human expert designs in 83% of tasks, with an average normalized

improvement of 52%. Likewise, ref. [24] developed the Decision Language Model (DLM) for dynamic restless multi-armed bandit problems in public health resource allocation. DLM interprets human policy preference prompts, generates reward functions as code, and iteratively refines them using simulation feedback, enabling policy outcomes to be dynamically shaped through natural language alone.

While LLMs have shown potential in resource allocation and control, their use for dynamically generating and refining rewards in real-time, multi-objective scheduling, especially in the context of smart city object detection, has not yet been explored. This research gap motivates our proposed LLM-driven framework for adaptive, goal-aware edge scheduling.

3. Materials and Methods

3.1. Task Offloading Actions in Edge-Based Object Detection

We consider a multi-node edge computing environment for real-time object detection, where distributed edge nodes collaboratively process incoming inference requests from various video streams. The scheduler must decide, at each decision epoch, how to offload these requests across available edge nodes to balance performance metrics such as latency, accuracy, and energy consumption.

Let $\mathcal{N} = \{1, \ldots, M\}$ denote the set of edge nodes jointly serving inference requests. Time is slotted as $t \in \{0, 1, 2, \ldots\}$. At the beginning of slot t, a batch of new requests $\mathcal{D}_t$ arrives (or is released from queues). Each node $j \in \mathcal{N}$ exposes an observable operational state vector

$$\Sigma_{j,t} = \left(\mathrm{RAM}_{j,t},\ \mathrm{workload}_{j,t},\ \mathrm{queue}_{j,t},\ \mathrm{GPU}_{j,t},\ \mathrm{power}_{j,t},\ \mathrm{channel}_{j,t}\right), \tag{1}$$

which captures the following: (i) available memory ($\mathrm{RAM}_{j,t}$), (ii) CPU workload ($\mathrm{workload}_{j,t}$), (iii) queue length ($\mathrm{queue}_{j,t}$), (iv) fine-grained GPU utilization ($\mathrm{GPU}_{j,t}$), (v) device-level power cap or consumption budget ($\mathrm{power}_{j,t}$), and (vi) wireless channel quality indicators ($\mathrm{channel}_{j,t}$).

Together, these features reflect the heterogeneous and resource-constrained characteristics of practical edge deployments. The global system state is then defined as $\Sigma_t = \{\Sigma_{j,t}\}_{j \in \mathcal{N}}$, augmented with request batch statistics $\mathcal{D}_t$ (e.g., request types and input sizes). This unified formulation enables the scheduler to account for both node-level resource availability and network dynamics when making offloading decisions.

At each slot, the offloading decision is modeled as a binary assignment matrix

$$X_t = \left[x_{i,j,t}\right]_{i \in \mathcal{D}_t,\ j \in \mathcal{N}}, \quad x_{i,j,t} \in \{0, 1\}, \tag{2}$$

where $x_{i,j,t} = 1$ indicates that request i is assigned to node j. Each request can be served by at most one node, and each node has a finite capacity $C_{j,t}$ in slot t:

$$\sum_{j \in \mathcal{N}} x_{i,j,t} \leq 1, \qquad \forall i \in \mathcal{D}_t, \tag{3}$$

$$\sum_{i \in \mathcal{D}_t} x_{i,j,t} \leq C_{j,t}, \qquad \forall j \in \mathcal{N}, \tag{4}$$

The set of feasible actions $\mathcal{A}(\Sigma_t, \mathcal{D}_t)$ is determined by constraints (3) and (4). After action X_t is applied, queues are updated and new arrivals are incorporated, generating the next state via a controlled transition kernel $P(\Sigma_{t+1} \mid \Sigma_t, X_t)$.

3.2. Preference-Driven Utility Modeling

We consider an index set of optimization objectives $\mathcal{O}$, which may include throughput (S), latency (L), energy consumption (E), and detection accuracy (A). For each $o \in \mathcal{O}$, a data-driven measurement operator returns the realized metric under assignment X_t:

$$m_{o,t} = \mathcal{M}_o\big(\Sigma_t,\ X_t;\ \mathcal{D}_t\big), \tag{5}$$

where $\mathcal{D}_t$ contains contemporaneous logs or features from the replay dataset. No parametric performance model is assumed, enabling direct use of empirical observations or simulated measurements. Each objective has a sign $s_o \in \{+1, -1\}$ indicating whether larger or smaller values are preferable, with the following as an example:

$$m_{S,t} = \sum_{i \in \mathcal{D}_t} \sum_{j \in \mathcal{N}} x_{i,j,t}, \qquad s_S = +1. \tag{6}$$

Optimization objectives are expressed by a preference descriptor $\mathbf{z}_t \in \mathbb{R}^{|\mathcal{O}|}$ provided by upper-layer controllers or user prompts. To preserve the Markov property, we augment the state with either the preference vector or a version identifier:

$$\tilde{\Sigma}_t = (\Sigma_t, \mathbf{z}_t) \quad \text{or} \quad \tilde{\Sigma}_t = (\Sigma_t, v_t), \tag{7}$$

where v_t denotes the identifier of the active preference profile.

The scalar utility used for DRL training is computed by a preference-aware aggregator:

$$U_t = \mathcal{U}_{\psi_t}\Big(\{(m_{o,t}, s_o)\}_{o \in \mathcal{O}},\ \mathbf{z}_t\Big), \tag{8}$$

where ψ_t parameterizes the aggregation rule. In our framework, the implementation of $\mathcal{U}_{\psi_t}$ is automatically generated by a large language model (LLM) from the given preferences $\mathbf{z}_t$, yielding an executable code that satisfies the following: (i) boundedness, $U_{\min} \le U_t \le U_{\max}$; (ii) directional consistency, $s_o\, \partial U_t / \partial m_{o,t} \ge 0$ with other metrics fixed; (iii) responsiveness to preference changes, $s_o\, \partial U_t / \partial z_{o,t} \ge 0$; and (iv) evaluability in the following form:

$$U_t = \tilde{\mathcal{U}}(\Sigma_t, X_t, \mathcal{D}_t, \mathbf{z}_t; \psi_t).$$

Operational constraints are expressed in a preference-adaptive form:

$$\mathcal{C}_t = \big\{(k, \bowtie_k, b_{k,t}) \ :\ k \in \mathcal{K}_t \subseteq \mathcal{K}\big\}, \qquad b_{k,t} = \mathcal{B}_k(\mathbf{z}_t), \tag{9}$$

where $m_{k,t} = \mathcal{M}_k(\Sigma_t, X_t; \mathcal{D}_t)$ is the measured KPI and $\bowtie_k \in \{\le, \ge\}$ indicates the constraint direction. The slot-wise violation is

$$g_{k,t} = \begin{cases} \big[m_{k,t} - b_{k,t}\big]_+, & \text{if } m_{k,t} \le b_{k,t} \text{ is required,} \\ \big[b_{k,t} - m_{k,t}\big]_+, & \text{if } m_{k,t} \ge b_{k,t} \text{ is required,} \end{cases} \qquad [x]_+ = \max\{x, 0\}. \tag{10}$$

Let $\mathbf{g}_t = \{g_{k,t}\}_{k \in \mathcal{K}_t}$ and define a hard-feasibility indicator $\chi_t = \mathbb{1}\{\|\mathbf{g}_t\|_\infty = 0\}$. The capacity constraint (4) is enforced as a hard limit; other KPIs can be imposed strictly via χ_t or incorporated softly through the utility design in (8).

3.3. MDP Formulation and DQN-Based Learning Framework

In our formulation, scheduling actions are taken at each decision epoch in a slot-based manner, which is standard in edge computing environments where requests arrive in batches over time. Although the policy operates on a per-slot basis, the agent is trained to

maximize the expected discounted cumulative reward over an infinite horizon. This enables the learned policy to implicitly account for long-term trade-offs across multiple slots, rather than behaving as a purely myopic heuristic that only optimizes the immediate reward.

Therefore, the adaptive offloading and scheduling problem is formulated as a Markov Decision Process (MDP)

$$\langle \tilde{\mathcal{S}}, \mathcal{A}, P, U, \beta \rangle,$$

where $\tilde{\mathcal{S}}$ is the *augmented state space* that includes both the operational state Σ_t and the active preference descriptor $\mathbf{z}_t$ from (7); $\mathcal{A}$ is the feasible action set $\mathcal{A}(\Sigma_t, \mathcal{D}_t)$ defined by (3) and (4); P is the transition kernel determined by queue dynamics and new arrivals; U is the utility function in (8); and $\beta \in (0, 1)$ is the discount factor.

The objective is to find a policy $\pi(X \mid \tilde{\Sigma})$ that maximizes the expected discounted return:

$$\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \beta^t U_t \right]. \tag{11}$$

We adopted a *value-based* approach using Deep Q-Networks (DQN) to approximate the action-value function $Q(\tilde{\Sigma}, X)$. At each training step, the temporal-difference (TD) update is as follows:

$$Q(\tilde{\Sigma}_t, X_t) \leftarrow Q(\tilde{\Sigma}_t, X_t) + \alpha \Big(U_t + \beta \max_{X' \in \mathcal{A}} Q(\tilde{\Sigma}_{t+1}, X') - Q(\tilde{\Sigma}_t, X_t) \Big), \tag{12}$$

where $\alpha > 0$ is the learning rate.

Action masking is applied when $\chi_t = 0$ is used to guarantee feasibility. Alternatively, infeasible actions can be penalized directly through LLM-generated utility U_t, which allows the DRL agent to learn trade-offs between hard and soft constraints in preference-adaptive settings.

3.4. LLM-Driven Reward Function Integration and Policy Training

The overall procedure of our method is summarized in Algorithm 1, which outlines the LLM-guided reward design and DQN-based policy training process for edge offloading. Our method leverages LLMs to automatically generate reward functions for DRL in edge-based object detection scheduling. This enables the scheduling policy to adapt to changing optimization goals without manual reward engineering. As illustrated in Figure 2, the process operates in three integrated stages.

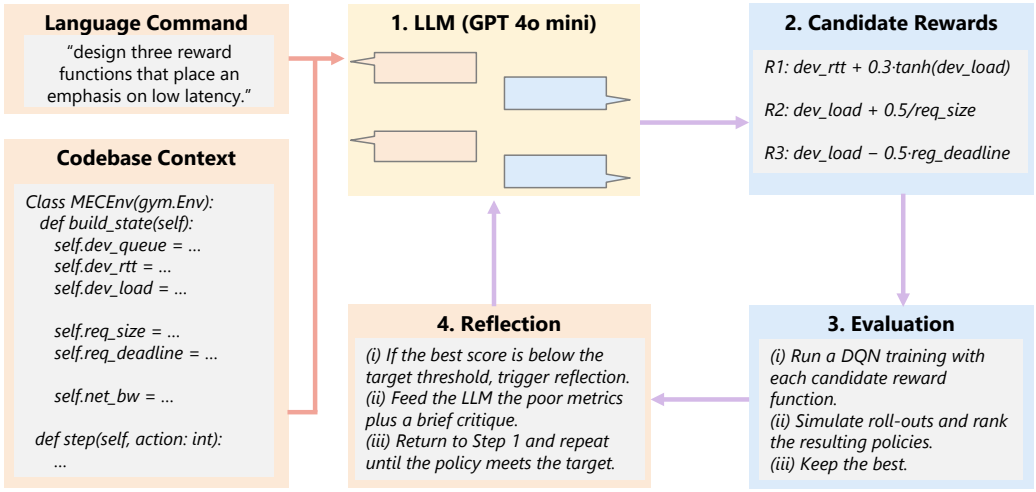

Figure 2. Overall workflow of the proposed LLM-assisted DRL framework for edge-based object detection offloading.

Algorithm 1: LLM-Guided Reward Design and DQN Training for Edge Offloading

Input : Instruction text ℓ; preference vector $\mathbf{z}$; context ζ (API, state, metrics, constraints)
Output : Best reward function $R^\star$; trained policy $\pi^\star$
Hyperparams : Max iterations I; candidates per round K; training epochs n_e; steps per epoch n_s; evaluation episodes N_{eval}; threshold τ

for *iteration* = 1 **to** I **do**
  `/* Step 1: Generate candidate reward functions */`
  candidate_rewards ← LLM_Generate(ℓ, $\mathbf{z}$, ζ, K)
  reflection_log ← $\emptyset$
  `/* Step 2: Train and evaluate each candidate */`
  for *each reward_fn in candidate_rewards* **do**
    Initialize DQN: Q_θ, target $\hat{Q}$, replay buffer $\mathcal{D}$
    for *epoch* = 1 **to** n_e **do**
      Reset environment, observe initial state
      for *step* = 1 **to** n_s **do**
        Mask invalid actions; choose action via ϵ-Greedy on Q_θ
        Execute action; observe next state, metrics, env data
        reward ← reward_fn(metrics, $\mathbf{z}$)
        Store transition in $\mathcal{D}$; update Q_θ using TD target
        Periodically update $\hat{Q}$ from Q_θ
    Evaluate over N_{eval} episodes → metrics, utility
    Append (reward_fn, metrics, utility) to reflection_log
  `/* Step 3: Select best candidate */`
  best ← candidate with highest score
  if *score(best)* $\geq \tau$ **then**
    return reward_fn(best), policy(best)
  `/* Step 4: Reflection and context update */`
  Update ζ with best candidate and results
  $\ell \leftarrow$ LLM_Reflect(ℓ, reflection_log)
return reward_fn(best), policy(best)

In the first stage, a natural language prompt is constructed to instruct the LLM to produce a reward function tailored to the current optimization preferences $\mathbf{z}_t$. The prompt includes (i) the optimization objectives (e.g., minimizing latency, maximizing accuracy, and reducing energy consumption); (ii) a description of the system model, observable state variables Σ_t, and performance metrics $\mathcal{O}$; and (iii) design requirements such as boundedness, directional consistency, and evaluability. An example of such a prompt is provided in Appendix A. Using this information, the LLM generates an executable Python code by implementing the utility function $\mathcal{U}_{\psi_t}$ in Equation (8) and mapping measured metrics $\{m_{o,t}\}$ and preferences $\mathbf{z}_t$ to a scalar utility U_t. It is worth noting that in the initial stage of our experiments, the LLM occasionally produced invalid functions (e.g., containing extra code or mismatched interfaces). To address this, we progressively refined the prompting strategy by introducing explicit RULE constraints, which enforced correct function signatures, bounded outputs, and compatibility within the simulation environment.

In the second stage, the validated reward function was embedded into the DQN training loop described in Section 3.3. The agent interacts with the simulated multi-node edge environment, observes the augmented state $\tilde{\Sigma}_t$, selects scheduling actions X_t, and

receives rewards U_t computed by the generated function. Q-network parameters are updated using temporal-difference learning (Equation (12)), with experience replay and ϵ-Greedy exploration ensuring stable and efficient policy learning.

Finally, the trained policy is evaluated in simulation under diverse workload and network conditions. Performance metrics $\{m_{o,t}\}$ and utilities U_t are recorded to assess whether the policy meets the intended objectives. If deficiencies are detected—such as latency spikes or suboptimal energy usage—the evaluation feedback is incorporated into a revised prompt. An example of such an iterative refinement prompt is provided in Appendix B. The LLM then generates an improved reward function, which is validated and used for retraining, forming an iterative optimization cycle.

4. Results

In our experiments, we employed GPT-4o-mini [37] as the primary large language model for generating reward functions, and additionally included Llama-3.1-8B-Instruct [38] as a lightweight, locally deployable model to assess the feasibility of on-device reward generation. GPT-4o-mini was selected because it provides strong code-generation and reasoning capabilities at a relatively low monetary cost, making it an efficient choice for iterative reward function generation. In contrast, Llama-3.1-8B-Instruct was chosen as an open-source alternative that can be fine-tuned and executed locally on edge servers, thereby eliminating reliance on proprietary APIs and enabling deployment in resource-constrained environments. Including both models allowed us to evaluate the trade-off between performance and deployability in LLM-based reward design. The EUA dataset [39] was used to model the spatial distribution of edge servers, containing the locations of base stations and users within the Melbourne Central Business District. For object detection workloads, we adopted the dataset provided by [34], which characterizes service execution latency under varying system configurations. Specifically, the dataset includes measurements obtained by running YOLO [40] and MobileNetV2 [41] while adjusting memory allocation, the number of CPU cores, workload intensity, GPU utilization, and the number of service requests. All simulations were conducted on a workstation equipped with an Intel(R) Core i7-13700K CPU, 64 GB RAM, and an NVIDIA RTX 4090 GPU. The DRL agents were implemented in Python 3.8.13 using the Stable-Baselines3 library, and integer linear programming (ILP) baselines were solved using the Python Mixed Integer Programming (MIP) library. For performance comparison, we evaluated our proposed LLM-guided DRL scheduling strategy against the RL-based approach from [34], as well as three additional baselines: ILP, a Greedy scheduling algorithm, and the Heterogeneous Earliest Finish Time (HEFT) heuristic [42].

4.1. Overhead of LLM-Based Reward Function Generation

To evaluate the practical feasibility of our framework, we measured the computational and monetary overhead of reward function generation using both GPT-4o-mini and Llama-3.1-8B-Instruct. All experiments were conducted under a latency threshold of 50 s and 70 servers, representing a high-load edge computing scenario.

Each reward design cycle consisted of two iterations. In the first iteration, we prompted the LLM to generate $K = 10$ candidate reward functions, which were then evaluated in the simulation. Based on the evaluation feedback, a second iteration was performed, prompting the LLM to generate other $K = 10$ refined candidates. This two-iteration process was repeated five times to collect statistics on token usage, API latency, and monetary cost. For each query, we recorded token usage, latency, and cost (when applicable), and reported the mean and standard deviation across repeated runs to quantify stability.

Table 1 summarizes the measured statistics for both GPT-4o-mini and Llama-3.1-8B-Instruct. For GPT-4o-mini, the first iteration required on average 3577 ± 27 tokens per query,

with an API latency of 59.6 ± 6.1 s and a monetary cost of USD 0.00184 ± 0.00002, achieving a performance score of 610 ± 14. In the second iteration, token usage increased to 4535 ± 72 on average due to more complex refinement prompts, while latency decreased slightly to 57.4 ± 5.8 s. Performance improved to 641 ± 9, confirming that refinement enhances both the effectiveness and stability of generated reward functions.

Table 1. Summary of LLM-based reward function generation statistics.

	Token Usage	Latency (s)	Cost (USD)	Performance
GPT Original	3577 ± 27	59.6 ± 6.1	0.00184 ± 0.00002	610 ± 14
Llama Original	3776 ± 98	75.4 ± 7.5	0	602 ± 10
GPT Iteration	4535 ± 72	57.4 ± 5.8	0.00200 ± 0.00004	641 ± 9
Llama Iteration	4586 ± 57	76.7 ± 5.8	0	599 ± 14

For Llama-3.1-8B-Instruct, which runs locally and incurs no monetary cost, the first iteration consumed 3776 ± 98 tokens on average, with a latency of 75.4 ± 7.5 s and a performance score of 602 ± 10. In the second iteration, token usage rose to 4586 ± 57, latency increased modestly to 76.7 ± 5.8 s, and performance slightly decreased to 599 ± 14. These results highlight the trade-off between local deployment—which eliminates API costs—and increased latency compared to cloud-based GPT inference.

The results demonstrate that LLM-based reward function generation is both computationally feasible and economically lightweight. GPT-4o-mini achieves lower latency and more consistent refinement quality, while Llama-3.1-8B-Instruct offers a viable local alternative that removes external API dependency at the cost of slightly higher latency and less stable refinement outcomes.

The observed differences stem from both inference and prompt-level factors. Llama-3.1-8B-Instruct was executed locally on a single RTX 4090 GPU, whereas GPT-4o-mini was accessed through a cloud-hosted, high-throughput endpoint. The limited computational bandwidth of local inference leads to higher wall-clock latency. Moreover, due to its smaller parameter capacity and weaker alignment for structured code generation, Llama required a more verbose prompt with additional `RULE` constraints to ensure syntactic correctness and environment compatibility, as detailed in Appendix C. This longer prompt and output sequence explain the higher token usage observed. Overall, while GPT-4o-mini remains preferable for efficient cloud-based prototyping, Llama-3.1-8B-Instruct demonstrates the practicality of on-device reward generation for privacy-sensitive or offline edge deployments.

4.2. Performance Evaluation Under Varying Server and User Scales

The evaluation considers latency thresholds of 30 s and 50 s, which were chosen to represent two practical operating regimes in smart city edge computing scenarios. A threshold of 30 s corresponds to latency-sensitive services (e.g., traffic monitoring or safety-critical alerts), where tight response times are required. In contrast, a 50 s threshold reflects more latency-tolerant services (e.g., batch analytics or non-critical monitoring), where higher delays are acceptable. This dual setting enables us to examine the adaptability of scheduling strategies under both stringent and relaxed timing requirements. The number of servers varied between 50, 60, and 70 to represent small-to-medium edge deployments typical in urban testbeds, while the number of users ranged from 100 to 700 to simulate a growing workload intensity. Together, these configurations span both resource-constrained and resource-abundant regimes, providing a comprehensive view of system behavior.

Figure 3 presents the results for a 30 s latency threshold. Across all server configurations, the proposed LLM-based RL consistently achieved the highest number of scheduled users, outperforming Original RL, ILP, HEFT, and Greedy baselines. When servers are lim-

ited (e.g., 50 servers), all algorithms exhibit constrained performance due to capacity limits. In this case, the LLM-based RL provides noticeable gains for medium-to-high user loads (400–700 users). For instance, with 50 servers and 700 users, the LLM-based RL schedules 580 users, compared to 567 for Original RL, 348 for ILP, and only 291 for Greedy. As server availability increases to 60 and 70, the performance of all methods improves; however, the LLM-based RL maintains its lead, scheduling 626 users with 60 servers and 700 users, versus 606 for Original RL and 411 for ILP. With 70 servers and 700 users, the LLM-based RL achieves 632 scheduled users, nearly saturating system capacity, while ILP and HEFT remain below 460. Compared to HEFT, which represents a modern heuristic baseline, the LLM-based RL demonstrates clear improvements of 5–10% under high load, confirming that automatically generated reward functions enhance long-term scheduling effectiveness.

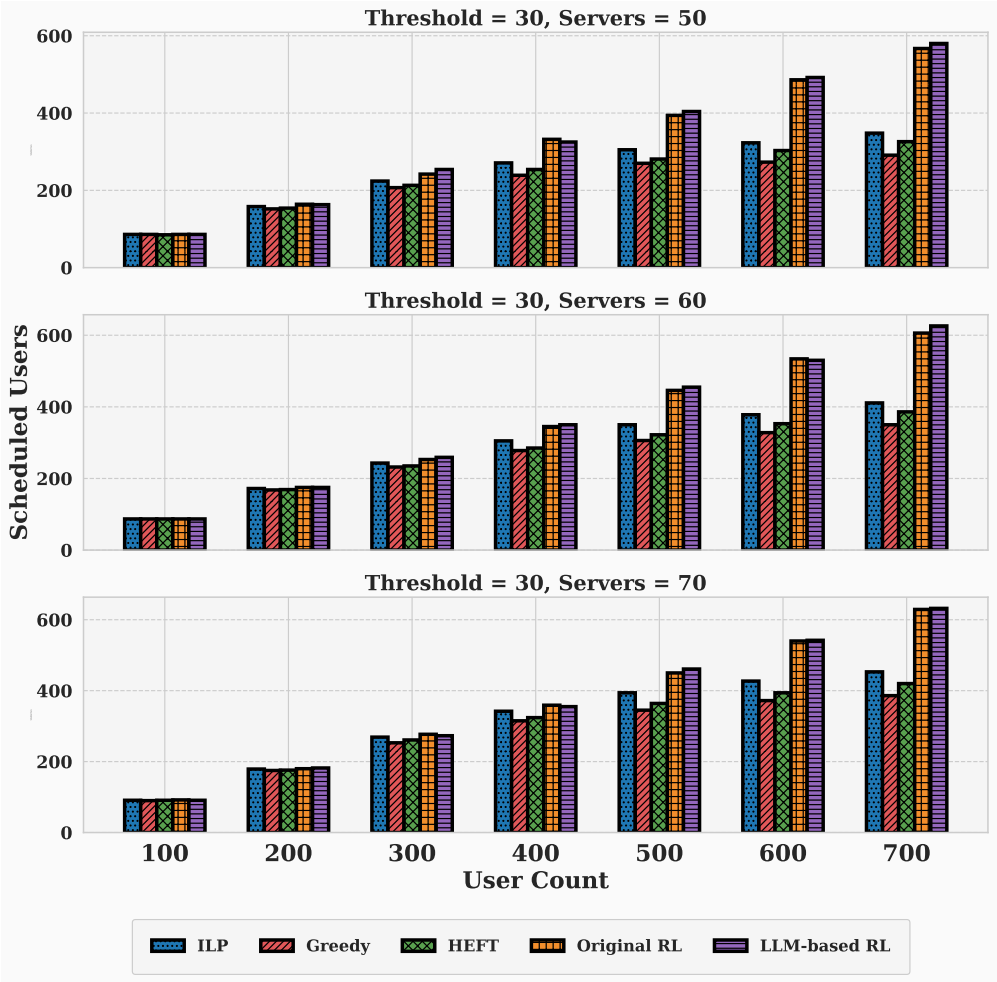

Figure 3. Scheduled users vs. user count across algorithms and server configurations (threshold = 30).

Figure 4 reports the results for a 50s latency threshold. As expected, all methods benefit from the increased temporal slack, achieving higher user allocations across all server and user load combinations. Nevertheless, the LLM-based RL maintains a consistent advantage, particularly under medium-to-high load conditions. For example, with 70 servers and 700 users, the LLM-based RL successfully schedules 646 users, compared to 620 for Original RL, 580 for ILP, and 529 for Greedy. Even in resource-abundant settings where performance differences diminish at low loads (100–200 users), the LLM-based RL continues to outperform other methods at scale, effectively leveraging the additional latency budget to balance throughput and constraint satisfaction.

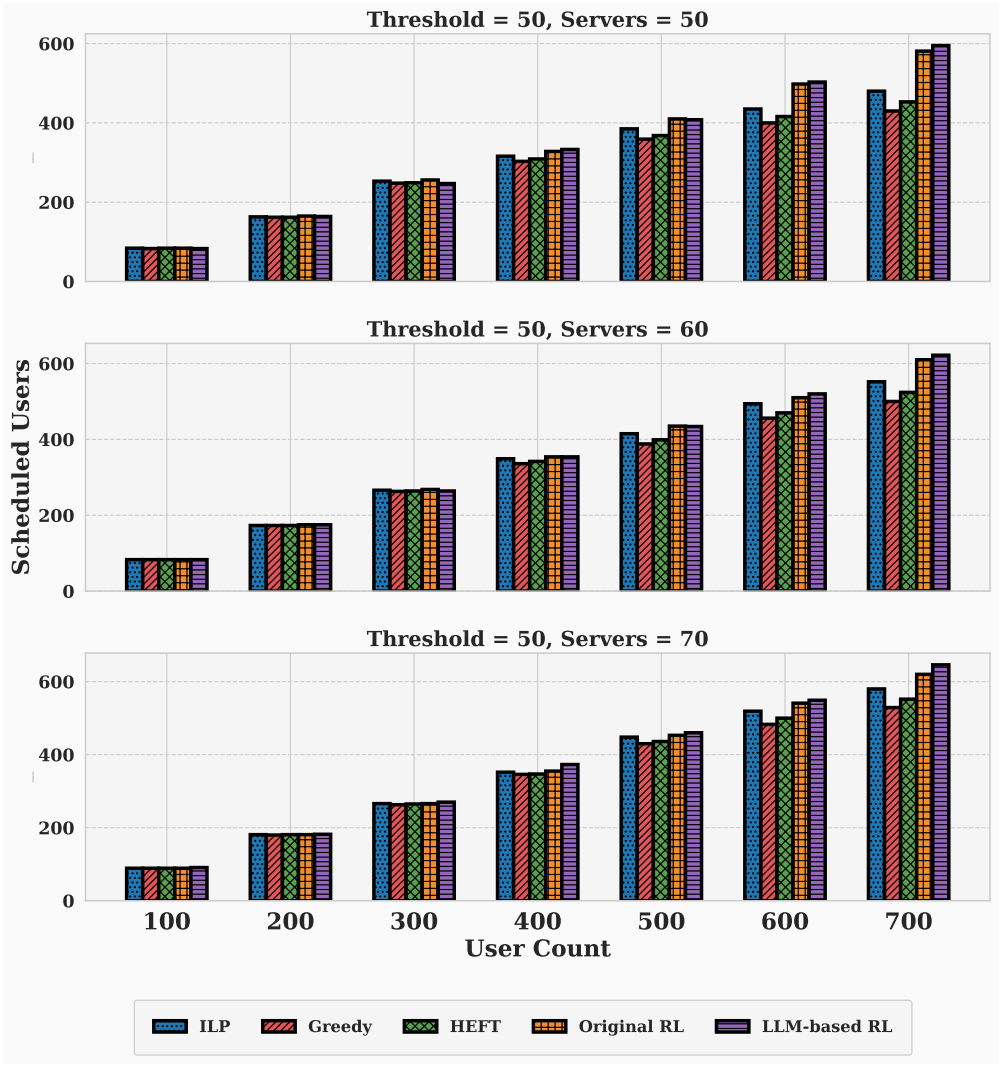

Figure 4. Scheduled users vs. user count across algorithms and server configurations (threshold = 50).

Overall, the results across both latency thresholds demonstrate that the proposed LLM-based RL framework maintains superior performance in nearly all scenarios. Its adaptability to varying resource capacities and latency requirements allows it to consistently outperform heuristic baselines (Greedy and HEFT), optimization-based methods (ILP), and conventional RL approaches. On average, LLM-based RL achieves 5–15% higher user allocations in medium-to-high load scenarios, confirming the effectiveness of LLM-driven reward generation in dynamic edge computing environments.

4.3. Latency-Minimization Scheduling with Prompt-Driven Objective Changing

To further evaluate the adaptability of our LLM-based framework, we shifted the optimization objective from solely maximizing the number of allocated users to minimizing the average latency while maintaining comparable allocation counts. This change is implemented by modifying the natural language prompt provided to the LLM, which, in turn, generates a new reward function reflecting the updated priorities.

Figure 5 shows that the LLM-based policy yields a consistently lower average latency than the RL baseline across all user volumes. From visual readings of the plot, the latency gap is about 0.6 s at 100 users (≈30.4 vs. ≈31.0 s), widens to roughly 1.6 s at 200 users (≈31.5 vs. ≈33.1 s), and remains around 1.8 s at 300 users (≈32.6 vs. ≈34.4 s). At 400 users, the advantage reaches about 2.4 s (≈32.5 vs. ≈34.9 s). Notably, the LLM-based curve flattens beyond 300 users, indicating better adaptation under heavier loads, whereas the

RL baseline continues to trend upward. These results confirm that, after switching the optimization objective via a prompt (from maximizing allocations to minimizing latency while preserving allocations), the LLM-generated reward enables the scheduler to reduce latency substantially without sacrificing allocation levels.

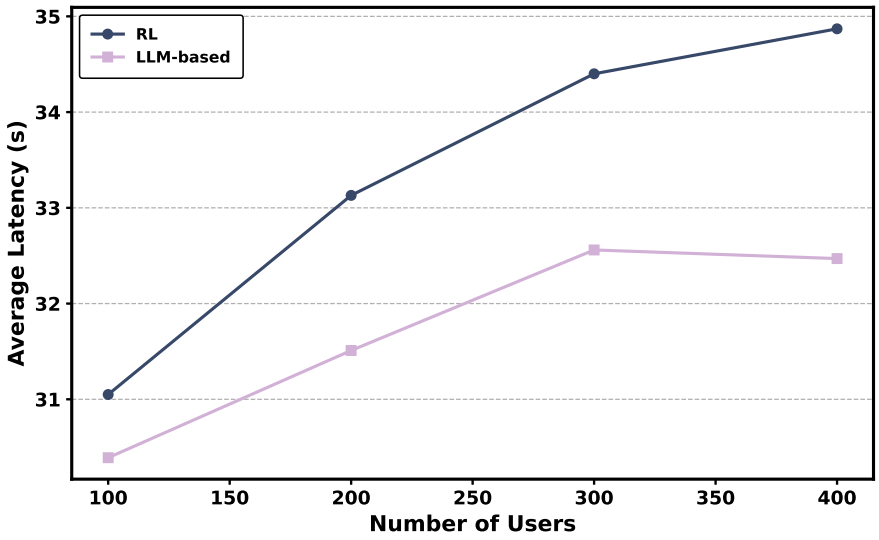

Figure 5. Average latency under the latency-minimization objective while maintaining comparable allocation counts.

5. Conclusions and Future Work

In this work, we present an adaptive scheduling framework for edge-based object detection that integrates large language models into deep reinforcement learning to automatically generate reward functions. By treating optimization goals as prompts, our approach enables rapid adaptation of scheduling policies to changing objectives without manual reward engineering. We modeled the scheduling process as a multi-node Markov Decision Process and employed a DQN-based solution.

Our simulations on real-world datasets suggest that LLM-based reward generation offers three main advantages over fixed, hand-crafted reward designs: (i) the system can seamlessly adapt to new optimization priorities; (ii) the generated rewards help the DRL agent learn more optimal scheduling strategies; and (iii) manual trial-and-error reward tuning is replaced by automated, prompt-driven design. However, our approach also introduces challenges, including potential variability in LLM outputs, computational overhead for reward validation, and the need for careful prompt engineering to ensure alignment with system constraints.

Future work will explore several promising directions. First, extending the framework to multi-objective reinforcement learning would allow the scheduler to balance latency, energy, and accuracy concurrently without repeated re-prompting, with LLMs assisting in dynamic weight adaptation as requirements evolve. Second, fine-tuning lightweight LLM variants for reward function generation would allow on-device deployment at edge nodes, thereby reducing reliance on external API calls and alleviating potential latency or availability issues.

Overall, our work demonstrates that coupling LLM-driven reward generation with DRL provides a powerful and generalizable paradigm for adaptive scheduling in edge computing systems, paving the way toward self-optimizing, goal-aware intelligence for next-generation smart city applications.

Author Contributions: Conceptualization, X.Y. and H.L.; literature review, X.Y. and H.L.; methodology, X.Y. and H.L.; formal analysis, X.Y. and H.L.; writing—original draft preparation, X.Y. and H.L.; visualization, X.Y. and H.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by JSPS KAKENHI, Grant Number JP23K11063, from Japan Society for the Promotion of Science.

Data Availability Statement: The data presented in this article are available upon request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Example of Prompt Design for LLM-Based Reward Function Generation

To enable automatic reward function generation, we designed a carefully engineered prompt for the large language model, ensuring that the generated code was both functionally correct and directly executable in our `MECEnv` simulation environment. The prompt explicitly defines the environment variables, strict implementation constraints, and distinct optimization objectives, guiding the LLM to produce reward functions that align with different scheduling goals.

```
You are tasked with generating Python reward functions for a Deep
Reinforcement Learning environment that schedules object detection
inference tasks to multiple edge servers. The generated functions
will be used directly inside the ‘MECEnv‘ class without any further
modification, so strict adherence to the following rules is required.

ENVIRONMENT DEFINITIONS (STRICT CONSTRAINTS):

- The reward function must be defined exactly as follows:
    def get_reward(self, state, action):
        ...

- You must filter the DataFrame with:
    fetch_state = self.dual_s_base.loc[
        (self.dual_s_base['ram'] == gram) &
        (self.dual_s_base['cores'] == gcores) &
        (self.dual_s_base['workload_cpu'] == gwl_c) &
        (self.dual_s_base['workload_gpu'] == gwl_g) &
        (self.dual_s_base['users_yolo'] == gs1) &
        (self.dual_s_base['users_mnet'] == gs2)
    ]

    ...
```

```
OPTIMIZATION GOALS:

You must produce ten distinct reward functions with different
optimization priorities:
1. Latency-first optimization: Strongly reward low latency and penalize
anything close to threshold.
2. YOLO-priority optimization: Prioritize YOLO capacity over MNet.


RULES:

- All functions must still penalize configurations with latency above
the threshold.
- Each function must visibly differ in weighting and structure, not just
small coefficient changes.
- The reward structure must make the optimization focus obvious from the
code.
- Output format:
    # Candidate 1: <name>
    def get_reward(self, state, action):
        ...
- Each function must be complete and runnable inside the ‘MECEnv‘
  environment without modification.
```

Appendix B. Example of Iterative Prompt for Reward Function Refinement

To further improve the quality of generated reward functions, we designed an iterative prompt that incorporates reflection and refinement based on the performance of previously tested candidates. In this stage, the LLM is instructed to analyze the top-performing reward functions, identify strengths and weaknesses, and propose diverse improvements.

```
In the previous stage, we generated ten candidate reward functions and
selected the top three based on scheduling performance.

Now, your task is as follows:
- Reflect on the strengths and weaknesses of these top three reward
functions.
- Propose diverse improvements rather than small coefficient tweaks.
- Design 10 new candidate reward functions for the next training cycle.

ENVIRONMENT DEFINITIONS (STRICT CONSTRAINTS):

- The reward function must be defined exactly as follows:
    def get_reward(self, state, action):
        ...
```

```
- You must filter the DataFrame with the following:
    fetch_state = self.dual_s_base.loc[
        (self.dual_s_base['ram'] == gram) &
        (self.dual_s_base['cores'] == gcores) &
        (self.dual_s_base['workload_cpu'] == gwl_c) &
        (self.dual_s_base['workload_gpu'] == gwl_g) &
        (self.dual_s_base['users_yolo'] == gs1) &
        (self.dual_s_base['users_mnet'] == gs2)
    ]

    ...

============================
REFERENCE: Top Three Best Performing Reward Functions
============================

# Candidate 1: Reward Based on User Count
def get_reward(self, state, action):
    gram, gcores, gwl_c, gwl_g, gs1_old, gs2_old = state
    u1 = (action // 5) * 4 + 1
    u2 = (action % 5) * 4 + 1

    fetch_state = self.dual_s_base.loc[
        (self.dual_s_base['ram'] == gram) &
        (self.dual_s_base['cores'] == gcores) &
        (self.dual_s_base['workload_cpu'] == gwl_c) &
        (self.dual_s_base['workload_gpu'] == gwl_g) &
        (self.dual_s_base['users_yolo'] == u1) &
        (self.dual_s_base['users_mnet'] == u2)
    ]

    if fetch_state.empty:
        return -20

    time1 = fetch_state.sample().iloc[0]['time_yolo']
    time2 = fetch_state.sample().iloc[0]['time_mnet']
    tm = max(time1, time2)

    if tm <= self.latency_threshold:
        return u1 + u2
    else:
        return -10

# Candidate 2:
    ...

# Candidate 3:
    ...
```

```
============================
RULES
============================
- Each function must be runnable in MECEnvThreatsUnd without modification.
- Each candidate must visibly differ in logic, not only coefficients.
- Output format:
    # Candidate 1: <name>
    def get_reward(self, state, action):
        ...
- Do NOT include markdown code fences.
```

Appendix C. Differences in RULE Constraints Between GPT-4o-mini and Llama-3.1-8B-Instruct

This appendix presents the specific `RULE` constraints used in the reward function generation prompts for GPT-4o-mini and Llama-3.1-8B-Instruct. While both models follow a similar structure, the Llama variant employs additional constraints to ensure syntactic correctness and stability when executed locally, reflecting its weaker code-alignment capability compared to GPT-4o-mini. These distinctions are discussed in Section 4.1 and are provided here for completeness.

Appendix C.1. GPT-4o-mini RULES

```
RULES:

- All functions must still penalize configurations with latency above
  the threshold.
- Each function must visibly differ in weighting and structure, not just
  small coefficient changes.
- The reward structure must make the optimization focus obvious from the
  code.
- Output format:
    # Candidate 1: <name>
    def get_reward(self, state, action):
        ...
- Each function must be complete and runnable inside the ‘MECEnv‘
  environment without modification.
```

Appendix C.2. Llama-3.1-8B-Instruct RULES

```
RULES:

- All functions must still penalize configurations with latency above
  the threshold.
- Each function must visibly differ in weighting and structure, not just
  small coefficient changes.
- The reward structure must make the optimization focus obvious
  from the code.
```

```
- Output format:
    # Candidate 1: <name>
    def get_reward(self, state, action):
        ...
- Each function must be complete and runnable inside the
  MECEnvThreatsUnd environment without modification.
- You MUST use self.dual_s_base instead of self.df to avoid
  missing attribute errors.
- Do NOT include markdown code fences (such as ‘‘‘python‘‘‘ or ‘‘‘);
  only return raw Python code.
- You must generate exactly ten (10) reward functions, no more and no
less.
- Once you have generated all 10 new candidate reward functions,
  output nothing else.
- End your output with the special token <END>.
```

References

1. Syed, A.S.; Sierra-Sosa, D.; Kumar, A.; Elmaghraby, A. IoT in smart cities: A survey of technologies, practices and challenges. *Smart Cities* **2021**, *4*, 429–475. [CrossRef]
2. Zaman, M.; Puryear, N.; Abdelwahed, S.; Zohrabi, N. A review of IoT-based smart city development and management. *Smart Cities* **2024**, *7*, 1462–1501. [CrossRef]
3. Kim, T.h.; Ramos, C.; Mohammed, S. Smart city and IoT. *Future Gener. Comput. Syst.* **2017**, *76*, 159–162. [CrossRef]
4. Soylu, E.; Soylu, T. A performance comparison of YOLOv8 models for traffic sign detection in the Robotaxi-full scale autonomous vehicle competition. *Multimed. Tools Appl.* **2024**, *83*, 25005–25035. [CrossRef]
5. Liu, Q.; Liu, Y.; Lin, D. Revolutionizing target detection in intelligent traffic systems: Yolov8-snakevision. *Electronics* **2023**, *12*, 4970. [CrossRef]
6. Li, Y.; Huang, Y.; Tao, Q. Improving real-time object detection in Internet-of-Things smart city traffic with YOLOv8-DSAF method. *SCient. Rep.* **2024**, *14*, 17235. [CrossRef]
7. Gui, S.; Song, S.; Qin, R.; Tang, Y. Remote sensing object detection in the deep learning era—A review. *Remote Sens.* **2024**, *16*, 327. [CrossRef]
8. Adegun, A.A.; Fonou-Dombeu, J.V.; Viriri, S.; Odindi, J. Ontology-Based Deep Learning Model for Object Detection and Image Classification in Smart City Concepts. *Smart Cities* **2024**, *7*, 2182–2207. [CrossRef]
9. Liang, S.; Wu, H.; Zhen, L.; Hua, Q.; Garg, S.; Kaddoum, G.; Hassan, M.M.; Yu, K. Edge YOLO: Real-time intelligent object detection system based on edge-cloud cooperation in autonomous vehicles. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 25345–25360. [CrossRef]
10. Hu, F.; Mehta, K.; Mishra, S.; AlMutawa, M. A Dynamic Distributed Scheduler for Computing on the Edge. In Proceedings of the 2024 International Symposium on Parallel Computing and Distributed Systems (PCDS), Singapore, 21–22 September 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 1–7. [CrossRef]
11. Liu, Y.; Qu, H.; Chen, S.; Feng, X. Energy efficient task scheduling for heterogeneous multicore processors in edge computing. *Sci. Rep.* **2025**, *15*, 11819. [CrossRef]
12. Fang, C.; Zhang, T.; Huang, J.; Xu, H.; Hu, Z.; Yang, Y.; Wang, Z.; Zhou, Z.; Luo, X. A DRL-driven intelligent optimization strategy for resource allocation in cloud-edge-end cooperation environments. *Symmetry* **2022**, *14*, 2120. [CrossRef]
13. Mekala, M.S.; Patan, R.; Gandomi, A.H.; Park, J.H.; Jung, H.Y. A DRL based 4-r Computation Model for Object Detection on RSU using LiDAR in IIoT. In Proceedings of the 2021 IEEE Symposium Series on Computational Intelligence (SSCI), Orlando, FL, USA, 5–7 December 2021; IEEE: Piscataway, NJ, USA,2021; pp. 1–8. [CrossRef]
14. Nie, L.; Wang, H.; Feng, G.; Sun, J.; Lv, H.; Cui, H. A deep reinforcement learning assisted task offloading and resource allocation approach towards self-driving object detection. *J. Cloud Comput.* **2023**, *12*, 131. [CrossRef]
15. Arulkumaran, K.; Deisenroth, M.P.; Brundage, M.; Bharath, A.A. Deep reinforcement learning: A brief survey. *IEEE Signal Process. Mag.* **2017**, *34*, 26–38. [CrossRef]

16. Karalakou, A.; Troullinos, D.; Chalkiadakis, G.; Papageorgiou, M. Deep reinforcement learning reward function design for autonomous driving in lane-free traffic. *Systems* **2023**, *11*, 134. [CrossRef]
17. Callegaro, D.; Levorato, M.; Restuccia, F. Smartdet: Context-aware dynamic control of edge task offloading for mobile object detection. In Proceedings of the 2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), Belfast, UK, 14–17 June 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 357–366. [CrossRef]
18. Guo, Y.; Zhang, Y.; Wu, L.; Li, M.; Cai, X.; Chen, J. Edge computing service deployment and task offloading based on multi-task high-dimensional multi-objective optimization. *arXiv* **2023**, arXiv:2312.04101. [CrossRef]
19. Kang, D.; Lee, J.; Baek, H. Real-time scheduling for multi-object tracking tasks in regions with different criticalities. *J. Syst. Archit.* **2025**, *160*, 103349. [CrossRef]
20. Huang, Z.; Wu, X.; Dong, S. Multi-objective task offloading for highly dynamic heterogeneous Vehicular Edge Computing: An efficient reinforcement learning approach. *Comput. Commun.* **2024**, *225*, 27–43. [CrossRef]
21. Noh, H.; Shim, B.; Yang, H.J. Adaptive resource allocation optimization using large language models in dynamic wireless environments. *IEEE Trans. Veh. Technol.* 2025, *early access*. [CrossRef]
22. Wu, D.; Wang, X.; Qiao, Y.; Wang, Z.; Jiang, J.; Cui, S.; Wang, F. Netllm: Adapting large language models for networking. In Proceedings of the ACM SIGCOMM 2024 Conference, Sydney, Australia, 4–8 August 2024; pp. 661–678. [CrossRef]
23. Sun, C.; Huang, S.; Pompili, D. LLM-Based Multi-Agent Decision-Making: Challenges and Future Directions. *IEEE Robot. Autom. Lett.* **2025**, *10*, 5681–5688. [CrossRef]
24. Behari, N.; Zhang, E.; Zhao, Y.; Taneja, A.; Nagaraj, D.; Tambe, M. A decision-language model (DLM) for dynamic restless multi-armed bandit tasks in public health. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 3964–4002. [CrossRef]
25. Ma, Y.J.; Liang, W.; Wang, G.; Huang, D.A.; Bastani, O.; Jayaraman, D.; Zhu, Y.; Fan, L.; Anandkumar, A. Eureka: Human-level reward design via coding large language models. *arXiv* **2023**, arXiv:2310.12931. [CrossRef]
26. Kalyuzhnaya, A.; Mityagin, S.; Lutsenko, E.; Getmanov, A.; Aksenkin, Y.; Fatkhiev, K.; Fedorin, K.; Nikitin, N.O.; Chichkova, N.; Vorona, V.; et al. LLM Agents for Smart City Management: Enhancing Decision Support Through Multi-Agent AI Systems. *Smart Cities* **2025**, *8*, 19. [CrossRef]
27. Misbah, S.; Shahid, M.F.; Siddiqui, S.; Khanzada, T.J.S.; Ashari, R.B.; Ullah, Z.; Jamjoom, M. Generative AI-Driven Smart Contract Optimization for Secure and Scalable Smart City Services. *Smart Cities* **2025**, *8*, 118. [CrossRef]
28. Hu, L.; Ni, Q. IoT-driven automated object detection algorithm for urban surveillance systems in smart cities. *IEEE Internet Things J.* **2017**, *5*, 747–754. [CrossRef]
29. Alsubaei, F.S.; Al-Wesabi, F.N.; Hilal, A.M. Deep learning-based small object detection and classification model for garbage waste management in smart cities and iot environment. *Appl. Sci.* **2022**, *12*, 2281. [CrossRef]
30. Khan, S.; Nazir, S.; Khan, H.U. Smart Object Detection and Home Appliances Control System in Smart Cities. *Comput. Mater. Contin.* **2020**, *67*, 895–915. [CrossRef]
31. Guo, S.; Zhao, C.; Wang, G.; Yang, J.; Yang, S. Ec^{2}detect: Real-time online video object detection in edge-cloud collaborative iot. *IEEE Internet Things J.* **2022**, *9*, 20382–20392. [CrossRef]
32. Pudasaini, D.; Abhari, A. Edge-based video analytic for smart cities. *Int. J. Adv. Comput. Sci. Appl.* **2021**, *12*, 1–10. [CrossRef]
33. Wang, N.; Pang, S.; Ji, X.; Wang, M.; Qiao, S.; Yu, S. Intelligent driving task scheduling service in vehicle-edge collaborative networks based on deep reinforcement learning. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 4357–4368. [CrossRef]
34. Panda, S.P.; Banerjee, A.; Bhattacharya, A. User allocation in mobile edge computing: A deep reinforcement learning approach. In Proceedings of the 2021 IEEE International Conference on Web Services (ICWS), Chicago, IL, USA, 5–10 September 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 447–458. [CrossRef]
35. Wang, C.; Yu, X.; Xu, L.; Wang, W. Energy-efficient task scheduling based on traffic mapping in heterogeneous mobile-edge computing: A green IoT perspective. *IEEE Trans. Green Commun. Netw.* **2022**, *7*, 972–982. [CrossRef]
36. Zhang, J.; Lin, Y.; Du, Z.; Zhong, L.; Bao, W.; Si, H.; Djahel, S.; Wu, C. Behavior Cloning with Fuzzy Logic for Accelerating Early-Stage DRL in Edge Offloading. *IEEE Trans. Consum. Electron.* 2025, *in press*. [CrossRef]
37. Hurst, A.; Lerer, A.; Goucher, A.P.; Perelman, A.; Ramesh, A.; Clark, A.; Ostrow, A.; Welihinda, A.; Hayes, A.; Radford, A.; et al. Gpt-4o system card. *arXiv* **2024**, arXiv:2410.21276. [CrossRef]
38. Dubey, A.; Jauhri, A.; Pandey, A.; Kadian, A.; Al-Dahle, A.; Letman, A.; Mathur, A.; Schelten, A.; Yang, A.; Fan, A.; et al. The llama 3 herd of models. *arXiv* **2024**, arXiv:2407.21783. [CrossRef]
39. Lai, P.; He, Q.; Abdelrazek, M.; Chen, F.; Hosking, J.; Grundy, J.; Yang, Y. Optimal edge user allocation in edge computing with variable sized vector bin packing. In Proceedings of the International Conference on Service-Oriented Computing, Hangzhou, China, 12–15 November 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 230–245. [CrossRef]
40. Ge, Z.; Liu, S.; Wang, F.; Li, Z.; Sun, J. Yolox: Exceeding yolo series in 2021. *arXiv* **2021**, arXiv:2107.08430. [CrossRef]

41. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 4510–4520. [CrossRef]
42. Topcuoglu, H.; Hariri, S.; Wu, M.Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.* **2002**, *13*, 260–274. [CrossRef]