Active Prompt Caching in Edge Networks for Generative AI and LLMs: An RL-based Approach

Emna Baccour[1], Aiman Erbad[2], Amr Mohamed[2], Mounir Hamdi[1], and Mohsen Guizani[3]

[1] College of Science and Engineering, Hamad Bin Khalifa University, Qatar Foundation, Doha, Qatar.
[2] College of Engineering, Qatar University, Doha, Qatar.
[3] Mohamed Bin Zayed University of Artificial Intelligence (MBZUAI), Abu Dhabi, UAE.

2025 IEEE Wireless Communications and Networking Conference (WCNC) | 979-8-3503-6836-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/WCNC61545.2025.10978306

***Abstract*—Generative AI (GAI) and Large Language Models (LLMs) have revolutionized natural language processing and content creation. However, their significant computational demands during inference often require cloud servers, which are currently the only viable option for handling complex multi-modal models like GPT-4. The inherent complexity of these models increases latency, posing challenges even within cloud environments. Furthermore, cloud reliance brings other challenges, including high bandwidth consumption to transfer diverse data types. Worse, in personalized GAI applications like virtual assistants, similar prompts frequently occur, causing redundant transmission and computation of replies, which further increases overhead. Accelerating the inference of multi-modal systems is, therefore, critical in artificial intelligence. In this paper, we aim to improve the inference efficiency through prompt caching; if a current prompt is semantically similar to a previous one, the system can reuse the earlier response without invoking the model again. We leverage collaborative edge computing to cache popular replies and store their request embeddings. New prompts are locally processed to extract embeddings, with their qualities determined by the resources available on edge servers. Our problem is formulated as an optimization to manage offloading decisions for GAI tasks, aiming to avoid cloud inferences and minimize latency while maximizing reply quality. Given its non-convex nature, we propose to solve it via Block Successive Upper Bound Minimization (BSUM). Reinforcement learning is employed to actively pre-cache prompts, tackling the complexity of unknown prompt popularity. Our approach demonstrates near-optimal performance, significantly outperforming cloud-only solutions.**

***Index Terms*—Generative AI, LLM, collaborative edge computing, prompts caching, BSUM, RL.**

I. Introduction

The emergence of Generative AI (GAI), particularly Large Language Models (LLM) like GPT-4, has revolutionized complex task handling across various domains [1]. The foundation of GAI typically lies in transformers, which, with billions of parameters, enable impressive performance. However, the massive scale of these models, especially multimodal variants (e.g., DALL-E), demands significant computational resources, requiring cloud-based infrastructure. This reliance on cloud servers introduces several challenges, as transferring diverse data types (e.g., images, audio) consumes considerable bandwidth and high latency, putting strain on network infrastructure. Additionally, the inference process is time-consuming even in powerful environments, which is critical for real-time applications. This challenge motivated major technology companies like OpenAI to develop more lightweight models that can reduce these limitations while maintaining high performance.

The demand for personalized GAI applications, such as virtual assistants and chatbots, has grown significantly in fields like education and customer service. For instance, universities like Deakin in Australia have implemented personalized virtual assistants [2] to support students with enrollment queries and academic resources. In such personalized applications, certain prompts are frequently repeated, especially during periods like exams and registration deadlines, resulting in redundant transmission and computation of responses. This repetition increases computational burden and latency, leading to inefficient use of network resources. In many cases, chatbots rely on pre-built prompts to handle common questions. This highlights the need for more efficient and responsive LLMs that can handle these repetitive interactions with minimal resources, motivating solutions tailored for personalized applications.

Speeding up GAI and LLM inference has become a critical research focus. Edge computing has emerged as a promising solution, bringing LLMs closer to data sources. Proposed approaches include distributed inference [3], partitioning LLMs across edge devices, and quantization [4], reducing model size at the cost of accuracy. Configuring LLM inference tasks under hardware constraints by aggregating GPU, CPU, and disk memory has also been explored [5]. However, LLM's substantial memory requirements (e.g., Llama2-7B needs 28GB) make full model deployment on edge devices challenging. Moreover, the existing works do not address the issue of handling redundant requests efficiently, which hinders personalized GAI adoption and highlights the need for innovative solutions.

A natural approach to reduce resource consumption and latency is to minimize model calls through caching, a well-established technique widely used for web retrieval. In this paper, we introduce two effective strategies to apply caching tailored specifically for GAI and LLM personalized applications: (a) for new prompts, the system checks if a semantically similar prompt exists in the edge cache. If found, it reuses the cached response, avoiding redundant cloud model invocations; (b) We leverage the transformer model to compare prompt similarities, utilizing the prefill phase of the GAI model for embedding generation. The depth of the transformer is adjusted based on available computational resources, affecting the accuracy of the similarity check. If no match is found in

979-8-3503-6836-9/25/$31.00 ©2025 IEEE

the edge cache, the remaining transformer layers and response generation are processed in the cloud, effectively distributing computation across edge and cloud resources. Authors in [6] also address LLM caching in edge environments, focusing only on efficient techniques for prompt comparison. However, they did not consider the resource requirements of their technique and cache management, latency reduction, or enhancing edge capabilities, which are crucial aspects in our approach.

To further enhance performance, we adopt collaborative edge computing [7], combining edge servers and cloud resources to dynamically distribute caching and computation for more efficient and responsive personalized GAI applications. Particularly, (1) we profile transformer model's computational and memory requirements for semantic similarity; (2) We formulate edge intelligence optimization for GAI/LLM caching to minimize latency and maximize quality; (3) We apply BSUM technique to solve the non-convex optimization problem; (4) We use Reinforcement Learning (RL) for active pre-caching, with an Age of Request (AoR) cache replacement strategy, tailored specifically for GAI tasks at the edge; (5) We evaluate the RL-based system using GPT-3 and Llama, showing effectiveness versus cloud-only solutions.

II. Collaborative edge computing for efficient and active prompt caching

A. System model

As shown in Fig. 1, we consider a wireless network composed of a set $\mathcal{K} = \{1, 2, \ldots, K\}$ of K servers, each of which is attached to one Base Station (BS). Servers are grouped into collaboration spaces (i.e., clusters), where they collaborate by sharing resources. In a collaboration space, each server has both caching and computational resources leveraged to serve replies of potential prompts. Let S_k and M_k denote the caching capacity and the RAM memory of the server k, respectively. M_k serves to cache short-term data generated while processing a deep model, for example. P_k and B_k present the computing speed and the bandwidth capacity of k, respectively. The maximum parallel computation capacity of any server is limited and is defined as C_k, to ensure efficient management of multiple requests within the server's capacity.

When a user submits a prompt via application APIs (e.g., ChatGPT) instead of forwarding it directly to the cloud, the prompt is first processed at the edge by generating its embedding. Users' requests are indexed by $r \in \mathcal{R} = \{1, 2, \ldots, R\}$. Each request r is presented by the tuple $< I_r, O_r, E_{r,l} >$, where I_r is the input prompt length, O_r is the output length which is categorized into multiple levels $\{o_1, o_2, \ldots, o\}$ depending on the GAI or LLM model, and $E_{r,l}$ is the embedding size generated using an encoding algorithm l. The collection of potentially cached prompts and their related embeddings and replies within the cluster is indexed by $\mathcal{H} = \{1, 2, \ldots, H\}$. If a semantically similar embedding exists within the cluster, the reply is directly served from the cache.

A user with a request r is assumed to communicate with the nearest BS h_r, referred to as home server. However, in any given collaboration space, servers can exchange data and tasks based on their available resources. The home server may receive the prompt, but embedding computation and matching might occur on different servers within the cluster. For uncached embeddings, the system resorts to the cloud, where the full GAI model or LLM is utilized to compute the reply. Here, we assume joint cluster resources suffice to compute and serve incoming requests at time t, if their replies are cached. Various scenarios for fetching and processing prompts will be detailed in subsequent subsections.

1) Computation and communication models: We define $Cp_{r,l}$ as the computation complexity of the algorithm $l \in \mathcal{L} = \{1, 2, \ldots, L\}$ serving to generate the prompt r embedding. The computation latency of l in a server k is therefore:

$$T^c_{r,k,l} = \frac{Cp_{r,l}}{P_k} \tag{1}$$

In case the resources are not available in the cluster cache, the computation latency of the GAI models in the cloud is defined as T^L. The transmission latency to serve a reply depends on the scenario. However, in general, we have three types of data exchange: (1) the transmission of the prompt r from the user to home server h_r which is equal to:

$$T^u_r = \frac{|I_r|}{\rho_{r,h_r}}, \tag{2}$$

where $|I_r|$ is the size of the input and ρ_{r,h_r} is the data rate between the user and its home server. It is worth noting that the rate depends on the distance of the user from the home BS and it is changing over time depending on the mobility of users; (2) the data transmission between two neighboring servers i and j, which is expressed as follows:

$$T_{r,i,j} = \begin{cases} T^1_{r,i,j} = \frac{|I_r|}{\rho_{i,j}}, \text{ the home server } i \text{ does not} \\ \quad \text{have enough computation resources.} \\ T^2_{r,i,j} = \frac{E_{r,l}}{\rho_{i,j}}, \text{ the computation server } i \\ \quad \text{does not have the reply in its cache.} \\ T^3_{r,i,j} = \frac{|O_r|}{\rho_{i,j}}, \text{ the fetching server } i \\ \quad \text{sends the reply to home server } j. \end{cases} \tag{3}$$

(3) If the reply is not cached on edge, the computing server is notified, and the reply is requested from the cloud c by sending the embedding, costing the system a latency expressed as:

$$T^B_{r,i,c} = \frac{E_{r,l}}{\rho_{i,c}}, \tag{4}$$

B. Generative AI profiling

This section profiles the memory footprint, computation, and latency in generative multi-modal inferences. Multi-modal systems comprise multiple components handling various input/output data types (text, images, audio). Typically, the common element is the multi-layer transformer generating data embeddings for context understanding. Each input type undergoes initial processing: division into smaller parts and linear projection to create a continuous vector space (see Fig. 1). These steps have varying complexities, scaling with

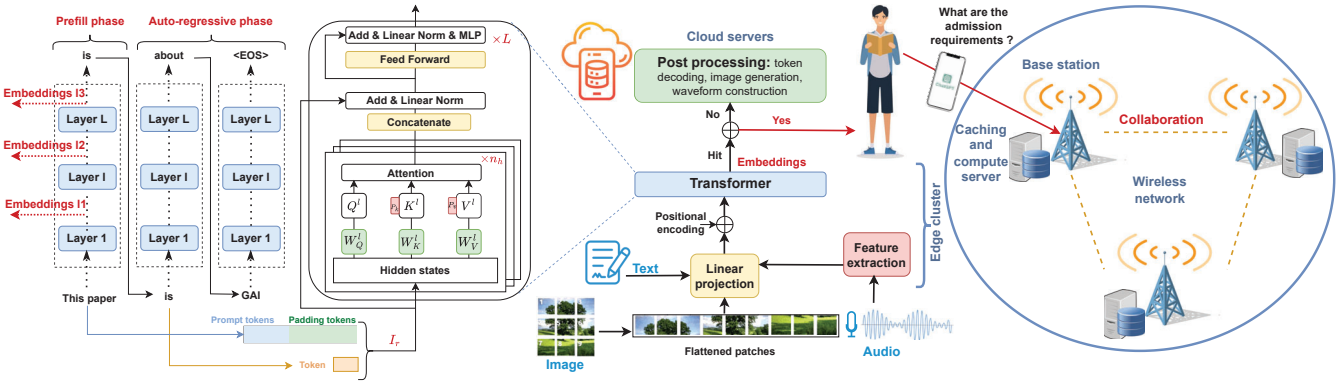

Fig. 1: System model.

the number of parts and projection dimensionality D^{out}. The complexity for projecting N segments using a linear layer is:

$$Cp_r^1 = N \times D_r^{in} \times D^{out}, \tag{5}$$

where D_r^{in} represents the size of each data segment. The output of the projection is the input of the transformer. We propose performing the projection and transformer computation phases at the edge that will serve as the algorithm generating the embeddings. Due to resource constraints and transformers' computational complexity, a server can compute only a limited number of layers. For simplicity, we assume L fixed transformer depths are feasible. Computing more layers improves the model's contextual understanding and embedding quality. If the reply is not found, this computation is not wasted, as the output is sent to the cloud to continue generating the appropriate response. The complexity of generating the response varies with the type of output data. For text, it involves a series of transformers, while image generation includes an image decoder based on the GLIDE model [8], combining diffusion with a UNet architecture. The UNet's encoder captures context through convolutional and pooling layers, while the decoder reconstructs the image using upsampling and convolutional layers. Convolutional task complexity is detailed in [9], [10]. Implementing prompt caching to bypass this computationally intensive phase when possible can achieve substantial gains, reducing latency and computational load.

This paper focuses mainly on computations done on edge servers. In transformers, attention layers are the most computationally intensive, computing relevance between input elements to capture sequence dependencies. Due to their high computational cost, we analyze the arithmetic intensity of these layers. Fig. 1 illustrates a transformer-based LLM with two stages: prefill and auto-regressive. The prefill stage, computed on edge, processes all prompt tokens to produce the embedding. If a semantically similar prompt is not found in the edge, the auto-regressive stage, done on cloud, utilizes generated embedding to produce subsequent tokens.

1) Memory footprint and computation requirements during prefill stage: The memory footprint in a transformer primarily arises from the key-value (KV) cache. The KV cache, related to the Attention mechanism, enables inter-token interaction by storing token key and value matrices for weighted averaging. Let d_T represent the transformer's hidden dimension and d_F the hidden dimension of the Feed-Forward Network (FFN). The transformer's hidden dimension can be calculated as:

$$d_T = d_h \times n_h, \tag{6}$$

where n_h is the number of attention heads, and d_h is the dimension of a single attention head. For the l-th transformer layer, the weight parameters are presented by $W_Q^l, W_K^l, W_V^l \in \mathbb{R}^{d_T \times d_T}$, $W_O^l \in \mathbb{R}^{d_T \times d_T}$, $W_1^l \in \mathbb{R}^{d_T \times d_F}$, and $W_2^l \in \mathbb{R}^{d_F \times d_T}$, where W_Q^l, W_K^l, W_V^l are the matrices of the multi-head attention layer, W_O^l is the output projection matrix, and W_1^l and W_2^l are the MLP weights for two linear transformations (see Fig. 1). Before the prefill stage, all input prompts need to be extended to the maximum token length $S_m \geq |I_r|$ for parallel processing. Given an input prompt $I_r \in \mathbb{R}^{S_m \times d_T}$, the operations during the prefill stage are the calculation of the query weighted matrix, the key weighted matrix, and the value weighted matrix, which are presented respectively by:

$$Q^l = I_r \times W_Q^l, \; K^l = I_r \times W_K^l, \; V^l = I_r \times W_V^l \tag{7}$$

The Self-Attention mechanism can be done as follows:

$$I_{out}^l = softmax(\frac{Q^l {K^l}^T}{\sqrt{d_h}}) V^l W_O^l \tag{8}$$

The FFN mechanism is: $I^{l+1} = ReLu(I_{out}^l \times W_1^l) W_2^l$ (9)

We note that only multiplication operations, quantified as Floating point operations (FLOPs), are taken into account. On these bases, equations in (7) cost the system a computation of $3nS_m d_T^2$, where each parameter is stored using n-byte floating point format. Equation (8) costs the system a computation equal to $2nS_m^2 d_T + nS_m d_T^2$ and equation (9) costs it a computation equal to $2nS_m d_F d_T$. Hence, the overall computation for the inference of l transformer layers is:

$$\begin{aligned} Cp_{r,l} &= Cp_{r,l}^1 + Cp_{r,l}^2 \\ Cp_{r,l}^2 &= l(3nS_m d_T^2 + 2nS_m^2 d_T + nS_m d_T^2 + 2nS_m d_F d_T) \end{aligned} \tag{10}$$

Each edge server generates embeddings using l transformer layers, depending on its computational capacity. The total

memory footprint of KV cache $(K^l, V^l),\ \forall l \leq L$ for all input prompts across all layers is expressed as follows:

$$Mo_l = 2lnS_m d_T \tag{11}$$

This KV data is utilized solely for inference at step t and cached in M_k. Once the inference is complete and the response is delivered, the cache is cleared to free up server memory.

2) Memory footprint and latency in cloud stage: The second stage of the GAI model depends on the type of the output data. However, if the edge servers compute only $l \leq L$ transformer layers, the cloud servers must process $Cp^2_{r,L-l}$ FLOPs to complete the remaining transformation. In case of LLM, the auto-regressive stage is computed next and it involves multiple transformer iterations based on the number of generated tokens. The self-attention and FFN mechanisms remain unchanged, but the calculations for the Query, Key, and Value weight matrices of token $I^l_A \in \mathbb{R}^{1\times d_T}$ are as follows:

$$\begin{aligned} Q^l = I^l_A \times W^l_Q,\ K^l \leftarrow (K^l \oplus I^l_A \times W^l_K),\\ V^l \leftarrow (V^l \oplus I^l_A \times W^l_V) \end{aligned} \tag{12}$$

The total computation latency with an output length of O_r is:

$$\begin{aligned} &T^L_{r,l} = \frac{Cp^3_{r,l}}{P_c}, \quad \text{where } Cp^3_{r,l} = Cp^2_{r,L-l} + L(O_r-1)\times\\ &(3nd^2_T + 2n(S_m + \frac{O_r}{2})d_T + nd^2_T + 2nd_F d_T) \end{aligned} \tag{13}$$

C. Problem formulation

Our goal is to minimize GAI reply generation latency over period T, while maximizing embedding comparison quality by computing the maximum transformer layers per request, subject to resource constraints. Our system uses four decision variables: $Y_{t,r,k}$ (1 if prompt r is computed in server k, 0 otherwise), $X_{t,r,l}$ (1 if r uses transformer depth l), $A_{t,r,k}$ (1 if reply to r is fetched from server k), and $Z_{t,r,k}$ (1 if reply to r is cached in server k). The problem formulation is:

$$\min_{(X_{t,r,l},Y_{t,r,k},A_{t,r,k},Z_{t,r,k})} \quad G^T = \sum_{t=1}^{T}(L^t + \sum_{r=1}^{R}\sum_{l=1}^{L}\alpha_l X_{t,r,l})$$

$$\begin{aligned} L^t = \sum_{r=1}^{R}\Bigg[\sum_{\substack{k=1\\k\neq h^t_r}}^{K}\Bigg[T^1_{t,r,h^t_r,k}Y_{t,r,k} + \sum_{\substack{i=1\\i\neq k}}^{K}\Bigg[\left(T^3_{t,r,i,h^t_r}Y_{t,r,k}A_{t,r,i}\right)\mathbf{1}_{i\neq h^t_r}\\ + \sum_{l=1}^{L}T^2_{t,r,l,i,k}A_{t,r,i}Y_{t,r,k}X_{t,r,l}\Bigg] + \sum_{l=1}^{L}T^c_{t,r,k,l}Y_{t,r,k}X_{t,r,l}\Bigg] + T^u_{t,r} +\\ \mathbf{1}_{(\sum_{j=1}^{K}Z_{t,r,j}=0)}\times\left(T^3_{t,r,c,h^t_r} + \sum_{l=1}^{L}\sum_{j=1}^{K}(T^B_{t,r,j,c}X_{t,r,l}Y_{t,r,j} + T^L_{r,l})\right)\Bigg] \end{aligned} \tag{14a}$$

$$\textbf{s.t} \quad \sum_{k=1}^{K}A_{t,r,k} + (\sum_{j=1}^{K}Z_{t,r,j} == 0) = 1 \quad \forall r \leq R, \forall t \leq T \tag{14b}$$

$$A_{t,r,k} \leq Z_{t,r,k} \quad \forall r \leq R, \forall k \in \mathcal{K}, \forall t \leq T, \tag{14c}$$

$$\sum_{k=1}^{K}Y_{t,r,k} = 1 \quad \forall r \leq R, \forall t \leq T \tag{14d}$$

$$\sum_{l=1}^{K}X_{t,r,l} = 1 \quad \forall r \leq R, \forall t \leq T, \tag{14e}$$

$$\sum_{r=1}^{R}\sum_{l=1}^{L}Cp_{t,r,l}Y_{t,r,k}X_{t,r,l} \leq C_k \quad \forall k \in \mathcal{K}, \forall t \leq T \tag{14f}$$

$$\begin{aligned} &Bd_{t,h^t_r} += \sum_{r=1}^{R}\Big[|O_r| + |I_r|\Big]_{h^t_r},\\ &Bd_{t,i} += \sum_{r=1}^{R}\sum_{k=1}^{K}(|O_r|Y_{t,r,k}A_{t,r,i})\,\mathbf{1}_{\substack{i\neq h^t_r\\ i\neq k}},\\ &Bd_{t,k} += \sum_{r=1}^{R}\sum_{\substack{i=1\\i\neq k}}^{K}\sum_{l=1}^{L}E_{r,l}A_{t,r,i}Y_{t,r,k}X_{t,r,l} +\\ &\left(\sum_{j=1}^{K}Z_{t,r,j} == 0\right)\times\sum_{r=1}^{R}\sum_{l=1}^{L}E_{r,l}Y_{t,r,k}X_{t,r,l}\\ &Bd_{t,k} \leq B_k, \quad \forall k \in \mathcal{K}, \forall t \leq T \end{aligned} \tag{14g}$$

$$\begin{aligned} &m_{t,h^t_r} += \sum_{r=1}^{R}\Big[|O_r| + |I_r|\Big]_{h^t_r},\\ &m_{t,i} += \sum_{r=1}^{R}\sum_{k=1}^{K}\sum_{l=1}^{L}(E_{r,l}Y_{t,r,k}A_{t,r,i}X_{t,r,l})\,\mathbf{1}_{\substack{i\neq h^t_r\\ i\neq k}},\\ &m_{t,k} += \sum_{r=1}^{R}(|I_r|Y_{t,r,k})\,\mathbf{1}_{k\neq h^t_r} + \sum_{l=1}^{L}(d_T l + Mo_l)Y_{t,r,k}X_{t,r,l},\\ &m_{t,k} + \sum_{h=1}^{H}(O_h + \sum_{l=1}^{L}E_{h,l})Z_{t,h,k} \leq S_k + M_k\ \forall k \in \mathcal{K}, \forall t \leq T \end{aligned} \tag{14h}$$

$$X_{t,r,l}, Y_{t,r,k}, A_{t,r,k}, Z_{t,r,k} \in \{0,1\}.\ \forall r \leq R, t \leq T, l \in \mathcal{L}, k \in \mathcal{K}. \tag{14i}$$

The objective function G^T in Eq. (14a) consists of two parts. The first, L^t, represents transmission and computation latencies for different scenarios (see Eq. (1), (2), (3), (4), and (13)). The second reflects embedding quality, determined by the number of transformer layers per request, represented by the quality degradation factor α_l, which decreases with more layers. Both parts are normalized for fair optimization. Constraint (14b) ensures replies are fetched from either the cloud or a single server. Constraint (14c) guarantees replies are fetched only from servers where they are stored. Constraints (14d) and (14e) ensure each request is processed on one server using a single transformer size. Constraints (14f), (14g), and (14h) establish limits on computational, bandwidth, and memory resources, respectively.

D. RL-based Iterative Optimization Control

The optimization problem in (14) is NP-hard due to its non-convex structure and impractical because of the uncertainty in prompt arrivals and their popularity. To address this problem, we propose a two-step solution: (1) for each t, pre-cache potentially requested prompts using Deep Reinforcement Learning (DRL) to solve the caching strategy Z_t; and (2) relax the optimization $G^t(X_t, Y_t, A_t)$ via an iterative control process, namely BSUM, transforming it into a convex problem.

1) RL-based active prompts pre-caching: DRL is suitable for our problem as it learns prompt popularity patterns over time, adapts to environmental changes, and dynamically optimizes caching decisions. The DRL agent interacts with

the environment and learns through actions and rewards to optimize policy. RL solves Markov decision processes (MDP), represented by $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$, $\mathcal{R}$, and $\mathcal{P}$ are sets of states, actions, rewards, and transition probabilities, respectively, and γ is the discount factor. The agent is trained episodically until convergence, with each episode consisting of T steps. We propose using Proximal Policy Optimization (PPO), a policy-based approach suitable for both discrete and continuous action spaces. Algorithm 1 presents the pseudocode for the adopted PPO framework.

Each time step's state is represented as $s_t = \{\mathcal{R}_{t-1}, \{h_1, h_2, \ldots, h_R\}_{t-1}\}$, where $\mathcal{R}_{t-1}$ represents the set of requests of the previous time step and $\{h_1, h_2, \ldots, h_R\}_{t-1}$ is the set of their home servers (line 8). For each received request, the RL agent decides for each server whether to store the prompt, its reply, and its embeddings. However, this results in a large binary action space of size $K \times R$, making the system difficult to scale. To address this, we propose designing the action space as a set $\{a_t^1, a_t^2, \ldots a_t^R\}$ of R elements (line 9). Each element $0 \leq a_t^r \leq 2^K$ is then decoded into K binary decisions as follows:

$$a_t^r = \left[b_i | b_i = \left\lfloor \frac{a_t^r}{2^i} \right\rfloor \mod 2, \quad 0 \leq i < K \right] \tag{15}$$

Based on the caching decisions, new prompts are stored by replacing cached items that have the higher Age of Request (AoR). The AoR of an item s in server k is defined as follows:

$$AoR_{k,s}^t = \max(0, AoR_{k,s}^{t-1} + (1 - \mathbf{1}_{h_s \in \{h_1, \ldots, h_R\}}) \times (O_s + \sum_{l=1}^{L} E_{s,l}) \times \vartheta_{t,s,k} - \delta)$$

$$\text{where} \quad \vartheta_{t,s,k} = \begin{cases} 1/2, & \text{if } \sum_{\substack{i=1 \\ i \neq k}} A_{t,s,i} \geq 1, \\ 1, & \text{otherwise.} \end{cases} \tag{16}$$

The AoR, similar to Age of Information (AoI), measures the freshness of prompts based on request frequency, with a decay factor δ. At each time step t, if prompt s is requested from home server k, its AoR remains unchanged. Otherwise, AoR increases by the storage size of s and its embeddings, reduced by half if s is requested from another BS. This prioritizes smaller prompts and embeddings over larger ones (line 23).

Next, after updating the cluster cache, the agent receives a reward expressed as:

$$rw_t = \frac{\Delta}{G^t(X_t^*, Y_t^*, A_t^*)}, \tag{17}$$

which is inversely correlated with our objective (Δ is a constant). The agent learns to maximize rewards, which, in turn, minimizes latency by optimizing the caching strategy (line 24). The optimal $G^t(X_t^*, Y_t^*, A_t^*)$ is derived through the BSUM iterative control that will be detailed in what follows (lines 13-22). The training process generates samples of $\{s_t, a_t, r_t, s_{t+1}\}$ over multiple episodes and stores them in an experience memory $\mathbb{D}$. A mini-batch is then sampled (lines 26) to update the main policy $\mathbf{\Pi}^{(\theta)}$. Finally, the targeted optimal policy is synchronized with the main policy by replacing θ_p with θ, ensuring monotonic improvement (line 28).

Algorithm 1 RL-based Iterative Optimization Control

1: **Input:** T, $\mathcal{R}$, $\mathcal{K}$, $\mathcal{H}$, $\mathcal{L}$ M_k, S_k, P_k, B_k, C_k $\forall k \in \mathcal{K}$
2: **Output:** X^*, A^*, Y^*, Z^*
3: $\theta_p \leftarrow \theta$
4: **Repeat**
5: Randomly set $\mathcal{R}_{t-1}$
6: Update AoR using (16)
7: **for** $t = 1 : T$ **do**
8: $s_t \leftarrow \{\mathcal{R}_{t-1}, \{h_1, h_2, \ldots, h_R\}_{t-1}\}$
9: Select $a_t = \{a_t^1, a_t^2, \ldots a_t^R\}$ based on ϵ-greedy policy
10: Decode a_t using (15)
11: Cache replacement Z_t^* based on (16) s.t S_k, $\forall k \in \mathcal{K}$
12: $s_{t+1} \leftarrow \{\mathcal{R}_t, \{h_1, h_2, \ldots, h_R\}_t\}$
13: **BSUM iterative control**
14: Initialize $i = 0$, $\eta > 0$
15: Find initial feasible points $(X^{(0)}, A^{(0)}, Y^{(0)})$
16: **Repeat**
17: Choose convex set W^i
18: Let $X_j^{(i+1)} \in \min_{X_j \in \mathbb{X}} G_j(X_j, X^{(i)}, A^{(i)}, Y^{(i)})$
19: Find $A_j^{(i+1)}$ and $Y_j^{(i+1)}$ by solving (23) and (24)
20: $i = i + 1$
21: **Until** $||\frac{G_j^{(i)} - G_j^{(i+1)}}{G_j^{(i)}}|| \leq \eta$
22: Generate (X_t^*,A_t^*,Y_t^*) by rounding ($X_j^{(i+1)}$, $A_j^{(i+1)}$,$Y_j^{(i+1)}$)
23: Update AoR using (16)
24: $rw_t \leftarrow \frac{\Delta}{G^t(X_t^*, Y_t^*, A_t^*)}$
25: Save (s_t, a_t, r_t, s_{t+1}) in the experience memory $\mathbb{D}$
26: Sample a mini-batch of (s_n, a_n, r_n, s_{n+1}) from $\mathbb{D}$
27: Using the mini-batch, update the main policy $\mathbf{\Pi}^{(\theta)}$
28: $\theta_p \leftarrow \theta$
29: **Until** Convergence

2) BSUM-based Iterative Optimization Control: we apply the Block Successive Upper Bound Minimization (BSUM) method [11], to transform the problem into a convex one. BSUM is a distributed algorithm that handles block-structured optimizations by breaking the initial problem into smaller subproblems and iteratively optimizing individual blocks of variables. The BSUM optimization problem is written as:

$$\min_x g(x_1, x_2, \ldots, x_J), \quad \text{s.t. } x_j \in W_j, \forall j \leq J \tag{18}$$

where W_j is a closed convex set, and x_j is a block of variables. At each iteration i, a single block is optimized using:

$$x_j^i \in \arg \min_{x_j \in W_j} g(x_j, x_{-j}^{i-1}) \quad \text{where } x_{-j}^{i-1} = (x_1^{i-1}, \ldots, x_{j-1}^{i-1}, x_{j+1}^{i-1}, \ldots, x_J^{i-1}) \tag{19}$$

Since the overall problem is non-convex, BSUM introduces a proximal upper-bound function to ensure convergence. The proximal upper-bound function $h(x_j, y)$ must satisfy:

Assumption 1:

- $h(x_j, y) = g(y)$ and $h(x_j, y) > g(x_j, y_{-j})$: The upper-bound is a global upper-bound for the original function, meaning it always overestimates the objective function.

- $h'(x_j, y; q_j)\big|_{x_j=y_j} = g'(y; q)$: The gradient of the upper-bound at $x_j = y_j$ matches the gradient of the original function, preserving first-order behavior for optimization.

The commonly used schemes for choosing the proximal upper-bound function are quadratic upper-bound or linear upper-bound. For simplicity purposes, we construct our upper-bound by adding quadratic penalization to the objective function to ensure convexity of G_j^t and convergence of the problem:

$$G_j^t(X_j, X^{(i)}, A^{(i)}, Y^{(i)}) = G^t(X_j, \tilde{X}, \tilde{A}, \tilde{Y}) + \frac{\rho}{2}\|X_j - \tilde{X}\|^2 \quad (20)$$

This function is similarly applied to the variables A_j and Y_j, where $\rho > 0$ is the penalty parameter. At each iteration i, BSUM updates each block iteratively through minimizing the proximal upper-bound function until it converges to both a coordinate wise minimum and a stationary solution.

Remark 1: The BSUM algorithm converges to an η-optimal solution with sub-linear convergence, taking $O(\log(1/\eta))$ iterations. The η-optimal solution $x_j^\eta \in W_j$ is defined as:

$$x_j^\eta \in \{x_j \mid x_j \in W_j, h(x_j, x^i, y^i) - h(x_j^*, x^i, y^i) \leq \eta\} \quad (21)$$

where $h(x_j^*, x^i, y^i)$ is the optimal value of $h(x_j, y)$ with respect to x_j.

To implement BSUM within our iterative control framework, we introduce the variable sets $\mathbb{X} = \{X_j : \sum_{l=1}^{L} X_{r,l} = 1, X_{r,l} \in [0,1]\}$, $\mathbb{Y} = \{Y_j : \sum_{k=1}^{K} Y_{r,k} = 1, Y_{r,k} \in [0,1]\}$ and $\mathbb{A} = \{A_j : \sum_{k=1}^{K} A_{r,k} + (\sum_{j=1}^{K} Z_{r,j} == 0) = 1 \ \& \ A_{r,k} \leq Z_{r,k}, A_{r,k} \in [0,1]\}$ as the convex sets of X_j, Y_j, and A_j, respectively. To reduce the problems complexity, we relax the variables, allowing their values to lie within the interval $[0, 1]$.

At each iteration, the vectors $\tilde{X}, \tilde{A}, \tilde{Y}$, from the previous step i, are updated by solving the following optimization problems for iteration $i + 1$:

$$X_j^{(i+1)} \in \min_{X_j \in \mathbb{X}} L_j(X_j, X^{(i)}, A^{(i)}, Y^{(i)}) \quad (22)$$

$$A_j^{(i+1)} \in \min_{A_j \in \mathbb{A}} L_j(A_j, A^{(i)}, X^{(i+1)}, Y^{(i)}), \quad (23)$$

$$Y_j^{(i+1)} \in \min_{Y_j \in \mathbb{Y}} L_j(Y_j, Y^{(i)}, X^{(i+1)}, A^{(i+1)}). \quad (24)$$

Algorithm 1 is initialized by $i = 0$ with feasible points $(X^{(0)}, A^{(0)}, Y^{(0)})$ and caching decisions $Z^{(0)}$ derived from the RL system (lines 14-15). It proceeds iteratively, selecting the convex set at each step. In iteration $i + 1$, the solution is updated by solving the optimization problems (22), (23), and (24) until: $||\frac{G_j^{(i)} - G_j^{(i+1)}}{G_j^{(i)}}|| \leq \eta$.

Afterward, we round the relaxed variables $(X_j,\ Y_j,\ A_j)$ into binary vectors. The optimal values (X_t^*, A_t^*, Y_t^*) of step t represent the stability point, corresponding to the stationary solution that achieves coordinate-wise minimum (line 22).

III. System evaluation

This section presents the results of our RL-based cooperative system for replies caching and computation. We simulate 3 BSs with 5G data rates uniformly distributed between 1-20 Gbps, and user-server data rates uniformly distributed from 50-72.2 Mbps. NVIDIA Jetson AGX Xavier edge servers are used, with memory uniformly distributed between 1-2 GB, 8 GB RAM, and processor speed uniformly distributed between 5-10 TFLOPs/s, with a maximum computation of 10^5 TFLOPs per time slot. Cloud computation speed is 10 TFLOPs. We evaluate the well-known GPT-3 LLM and compare it to Llama-2 (see Table I). The arrival of user requests at each BS per time slot is modeled as a Poisson process with a rate of 4. The number of possibly requested prompts is equal to 50. The popularity of these prompts follows a Zipf distribution with a parameter of 0.5, while their sizes are uniformly distributed between 24 and 24 × S_m bits. Three transformer depths (96, 40, 30) are used, with reply lengths uniformly drawn from $\{128, 256, 512, 1024, 2048, 4096\}$. RL and optimization parameters are presented in Table II.

TABLE I: LLMs with public configurations [1].

LLM model	L	n_h	d_h	d_F	S_m
GPT-3	96	96	128	49152	2048
Llama-2	80	64	128	32768	4096

TABLE II: Hyper-parameters of the system.

Parameter	**Description**	**Value**
γ	Discount factor	0.1
	Learning rate	0.001
Policy	DNN policy	MLP, 2 layers of 64 neurons
η	Optimality parameter	0.2
ρ	Upper-bound parameter	0.2
δ	Vanishing factor	60
α	Embedding quality factor	[0,0.5,1]
Δ	Reward parameter	100

Fig. 2(a) illustrates the convergence behavior of our iterative optimization solved using CVXPY, as we vary the upper-bound parameter ρ. The results indicate that by iteration 12, the BSUM problems in (22), (23), and (24) converge to a coordinate-wise minimum and reach an optimal stationary point. We compared our relaxed solution to the binary-variable problem with higher computational complexity, and observed that the achieved minimum is close to optimal.

Fig. 2(b) shows cumulative rewards across training episodes with different arrival rates. We averaged the rewards over every 10 iterations for better presentation. Initially, the agent focuses on exploration to identify optimal prompts for caching, reducing latency, and improving embedding quality. As training progresses, exploration decreases, allowing the agent to pass smoothly to a refined policy and reach convergence.

Fig. 2(c) presents the hit ratio in edge, while varying the server cache size and the Zipf parameter. As we can see, the hit ratio is high when servers have the largest cache size since more prompts can be stored at the edge, reducing the need for cloud LLM calls and avoiding the lengthy computation of the full model. Regardless of the Zipf parameter, a larger cache

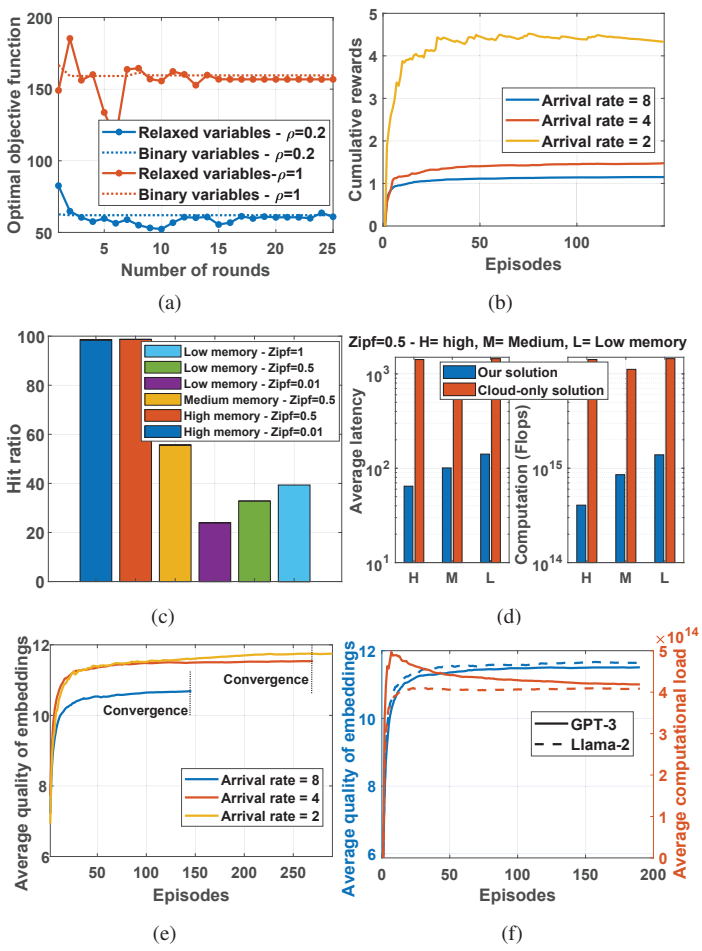

Fig. 2: Performance of our proposed system.

size combined with a collaborative BS cluster enables the edge to store a wide range of replies, effectively caching both highly popular and less popular prompts. On the other hand, smaller cache sizes result in reduced hit ratio and increased latency. When the cache size is limited, the influence of the Zipf α becomes more pronounced. With α close to 0, all prompts have nearly equal popularity, resulting in near-random caching, which lowers the hit ratio. A higher Zipf α means a smaller set of prompts dominates in popularity, making it easier to cache the most frequently requested prompts.

Fig. 2(d) compares our caching-based approach to the cloud-only solution, demonstrating that caching significantly reduces reply latency in personalized applications across various system configurations. This substantial improvement is attributed to our system's ability to bypass the full model computation by leveraging cached responses and processing only the prefill phase for semantic comparison.

Fig. 2(e) illustrates the impact of varying the arrival rate on the quality of generated embeddings. As the number of incoming requests increases, the embedding quality decreases, since the computation capacity of the cluster is allocated to more prompts. This forces the system to utilize fewer transformer layers for semantic comparison, resulting in reduced embedding quality. Fig. 2(f) compares the computational load of both GPT-3 and Llama-2. Notably, GPT-3 imposes a higher computational load due to its more complex architecture. In contrast, Llama-2, with its slightly lighter design, demonstrates improved performance over GPT-3, as it allows for greater computational availability. This enhances the quality of semantic comparisons and accelerates response latency.

IV. Conclusion

In this paper, we addressed the challenges of deploying GAI and LLMs for personalized applications, focusing on exhaustive resources, latency, and redundant requests. We propose to cache replies of popular prompts on edge and use a depth-adaptive transformer for semantic comparison adjusted to available edge resources. By integrating collaborative edge computing and using BSUM with RL for active cache management, our approach reduces cloud dependency and improves responsiveness. Validated on GPT-3 and Llama, our framework demonstrates an effective solution for optimizing GAI applications in wireless networks.

References

[1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2303.18223

[2] D. University, "Deakin's genie: A virtual digital assistant out of the bottle," https://www.deakin.edu.au/about-deakin/news-and-media-releases/articles/deakins-genie-a-virtual-digital-assistant-out-of-the-bottle, 2019, accessed: 2024-10-02.

[3] M. Zhang, J. Cao, X. Shen, and Z. Cui, "Edgeshard: Efficient llm inference via collaborative edge computing," 2024. [Online]. Available: https://arxiv.org/abs/2405.14371

[4] X. Zhang, J. Liu, Z. Xiong, Y. Huang, G. Xie, and R. Zhang, "Edge intelligence optimization for large language model inference with batching and quantization," in *2024 IEEE Wireless Communications and Networking Conference (WCNC)*, 2024, pp. 1–6.

[5] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," 2023. [Online]. Available: https://arxiv.org/abs/2303.06865

[6] H. Zhu, B. Zhu, and J. Jiao, "Efficient prompt caching via embedding similarity," 2024. [Online]. Available: https://arxiv.org/abs/2402.01173

[7] E. Baccour, A. Erbad, A. Mohamed, M. Guizani, and M. Hamdi, "Collaborative hierarchical caching and transcoding in edge network with ce-d2d communication," *Journal of Network and Computer Applications*, vol. 172, p. 102801, 2020.

[8] A. Nichol, P. Dhariwal, A. Ramesh, P. Shyam, P. Mishkin, B. McGrew, I. Sutskever, and M. Chen, "Glide: Towards photorealistic image generation and editing with text-guided diffusion models," 2022.

[9] E. Baccour, N. Mhaisen, A. A. Abdellatif, A. Erbad, A. Mohamed, M. Hamdi, and M. Guizani, "Pervasive ai for iot applications: A survey on resource-efficient distributed artificial intelligence," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2366–2418, 2022.

[10] E. Baccour, A. Erbad, A. Mohamed, M. Hamdi, and M. Guizani, "Rl-distprivacy: Privacy-aware distributed deep inference for low latency iot systems," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 4, pp. 2066–2083, 2022.

[11] M. Hong, T.-H. Chang, X. Wang, M. Razaviyayn, S. Ma, and Z.-Q. Luo, "A block successive upper bound minimization method of multipliers for linearly constrained convex optimization," 2014. [Online]. Available: https://arxiv.org/abs/1401.7079