# FaSei: Fast Serverless Edge Inference with Synergistic Lazy Loading and Layer-wise Caching

Zhaowu Huang[†], Fang Dong[†*], Xiaolin Guo[†], and Daheng Yin[‡]
[†]School of Computer Science and Engineering, Southeast University, Nanjing, China
[‡]School of Computing Science, Simon Fraser University, Burnaby, BC, Canada
Email: {zwh, fdong, xlinguo}@seu.edu.cn, daheng_yin@sfu.ca

IEEE INFOCOM 2025 - IEEE Conference on Computer Communications | 979-8-3315-4305-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/INFOCOM55648.2025.11044642

***Abstract*—Serverless edge computing (SEC) provides low-latency, resource-efficient deep learning (DL) services but faces a significant cold start time due to the loading of large DL models. Existing methods for cold starts include full and partial container caching. The former may be inefficient because large DL models cannot be cached in the resource-limited SEC; The latter only caches common packages for sharing, while user-specified DL models still need to be loaded before execution. We identify model lazy loading to mitigate cold start, which begins inference with a shallow model while lazily loading deeper layers in a pipeline manner. However, naively using lazy loading can result in significant inference bubbles because model loading time is typically much longer than inference time. To address this, we propose FaSei, a fast serverless edge inference method with synergistic model lazy loading and layer-wise caching, to reduce application completion time (ACT). Considering the impact of heterogeneous model-layer behaviors and SEC resources on ACT, we jointly optimize lazy loading, layer-wise caching, and function placement. We formulate it as an integer nonlinear programming problem and then design an approximation algorithm with a theoretical performance guarantee. Extensive experiments demonstrate that FaSei achieves up to $6.7\times$ speedup in reducing ACT.**

***Index Terms*—serverless edge computing, DL inference, cold start**

## I. Introduction

With the development of deep learning (DL), many artificial intelligence applications [1] have emerged, such as face recognition, object detection, and autonomous driving. To deliver low latency and resource-efficient DL inference [2], Serverless Edge Computing (SEC) [3]–[8] is proposed as a potential paradigm, which uses an event-driven model to provide services on the edge servers in proximity to the end devices. Specifically, Function-as-a-Service (FaaS) [9] is the prominent and widely adopted implementation [10] to provide service for users in SEC. Functions are launched on-demand based on specified event triggers, such as HTTP requests [11].

[*]The corresponding author is Fang Dong.

This work is supported by National Natural Science Foundation of China under Grants, No. 62232004, Jiangsu Provincial Frontier Technology Research and Development Program under Grant BF2024070, Shenzhen Key S&T Special Projects under Grant KJZD20240903100814018, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No.BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No.93K-9, and partially supported by Collaborative Innovation Center of Novel Software Technology and Industrialization and Collaborative Innovation Center of Wireless Communications Technology.

However, this trigger-based on-demand service provision paradigm introduces cold start time [3], [12] when invoking new functions. In SEC, a cold start contains three stages [13], [14]: (1) initiating a new container, (2) configuring its software environment, e.g., loading Python libraries. and (3) completing application-specific initialization in user code or file, such as loading the required DL models. The final stage stands as the predominant one, especially in edge inference applications, primarily due to the enormous size and resource-intensive nature of DL models, which exacerbates the cold start problem. For example, when evaluated on the VGG16, a representative DL model with a size of 528MB [15], the model loading time is approximately 1.96s [15], which comprises $83.48\%$ of the total cold start time and is roughly $7.08\times$ longer than the inference time. Thus, the long model loading time is intolerable for DL inference. *Hence, how to reduce the cold start time caused by model loading is a crucial problem.*

Many existing studies have focused on the cold start problem and reduced the cold start time, yet they are insufficient to resolve the above issues. Some approaches mitigate the cold start problem through full container caching and keep-alive [7], [16]–[21]. However, due to SEC's limited resources, it is not feasible to cache all containers or even an entire large model, as a fully initialized container for the large DL models will occupy hundreds or thousands of MBs in memory. Other approaches employ partial container caching or container-sharing techniques [22]–[25], which use the idea of Zygote to pre-import a portion of container packages, and the remaining necessary packages are loaded before execution. These methods mainly address the first and second stages of the cold start, which inevitably require the model file to be fully loaded before execution. Additionally, some approaches consider model tensor sharing [26] and model transformation [15] techniques, reusing tensors or model structures to reduce the cold start time associated with model loading. Nevertheless, they are still unable to reduce the cold start time of the first invocation of a non-reusable model.

Instead of loading the entire DL before execution, we have identified a unique opportunity of model lazy loading to reduce the long cold start time, where only a part of model layers is loaded to start the inference process, and subsequent model layers are loaded in parallel during the inference process (Fig. 1(a) shows an ideal case). This is possible because DL models execute sequentially and do not depend on subsequent model parameters, allowing inference to commence with only a portion of the model loaded [27], [28]. This significantly reduces the cold start time and the completion time. Intuitively, one might think loading only the first layer initially and lazily loading the rest could minimize cold start time. However, this naively using lazy loading does not fully unlock its potential. Recall that the model loading time is significantly longer than the inference time. Loading the model layer by layer during inference will result in inference bubbles due to the waiting time for subsequent layers to load, as shown in Fig. 1(b).

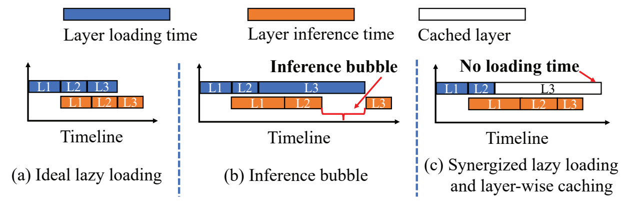


Fig. 1. The illustration of model lazy loading.

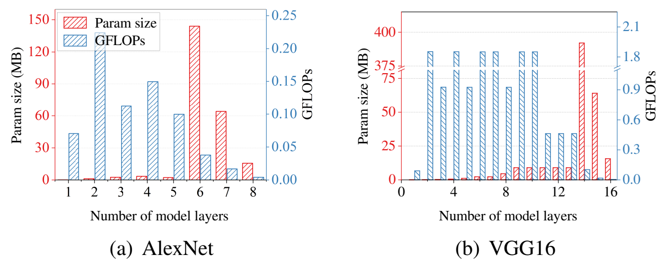


Fig. 2. The illustration of model layer parameter size and FLOPs of AlexNet and VGG16. FLOPs increase nearly inversely with layer parameter size.

Thus, we argue that model lazy loading and layer-wise caching must be synergized to eliminate inference bubbles and mitigate cold start. Specifically, we can cache part of model layers in advance, and the rest is loaded lazily during inference, as shown in Fig. 1(c). However, determining which model layers to cache or lazily load presents challenges. **Firstly, varying model layer behaviors affect inference bubbles.** DL model layers exhibit heterogeneous parameter sizes and behaviors. Specifically, parameter sizes and giga floating-point operations (GFLOPs) vary among layers. The loading and inference latency generally scale proportionally with layer size and GFLOPs. However, GFLOPs increase nearly inversely with layer size [29], as shown in Fig. 2. Intuitively, caching layers with longer loading time may help eliminate inference bubbles, as shown in Fig. 1(c), while these layers also consume more memory, limiting the number of layers that can be cached within the same memory budget. Additionally, because the inference is sequential, each layer's position and caching state affect inference bubbles. **Secondly, heterogonous SEC resources affect inference bubbles.** The heterogeneity of edge server resources means that the same model layer can have different execution time across servers, which will affect inference bubbles under the same model layer cache profile. Therefore, it is necessary to consider function placement when determining the layer-wise caching strategy, which further complicates the problem. To deliver fast edge inference, it is essential to address two questions: *(1) Which model layers should be loaded lazily versus cached? (2) Which edge server should functions be placed on?*

Motivated by the above analysis, in this paper, we propose FaSei, a fast serverless edge inference method with efficient synergistic model lazy loading and layer-wise caching. Specifically, FaSei formulates the joint optimization problem of lazy loading, layer-wise caching, and function placement as integer non-linear programming (INLP) to minimize the application completion time (ACT) in SEC, which has been proven to be an NP-hard problem. This formulation aims to answer the above two problems theoretically, comprehensively considering the heterogeneity of model layers and the SEC environment. FaSei essentially considers fine-grained layer-wise model execution optimization, where model loading and inference are tightly coupled and interdependent, forming a multi-stage sequential process. As a result, ACT is expressed as a nonlinear multi-stage recursively maximized form. To solve it, we creatively relax the multi-stage recursive maximization expression into a multi-stage independent maximization form. Based on this relaxation, the original INLP problem is given a chance to be transformed into an integer linear programming (ILP) problem. Then, we design an approximation algorithm and prove the theoretical gap between the approximate solution and the optimal solution of the original INLP problem. Finally, to evaluate the performance of the proposed method, we conduct extensive experiments and compare it against state-of-the-art approaches. The results show that FaSei significantly improves the application performance, achieving up to a 6.7× speedup in ACT reduction.

The main contributions of this paper are as follows:

- We propose a fast serverless edge inference method with synergistic model lazy loading and layer-wise caching.
- We formulate a joint optimization problem of lazy loading, layer-wise caching, and function placement as an INLP. Then, we design a novel algorithm to solve it and analyze its achievable approximation ratio.
- We conduct extensive experiments to evaluate our methods. The results demonstrate that our proposed methods achieve up to 6.7× speedup in terms of ACT reduction.

## II. System Model and Problem Formulation

### A. System Model

We consider an SEC network consisting of a set $\mathcal{N}$ of geographically distributed edge servers that provide end users with serverless computing services. Each edge server $n \in \mathcal{N}$ is equipped with limited computing $C_n$ resources. Each edge server $n$ allocates a memory capacity $M_n$ for caching, which limits containers that can be cached. These edge servers are interconnected through the network and the transmission delay for a unit of data between edge server $n$ and $n'$ is denoted as $d_{nn'}$.

Let a set $\mathcal{J}$ represent all requests from end users for DL-driven applications. For a request $j \in \mathcal{J}$, it is submitted to the nearest edge server $u_j$, and the edge server either processes it locally or forwards it to other edge servers. We recognize that each request $j$ is associated with a DL model $K_j$. Let $c_j$ denote the required computing resources by request $j$. Each $K_j$ has a hierarchical structure composed of a set of model layers $k \in K_j$ [30], [31]. Let $Q_{jk}$ denote the workload size of model layer $k \in K_j$. Each DL model $K_j$ is deployed in a container as a function to handle request $j$. When a model is loaded into a container, its layers exhibit heterogeneous behaviors, with the loading time $s_{jk}$ and memory footprint $\mu_{jk}$ varying across layers for each request $j \in \mathcal{J}$ and each model layer $k \in K_j$. Additionally, a base container $b$ without pre-loaded model layers incurs a cold start time of $\mu_{jb}$ and requires $s_{jb}$ memory.

## *B. Problem Formulation*

Requests may be scheduled to any server that has the required model deployed. We introduce binary variables $x_{jn}$ to indicate whether request $j$ is scheduled to edge server $n$ ($x_{jn} = 1$) or not ($x_{jn} = 0$). Specifically, $x_{jn} = 1$ means the function associated with request $j$ will be placed on server $n$. For request $j$'s function, it can only be placed on exactly one edge server. This implies

$$\sum_{n \in \mathcal{N}} x_{jn} = 1, \forall j \in \mathcal{J}. \tag{1}$$

For each request, some or all of the model layers can be cached in advance to reduce cold start time. We introduce binary variables $y_{jnk}$ to indicate whether model layer $k$ for request $j$ is cached on the edge server $n$ ($y_{jnk} = 1$) or not ($y_{jnk} = 0$). Similarly, $y_{jnb}$ represents whether a base container for request $j$ is cached on the edge server $n$. Obviously, the model layers can only be cached on edge server $n$ if the corresponding function is deployed to run on that edge server. Thus, we have

$$y_{jnk} \leq x_{jn}, \forall j \in \mathcal{J}, n \in \mathcal{N}, k \in K_j. \tag{2}$$

Similarly, a model layer $k \in K_j$ can be cached into request $j$'s function cached on edge server $n$ if and only if the base container $b$ required by request $j$ has already been cached on edge server $n$. Therefore, the relationship between variables $y_{jnk}$ and $y_{jnb}$ needs to satisfy

$$y_{jnk} \leq y_{jnb}, \forall j \in \mathcal{J}, n \in \mathcal{N}, k \in K_j. \tag{3}$$

Caching function containers on edge servers consumes memory resources. For each edge server $n \in \mathcal{N}$, the total memory resources required for caching functions must not exceed its memory capacity $M_n$.

$$\sum_{j \in \mathcal{J}} x_{jn} \Big( \sum_{k \in K_j} y_{jnk} s_{jk} + y_{jnb} s_{jb} \Big) \leq M_n, \forall n \in \mathcal{N}. \tag{4}$$

Each request requires a certain amount of computing resources to execute inference. For each edge server $n \in \mathcal{N}$, the sum of computing resources required by all requests scheduled to it must not exceed its computing capacity $C_n$.

$$\sum_{j \in \mathcal{J}} x_{jn} c_j \leq C_n, \forall n \in \mathcal{N}. \tag{5}$$

When the request $j$'s function is scheduled to an edge server $n$, and if the required base container is not already cached on, it will involve initializing the base container (i.e., the cold start of the base container), which includes setting up the environment and dependency packages. The cold start time of a base container required by request $j$ is

$$T_c(j) = \sum_{n \in \mathcal{N}} x_{jn} (1 - y_{jnb}) \mu_{jb}, \forall j \in \mathcal{J}. \tag{6}$$

Upon invoking request $j$'s function, model layers that have not been cached are required to be loaded. The loading time of model layer $k$ for request $j$ is denoted as

$$T_l^k(j) = (1 - \sum_{n \in \mathcal{N}} x_{jn} y_{jnk}) \mu_{jk}, \forall j \in \mathcal{J}, k \in K_j. \tag{7}$$

Due to the heterogeneity of edge servers, the execution time of a request $j$ may vary across different edge servers. We use $\xi_n$ to represent a factor that captures the impact of edge server $n$ capability on execution time [32]. The execution time of request $j$'s model layer $k$ can be expressed as

$$T_e^k(j) = \sum_{n \in \mathcal{N}} x_{jn} \frac{Q_{jk}}{c_j} \xi_n, \forall j \in \mathcal{J}, k \in K_j. \tag{8}$$

If request $j$'s function is implemented on edge server $n$, then transmitting request $j$ to edge server $n$ will incur a transmission time $T_t(j)$,

$$T_t(j) = \sum_{n \in \mathcal{N}} x_{jn} d_{u_j n} D_j, \forall j \in \mathcal{J}, \tag{9}$$

where $D_j$ is the data size of request $j$ and $d_{u_j n}$ is the unit transmission delay from server $u_j$ to $n$.

Considering that lazy loading allows the parallel loading of subsequent model layers that are not cached during execution. The execution of a model layer can only start after it has been loaded. The total time to execute and load the first $k$ model layers of request $j$ can be expressed as

$$T_f^k(j) = \max\{\sum_{i=1}^{k} T_l^i(j), T_f^{k-1}(j)\} + T_e^k(j). \tag{10}$$

Our optimization goal is to minimize the overall ACT of all requests. We formulate the joint optimization problem of lazy loading, layer-wise caching, and function placement as

$$\textbf{LLFP}: \min_{\mathbf{x},\mathbf{y}} \sum_{j \in \mathcal{J}} (T_c(j) + T_f^{|K_j|}(j) + T_t(j))$$
$$s.t.\ (1) - (5),$$

where $T_f^{|K_j|}(j)$ denotes the total time to execute and load the full model of request $j$. Constraint (1) ensures that the function can only be placed on one edge server. Constraints (2) and (3) ensure the position and order of layer-wise container cache. Constraints (4) and (5) ensure that the memory and computing resource capacity of each edge server is met.

### C. Complexity Analysis

The LLFP problem is proven to be NP-hard. To illustrate this, let's consider a special case, LLFP-A, where each edge server, with sufficient computing resources and bandwidth, is constrained by limited caching memory, while the transmission time of each function is ignored. LLFP-A can be viewed as a variant of the classical bin packing problem, with functions equating to items and edge servers serving as bins. Given the well-known NP-hard nature of the bin packing problem, we can conclude that the more general LLFP problem is similarly NP-hard.

## III. Algorithm Design

Due to the NP-hard nature of the LLFP problem, finding its optimal solution in polynomial time is impractical. To tackle this, we transform the LLFP problem into a simpler linear programming form, develop an approximation algorithm, and provide a theoretical analysis of the proposed algorithm.

### A. Problem Transformation

The LLFP problem involves quadratic terms, making it a quadratic programming problem that is challenging to solve. Therefore, by introducing an auxiliary variable $z_{jnk}$ to replace $x_{jn}y_{jnk}$, Equations (4), (6), and (7) can be transformed into linear forms, subject to the following constraints

$$z_{jnk} \le x_{jn}, \forall j \in \mathcal{J}, n \in \mathcal{N}, \tag{11}$$

$$z_{jnk} \le y_{jnk}, \forall j \in \mathcal{J}, n \in \mathcal{N}, k \in K_j, \tag{12}$$

$$x_{jn} + y_{jnk} - 1 \le z_{jnk}, \forall j \in \mathcal{J}, n \in \mathcal{N}, k \in K_j, \tag{13}$$

where $z_{jnk} = 1$ if and only if $x_{jn} = 1$ and $y_{jnk} = 1$. By using auxiliary variables $z_{jnk} = 1$, the quadratic terms can be replaced, allowing Equations (4), (6), and (7) to be rewritten

$$\sum_{j \in \mathcal{J}} \Big( \sum_{k \in K_j} z_{jnk} s_{jk} + z_{jnb} s_{jb} \Big) \le M_n, \forall n \in \mathcal{N}. \tag{14}$$

$$T_c(j) = \sum_{n \in \mathcal{N}} (1 - z_{jnb}) \mu_{jb}, \forall j \in \mathcal{J}. \tag{15}$$

$$T_l^k(j) = (1 - \sum_{n \in \mathcal{N}} z_{jnk}) \mu_{jk}, \forall j \in \mathcal{J}, k \in K_j. \tag{16}$$

In addition, the LLFP problem's recursive and maximum terms make it hard to solve. Therefore, we apply an approximation technique, relaxing and rewriting Equation (10) into

$$T_f^k(j) = T_l^1(j) + \sum_{i=2}^{k+1} \max\{T_l^i(j), T_e^{i-1}(j)\}. \tag{17}$$

When $k = |K_j|$, we can find $i = |K_j| + 1$ out of range of model layers. Thus, we define $T_l^{|K_j|+1} = 0$ for ease of representation.

After the above problem transformation, we can relax and rewrite the LLFP problem into LLFPR, which can be formulated as

$$\textbf{LLFPR}: \min_{\mathbf{x},\mathbf{y}} \sum_{j \in \mathcal{J}} (T_c(j) + T_f^{|K_j|}(j) + T_t(j))$$
$$s.t.\ (1) - (3), (5), (11) - (14),$$

where $T_f^{|K_j|}(j)$ is the total time to execute and load the entire model of request $j$ redefined by Equation (17). In the algorithm analysis part, we will theoretically analyze the relationship between these two problems.

Additionally, to further linearize the LLFPR problem, we employ an auxiliary real variable $G_{jk}$ instead of $\max\{T_l^k(j), T_e^{k-1}(j)\}$, subject to the following constraints

$$G_{jk} \ge T_l^k(j), \forall j \in \mathcal{J}, k \in K_j \setminus 1, \tag{18}$$

$$G_{jk} \ge T_e^{k-1}(j), \forall j \in \mathcal{J}, k \in K_j \setminus 1. \tag{19}$$

Constraints (18) and (19) ensure that $G_{jk}$ can always take the maximum value between $T_l^k(j)$ and $T_e^{k-1}(j)$. Through the above transformation, the LLFPR problem is further converted into the following linearized form

$$\textbf{LLFPR-ILP}: \min_{\mathbf{x},\mathbf{y}} \sum_{j \in \mathcal{J}} (T_c(j) + T_l^1 + \sum_{k=2}^{|K_j|+1} G_{jk} + T_t(j))$$
$$s.t.\ (1) - (3), (5), (11) - (14), (18), (19).$$

### B. Approximate Algorithm

For the transformed LLFPR-ILP, we designed an approximation algorithm with performance guarantees using randomized rounding, as detailed in Algorithm 1.

**Algorithm 1** Approximate Algorithm for LLFPR-ILP

```
1: Relax $x_{jn}$, $y_{jnb}$, $y_{jnk}$, $z_{jnb}$, and $z_{jnk}$ into real variables
   in range of $[0, 1]$ and obtain the LP form of LLFPR-ILP
2: Solve the LLFPR-LP problem to obtain the optimal solution
   $\tilde{x}_{jn}$, $\tilde{y}_{jnb}$, $\tilde{y}_{jnk}$, $\tilde{z}_{jnb}$, $\tilde{z}_{jnk}$, $\tilde{g}_{jk}$
3: Set $X_{jn}$, $Y_{jnb}$, $Y_{jnk}$, $Z_{jnb}$, and $Z_{jnk}$ to 0, $G_{jk} = \tilde{g}_{jk}$
4: repeat
5:     for $n \in \mathcal{N}$, $j \in \mathcal{R}$ do
6:         Set $X_{jn} \leftarrow 1$ with probability $\tilde{x}_{jn}$
7:         if $X_{jn} = 1$ then
8:             Set $Y_{jnb} \leftarrow 1$ with probability $\frac{\tilde{y}_{jnb}}{\tilde{x}_{jn}}$
9:         end if
10:        if $X_{jn} = 1$ and $Y_{jnb} = 1$ then
11:            Set $Z_{jnb} \leftarrow 1$ with probability $\frac{\tilde{z}_{jnb}}{\tilde{x}_{jn}\tilde{y}_{jnb}}$
12:        end if
13:    end for
14:    for $n \in \mathcal{N}$, $j \in \mathcal{R}$, $k \in K_j$ do
15:        if $X_{jn} = 1$ and $Y_{jnb} = 1$ then
16:            Set $Y_{jnk} \leftarrow 1$ with with probability $\frac{\tilde{y}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnb}}$
17:        end if
18:        if $X_{jn} = 1$ and $Y_{jnk} = 1$ then
19:            Set $Z_{jnk} \leftarrow 1$ with with probability $\frac{\tilde{z}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnk}}$
20:        end if
21:    end for
22: until $X_{jn}$, $Y_{jnb}$, $Y_{jnk}$, $Z_{jnb}$, and $Z_{jnk}$ define a feasible
    solution
```

First, we transform the LLFPR-ILP problem from its integer linear programming (ILP) form into a linear programming (LP) form, referred to as the LLFPR-LP problem, by relaxing the integer variables to real ones. We can employ an LP solver [33], [34] to find an optimal solution to the LP problem within polynomial time, denoted as $\tilde{x}_{jn}$, $\tilde{y}_{jnb}$, $\tilde{y}_{jnk}$, $\tilde{z}_{jnb}$, $\tilde{z}_{jnk}$, and $\tilde{g}_{jk}$ (lines 1-2). However, apart from the real-valued variable $G_{jk}$, solutions of the other variables usually fail to meet the integer constraints of the LLFPR-ILP problem due to their real values. To solve it, we use a randomized rounding technique to convert real solutions to integer values.

Specifically, for each request $j$, we define $X_{j,n}$ as an i.i.d event where request $j$ is placed on edge server $n$. Our algorithm interprets the fractional solutions as probabilities of these events occurring. The algorithm places the request $j$ on the edge server $n$ with probability $\tilde{x}_{jn}$ and does not place it with probability 1-$\tilde{x}_{jn}$ (line 6). We use $Y_{jnb}$ and $Y_{jnk}$ to represent i.i.d events where the empty container required by request $j$ is cached on edge server $n$ and the corresponding model layer is pre-loaded into the cached container, respectively. Since the edge server can only cache the required container and pre-load the corresponding model layer if the request is placed on $n$, the algorithm sets $Y_{jnb}$ and $Y_{jnk}$ to 1 with probabilities $\frac{\tilde{y}_{jnb}}{\tilde{x}_{jn}}$ and $\frac{\tilde{y}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnb}}$. Similarly, considering Constraints (11)-(13), we set $Z_{jnb}$ and $Z_{jnk}$ to 1 with probabilities $\frac{\tilde{z}_{jnb}}{\tilde{x}_{jn}\tilde{y}_{jnb}}$ and $\frac{\tilde{z}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnk}}$ (lines 7-20). However, This randomized approach may violate constraints, so we repeatedly check and, if necessary, rerun the procedure from lines 4 to 22 until a feasible solution is obtained. As shown in the algorithm analysis part, this process typically finds a feasible solution within a few iterations.

## C. Algorithm Analysis

We now theoretically evaluate the performance of the approximation algorithm. Let $P(\cdot)$ denote the probability of an i.i.d event. According to Algorithm 1, the probability of setting variable $X_{jn}$ to 1 is $P(X_{jn}=1)=\tilde{x}_{jn}$, the probabilities of setting $Y_{jnb}$, and $Y_{jnk}$ to 1 are

$$\begin{aligned}&P(Y_{jnb}=1)=P(Y_{jnb}=1|X_{jn}=1)P(X_{jn}=1)\\&=\frac{\tilde{y}_{jnb}}{\tilde{x}_{jn}}\tilde{x}_{jn}=\tilde{y}_{jnb},\end{aligned} \tag{20}$$

$$\begin{aligned}&P(Y_{jnk}=1)=P(Y_{jnk}=1|X_{jn}=1,Y_{jnb}=1)\times\\&P(X_{jn}=1)P(X_{jnb}=1)=\frac{\tilde{y}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnb}}\tilde{x}_{jn}\tilde{y}_{jnb}=\tilde{y}_{jnk}.\end{aligned} \tag{21}$$

**Lemma 1.** *Algorithm 1 can guarantee that each request will not be placed on multiple servers with high probability.*

*Proof.* The sum of the probabilities that each request $j$ is placed on all edge servers is

$$\sum_{n\in\mathcal{N}}P(X_{jn}=1)=\sum_{n\in\mathcal{N}}\tilde{x}_{jn}=1. \tag{22}$$

Equation (22) holds because of Constraint (1). Therefore, each request $j\in\mathcal{R}$ will not be placed on multiple edge servers with high probability. □

**Lemma 2.** *Algorithm 1 can obtain a feasible solution for the LLFPR-ILP problem with high probability.*

*Proof.* According to Lemma 1, Constraint (1) can be satisfied with a high probability for any request $j$, making variable $X_{jn}$ feasible with a high probability in each round. Equations (20)-(21) ensure that constraints (2)-(3) can also be satisfied, rendering variables $Y_{jnb}$ and $Y_{jnk}$ feasible with a high probability in each round. Similar to Equations (20)-(21), the probabilities of setting $Z_{jnb}$ and $Z_{jnk}$ to 1 are

$$\begin{aligned}&P(z_{jnb}=1)=P(z_{jnb}=1|X_{jn}=1,Y_{jnb}=1)\times\\&P(X_{jn}=1)P(Y_{jnb}=1)=\frac{\tilde{z}_{jnb}}{\tilde{x}_{jn}\tilde{y}_{jnb}}\tilde{x}_{jn}\tilde{y}_{jnb}=\tilde{z}_{jnb}.\end{aligned} \tag{23}$$

$$\begin{aligned}&P(z_{jnk}=1)=P(z_{jnk}=1|X_{jn}=1,Y_{jnk}=1)\times\\&P(X_{jn}=1)P(Y_{jnk}=1)=\frac{\tilde{z}_{jnk}}{\tilde{x}_{jn}\tilde{y}_{jnk}}\tilde{x}_{jn}\tilde{y}_{jnk}=\tilde{z}_{jnk}.\end{aligned} \tag{24}$$

Equations (23) and (24) ensure that Constraints (11)-(13) hold. Therefore, Algorithm 1 can obtain feasible values for $Z_{jnb}$ and $Z_{jnk}$ with high probability in each round. All variables obtained by our algorithm in each round rarely violate the constraints, enabling us to find a feasible solution for LLFPR-ILP within a few rounds. □

**Lemma 3.** *Algorithm 1 can satisfy each server's computing and cache memory capacity constraints in expectation.*

*Proof.* For each edge server $n\in\mathcal{N}$, the expected computing resources required to place all requests on it is

$$E(\sum_{j\in\mathcal{J}}X_{jn}c_j)=\sum_{j\in\mathcal{J}}P(X_{jn}=1)c_j=\sum_{j\in\mathcal{J}}\tilde{x}_{jn}c_j\le C_n, \tag{25}$$

where Inequality (25) holds because of Constraint (5). Similarly, the expected memory footprint of all containers including model layers cached on them is

$$\begin{aligned}&E(\sum_{j\in\mathcal{J}}(\sum_{k\in K_j}Z_{jnk}s_{jk}+Z_{jnb}s_{jb}))\\&=\sum_{j\in\mathcal{J}}(\sum_{k\in K_j}\tilde{z}_{jnk}s_{jk}+\tilde{z}_{jnb}s_{jb})\le M_n.\end{aligned} \tag{26}$$

The first equality holds due to Equations (23) and (24), and the last inequality holds due to Constraint (4). Therefore, the computing resources required and the total cache memory footprint on each edge server will not exceed its computing and cache memory capacities in expectation. □

**Lemma 4.** *The expected total ACT obtained by Algorithm 1 equals the optimal solution of the LLFPR-LP problem. Let $OL$ represent the optimal solution to the LLFPR-LP problem.*

*Proof.* The expected total ACT obtained by Algorithm 1 is

$$\begin{aligned}&E(\sum_{j\in\mathcal{J}}((1-\sum_{n\in\mathcal{N}}Z_{jnb})\mu_{jb}+(1-\sum_{n\in\mathcal{N}}Z_{jn1})\mu_{j1}+\sum_{i=2}^{|K_j|+1}G_{ji}\\&+\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j))=\sum_{j\in\mathcal{J}}((1-\sum_{n\in\mathcal{N}}\tilde{z}_{jnb})\mu_{jb}+\sum_{i=2}^{|K_j|+1}\tilde{g}_{ji}\\&+(1-\sum_{n\in\mathcal{N}}\tilde{z}_{jn1})\mu_{j1}+\sum_{n\in\mathcal{N}}\tilde{x}_{jn}d_{u_jn}D_j)=OL.\end{aligned} \tag{27}$$

□

**Lemma 5.** *Algorithm 1 obtains $\sum_{j\in\mathcal{J}}(\mu_{jb} + \mu_{j1} - \sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb} + Z_{jn1}\mu_{j1}))$ value that is no worse than $\delta_1\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + (1-\delta_1)OP$, where $OP$ is the optimal solution to the LLFPR-ILP problem, and $\delta_1 = \sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j - \Omega}}$.*

*Proof.* The expected value of $\sum_{j\in\mathcal{J}}(\mu_{jb} + \mu_{j1} - \sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb} + Z_{jn1}\mu_{j1}))$ can be expressed as

$$E(\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1} - \sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1}))) = \sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) - E(\sum_{j\in\mathcal{J}}\sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1}). \quad (28)$$

We denote the left of (28) as $\Omega$ and $\mu_{jb}+\mu_{j1}$ as $\sigma_j$. We have

$$E(\sum_{j\in\mathcal{J}}\sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1})) = \sum_{j\in\mathcal{J}}\sigma_j - \Omega. \quad (29)$$

Each term $Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1}$ in Equation (29) is independent and can be set to 0 or 1 with appropriate normalization. Based on Chernoff bound [35], we can get for $0\le\delta_1\le 1$

$$P(\sum_{j\in\mathcal{J}}\sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1})) \le (1-\delta_1)(\sum_{j\in\mathcal{J}}\sigma_j - \Omega) \le e^{-\frac{\delta_1^2(\sum_{j\in\mathcal{J}}\sigma_j-\Omega)}{2}}. \quad (30)$$

Due to $\Omega \le OL \le OP$, we can further get

$$P(\sum_{j\in\mathcal{J}}\sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1}) \le (1-\delta_1)(\sum_{j\in\mathcal{J}}\sigma_j - OP)) \le P(\sum_{j\in\mathcal{J}}\sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1})) \le (1-\delta_1)(\sum_{j\in\mathcal{J}}\sigma_j-\Omega) \le e^{-\frac{\delta_1^2(\sum_{j\in\mathcal{J}}\sigma_j-\Omega)}{2}}. \quad (31)$$

To make the lower bound as small as possible, we take $e^{\frac{-\delta_1^2(\sum_{j\in\mathcal{J}}\sigma_j-\Omega)}{2}} \le \frac{1}{e^{|\mathcal{N}|}}$, which means that the lower bound will converge rapidly to 0 as the number of edge servers increases. So $\delta_1$ must satisfy $\delta_1 \ge \sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j-\Omega}}$. In practice, $\sum_{j\in\mathcal{J}}\sigma_j - \Omega$ is generally much larger than $|\mathcal{N}|$, making $0\le\delta_1\le 1$. We set $\delta_1 = \sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j-\Omega}}$. Therefore, we have

$$E(\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1} - \sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1}))) \le \sum_{j\in\mathcal{J}}\sigma_j - (1-\delta_1)(\sum_{j\in\mathcal{J}}\sigma_j - OP) = \delta_1\sum_{j\in\mathcal{J}}\sigma_j + (1-\delta_1)OP. \quad (32)$$

□

**Lemma 6.** *Algorithm 1 obtains $\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \sum_{i=2}^{|K_j|+1}G_{ji})$ value that is no worse than $(1+\sqrt{\frac{3|\mathcal{N}|}{\Psi}})OP$.*

*Proof.* Similarly, we use $\Psi$ to denote the expectation value of $\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \sum_{i=2}^{|K_j|+1}G_{ji})$ and use $\mathcal{G}$ to denote $\sum_{i=2}^{|K_j|+1}G_{ji}$. Based on Chernoff bound [35], we can get for $0\le\delta_2\le 1$,

$$P(\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \mathcal{G}) \ge (1+\delta_2)\Psi) \le e^{-\frac{\delta_2^2\Psi}{3}}. \quad (33)$$

Due to $\Psi \le OL \le OP$, we can further obtain

$$P(\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \mathcal{G}) \ge (1+\delta_2)OP) \le P(\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \mathcal{G}) \ge (1+\delta_2)\Psi) \le e^{-\frac{\delta_2^2\Psi}{3}}. \quad (34)$$

To make the upper bound as small as possible, we take $e^{\frac{-\delta_2^2\Psi}{3}} \le \frac{1}{e^{|\mathcal{N}|}}$, which implies that the upper bound converges rapidly to 0 as the number of edge servers increases [22]. Similarly, we set $\delta_2$ to $\sqrt{\frac{3|\mathcal{N}|}{\Psi}}$, which allows us to get

$$\sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \mathcal{G}) \le (1+\sqrt{\frac{3|\mathcal{N}|}{\Psi}})OP. \quad (35)$$

□

**Theorem 1.** *The solution to the LLFPR-ILP problem obtained by Algorithm 1 can be guaranteed to be no more than $\sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j-\Omega}}\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + (2-\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + \sqrt{\frac{3|\mathcal{N}|}{\Psi}})OP$ with high probability.*

*Proof.* According to Lemmas 5 and 6, the solution to the LLFPR-ILP problem obtained by Algorithm 1 can be

$$\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1} - \sum_{n\in\mathcal{N}}(Z_{jnb}\mu_{jb}+Z_{jn1}\mu_{j1})) + \sum_{j\in\mathcal{J}}(\sum_{n\in\mathcal{N}}X_{jn}d_{u_jn}D_j + \sum_{i=2}^{|K_j|+1}G_{ji}) \le \delta_1\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + (1-\delta_1)OP + (1+\delta_2)OP. \quad (36)$$

Therefore, the solution acquired by Algorithm 1 will be no worse than $\sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j-\Omega}}\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + (2-\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + \sqrt{\frac{3|\mathcal{N}|}{\Psi}})OP$ with high probability. □

**Theorem 2.** *Our proposed problem transformation method ensures the solution obtained by Algorithm 1 for LLFP is no worse than $(1+\epsilon)(\sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}}\sigma_j-\Omega}}\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + (2-\sum_{j\in\mathcal{J}}(\mu_{jb}+\mu_{j1}) + \sqrt{\frac{3|\mathcal{N}|}{\Psi}})P^*)$, where $P^*$ denotes the optimal solution to the problem LLFP, and $\epsilon = \frac{\max Q_{jk}}{\min Q_{jk}}, \forall k\in K_j$.*

*Proof.* Constraints (18) and (19) guarantee that $G_{jk}$ can always take the maximum value between $T_l^k(j)$ and $T_e^{k-1}(j)$. Thus, the LLFPR and LLFPR-ILP problems are equivalent transformations, with identical optimal values.

Next, we analyze the relationship between the LLFPR and the original LLFP problems. Constraints (11)-(13) ensure that substituting $x_{jn}y_{jnk}$ with the auxiliary variable $x_{jnk}$ is an equivalent transformation, preserving the optimal value of the

LLFP problem. However, observing that Equation (10) does not equate to Equation (17), we find that there exists a gap between the optimal solutions of the LLFP and LLFPR problems. We delve into three cases to understand the relationship between the optimal solutions of the two problems.

**Case 1:** For $\forall k \in K_j \setminus 1$, if $T_l^k(j) \geq T_e^{k-1}(j)$, then we can get $\max\{\sum_{k=1}^{|K_j|} T_l^k(j), T_f^{|K_j|-1}(j)\} + T_e^{|K_j|}(j) = \sum_{k=1}^{|K_j|} T_l^k(j) + T_e^{|K_j|}(j) = T_l^1(j) + \sum_{k=2}^{|K_j|+1} \max\{T_l^k(j), T_e^{k-1}(j)\}$ according to Equations (10) and (17). In this case, the problem LLPFR is equivalent to LLPF with identical optimal values, that is $P^* = OP$.

**Case 2:** $\forall k \in K_j \setminus 1$, if $T_l^k(j) \leq T_e^{k-1}(j)$, then we have $\max\{\sum_{k=1}^{|K_j|} T_l^i(j), T_f^{|K_j|-1}(j)\} + T_e^{|K_j|}(j) = T_l^1(j) + \sum_{k=1}^{|K_j|} T_e^k(j) = T_l^1(j) + \sum_{k=2}^{|K_j|+1} \max\{T_l^k(j), T_e^{k-1}(j)\}$ according to Equations (10) and (17). So we have $P^* = OP$.

**Case 3:** We consider the case except for the above two cases, where the relationship between $T_l^k(j)$ and $T_e^{k-1}(j)$ is uncertain. We use $\tau_k$ to denote the time between the start of execution at model layer $k$ and the start of execution at model layer $k+1$. Thus, $P^*$ can be expressed as $P^* = T_l^1(j) + \sum_{k=1}^{|K_j|} \tau_k$. Since model layer $k$ can only start to execute after its loading is complete, that is, the execution of model layer $k$ needs to wait for the completion of its corresponding loading. Conversely, once model layer $k$ has finished loading, it can proceed to load model layer $k+1$ without having to wait for the completion of layer $k$'s execution. Consequently, we observe that the value of Equation (17) may exceed that of Equation (10) only when the preceding model layer $i$ of layer $k$ experiences a loading time shorter than the execution time of layer $i+1$. This insight lets us get the following relationships

$$OP \leq T_l^1(j) + \sum_{k=1}^{|K_j|} (\tau_k + \Delta_k). \tag{37}$$

where $\Delta_k = \max\{T_e^{k-1}(j) - T_l^k(j), 0\}, \forall k \in K_j$. Based on Equation (37), we can further get that

$$\frac{OP}{P^*} \leq \frac{T_l^1(j) + \sum_{k=1}^{|K_j|}(\tau_k + \Delta_k)}{T_l^1(j) + \sum_{k=1}^{|K_j|} \tau_k} \leq 1 + \frac{\sum_{k=1}^{|K_j|} \Delta_k}{\sum_{k=1}^{|K_j|} \tau_k}$$
$$\leq 1 + \frac{\max \Delta_k}{\min \tau_k} \leq 1 + \frac{\max T_e^k(j)}{\min T_e^k(j)} = 1 + \frac{\max Q_{jk}}{\min Q_{jk}}. \tag{38}$$

According to Theorem 1, we can get that the solution obtained by Algorithm 1 will not be worse than $\sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}} \sigma_j - \Omega}} \sum_{j\in\mathcal{J}} (\mu_{jb} + \mu_{j1}) + (2 - \sum_{j\in\mathcal{J}} (\mu_{jb} + \mu_{j1}) + \sqrt{\frac{3|\mathcal{N}|}{\Psi}})OP$. Based on the above analysis, we can conclude that the solution returned by Algorithm 1 will be no worse than $(1 + \frac{\max Q_{jk}}{\min Q_{jk}})(\sqrt{\frac{2|\mathcal{N}|}{\sum_{j\in\mathcal{J}} \sigma_j - \Omega}} \sum_{j\in\mathcal{J}} (\mu_{jb} + \mu_{j1}) + (2 - \sum_{j\in\mathcal{J}} (\mu_{jb} + \mu_{j1}) + \sqrt{\frac{3|\mathcal{N}|}{\Psi}})P^*)$ for the problem LLFP. □

## IV. Evaluation and Analysis

We now evaluate the performance of the proposed algorithm against existing studies by extensive simulations.

### A. Experiment Settings

We consider an edge computing system consisting of 5 edge servers. By default, we randomly set their container caching memory and computational resource capacities within the range of $[500, 600]$MB and $[100, 500]$ Giga Floating Point Operations Per Second (GFLOPS), respectively [36]. The bandwidth between edge servers is randomly set within $200 - 300$ Mbps [25]. We have selected 10 representative DL models as the workload for our applications, encompassing AlexNet, VGG11, VGG13, VGG16, VGG19, Inception V3, ResNet18, ResNet34, ResNet50, ResNet101, which are with the size of $46 - 548$MB. For these models, we used images with dimensions of $3 \times 224 \times 224$ as input data with a size of approximately 0.14MB. We measured the parameter data size and the number of FLOPs for each layer of these models as the model data to be loaded and the workload to be calculated. The size of the base container containing all the required packages except the DL model is 105MB [15]. In addition, the average loading time for a unit of data (specifically, 1 MB) is 0.00371212s [15]. The computing resource requirements for these applications are randomly set at an average of 55 GFLOPS by default [15]. To simulate the different computing speeds of heterogeneous servers, we randomly set the computing impact factor in the range $[1, 2]$.

To evaluate the performance of our method FaSei, we have implemented the following methods as benchmarks.

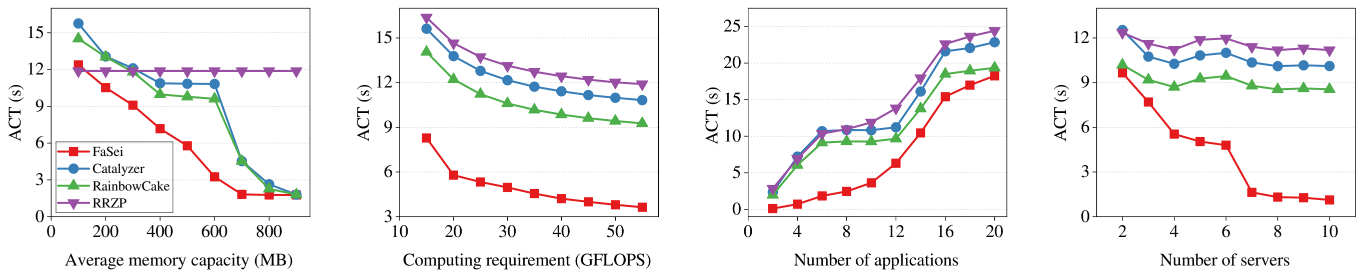
- RRZP [22]: RRZP only considers caching the base container, which can fork a copy for sharing when needed. It also proposes an approximate algorithm to place applications in a way that minimizes their completion time.
- RainbowCake [37]: RainbowCake makes full container caching decisions by balancing cold start time and memory consumption. When the cache capacity is insufficient, it caches the base containers instead.
- Catalyzer [38]: Catalyzer caches the full container preferentially for applications with longest cold start time.

Note that RainbowCake and Catalyzer are proposed for the cloud and do not address the placement of applications in the SEC. To ensure fairness, we use a greedy approach to place applications on the server that minimizes the ACT.

### B. Experiment Results

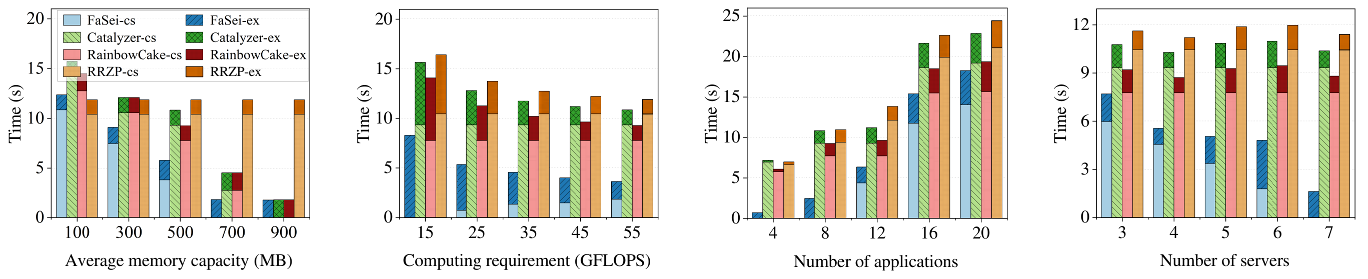
*1) Overall performance:* We evaluate the overall performance of FaSei, Catalyzer, RainbowCake, and RRZP by varying the average memory capacity, computing requirements, the number of applications, and the number of edge servers.

**The impact of different average memory capacities:** We first check how the average memory capacity affects the overall ACT by increasing it from 200MB to 900MB while fixing the number of applications at 10. From Fig. 3(a), we can see that FaSei can always outperform other algorithms. Compared with Catalyzer, RainbowCake, and RRZP, FaSei can achieve up to 2.98×, 2.65×, and 6.7× in ACT reduction. This verifies the effectiveness of our algorithm in reducing overall ACT. We can also see that the overall ACT of FaSei,



(a) Average memory capacity (b) Computing requirement (c) Number of applications (d) Number of servers

Fig. 3. The impact of (a) average memory capacity, (b) computing requirement, (c) number of applications, and (d) number of servers on the overall ACT.



(a) Average memory capacity (b) Computing requirement (c) Number of applications (d) Number of servers

Fig. 4. The impact of (a) average memory capacity, (b) computing requirement, (c) number of applications, and (d) number of servers on the breakdown of overall ACT in terms of cold start time (-cs) and execution time (-ex).

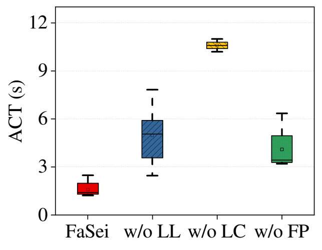
Catalyzer, and RainbowCake decreases with the increase of the average memory capacity. This is mainly because larger memory capacity allows more containers to be cached on the edge servers, reducing the cold start time. Such phenomena can be discovered in Fig. 4(a). In contrast, the overall ACT of RRZP stays the same as the cache memory increases. This is because RRZP only caches and shares the base container, requiring the entire model to be loaded before inference can be performed. As shown in Fig. 4(a), the cold start time and execution time of the RRZP remain unchanged. Therefore, RRZP can achieve better performance when the average memory capacity for caching is small (e.g., 100MB and 200MB), but when the average memory capacity for caching is increased, it cannot cache more models to further reduce the overall ACT.

**The impact of different computing requirements:** Then, we illustrate the impact of computing requirements on the overall ACT by varying it from 15GFLOPS to 55GFLOPS. As shown in Fig. 3(b), our proposed FaSei significantly outperforms all other competitors, achieving up to $2.98\times$, $2.55\times$, and $3.27\times$ reduction in the overall ACT, with computing requirements set at 55GFLOPS, compared to Catalyzer, RainbowCake, and RRZP. We can find that as the computing requirement increases, our algorithm becomes increasingly superior to other algorithms. When the computing requirement increases, the execution time becomes shorter, and model loading time increasingly dominates the overall ACT, as shown in Fig. 4(b). Our algorithm, focusing specifically on the overall ACT reduction, incorporates model lazy loading with layer-wise caching, which can effectively address the bottleneck posed by model loading time. This explains that although the proportion of execution time in our method is higher than other comparison algorithms, it significantly reduces the overall ACT, as shown in Fig. 4 (b).

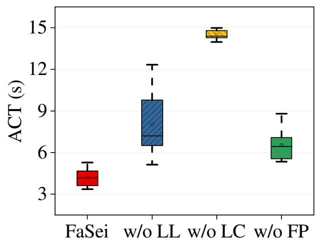
**The impact of the number of applications:** Next, we present the impact of the number of applications by varying it from 2 to 20. The results are reported in Fig. 3(c), we can find that FaSei always outperforms other algorithms. Compared to Catalyzer, RainbowCake, and RRZP, the proposed FaSei achieves an average overall ACT reduction of $5.3\times$, $4.47\times$, and $5.82\times$, respectively. In addition, we observe that as the number of applications increases, the acceleration ratio of FaSei decreases gradually compared to other algorithms. This is because the amount of memory available to each application for caching containers gradually decreases as the number of applications increases. Thus, the cold start time of each algorithm grows as shown in Fig.4(c). But even at 20 applications, our algorithm can still achieve 25%, 6%, and 33% reduction in the overall ACT. This shows that our algorithm has good scalability and remains effective when the number of applications increases.

**The impact of the number of servers:** Finally, we evaluate the impact of the number of edge servers on ATC by varying it from 2 to 10. As shown in Fig. 3(d), we can find that the proposed FaSei significantly outperforms Catalyzer, RainbowCake, and RRZP, achieving $4.44\times$, $3.76\times$, and $4.88\times$ speedups on average in overall ATC reduction, respectively. As the number of edge servers increases, the overall ACT gradually decreases. Notably, FaSei's ACT shows a more significant decline compared to other algorithms. This is because FaSei employs layer-wise model caching, a fine-grained caching strategy that effectively utilizes edge server cache resources, which can significantly reduce the cold start time, as shown in Fig. 4(d). In contrast, the compared algorithms employ coarse-grained caching methods that do not fully utilize the available cache resources. Even with an increased number of edge servers, the limited cache resources on each server prevent the full caching of containers or models, resulting in underutilization of cache resources. Therefore, the cold start time of the algorithms Catalyzer, RainbowCake, and RRZP remains nearly unchanged as the number of servers increases, as shown in Fig. 4(d). This experimental result demonstrates that FaSei can effectively reduce the overall ACT when the memory resources of a single edge server are limited in the SEC.

*2) Component analysis:* FaSei employs three components to reduce the overall ACT: lazy loading (LL), layer-wise caching (LC), and function placement (FP). We evaluate their individual contributions by measuring the increase in overall ACT after disabling each component. In particular, LL is disabled by starting the inference after all model layers are loaded; LC is disabled by removing container caching. FP is disabled by randomly placing functions on edge servers. We evaluate the individual contributions of the three components under scenarios with 5 and 10 applications while fixing other settings to their default values.



(a) A scenario with 5 applications



(b) A scenario with 10 applications

Fig. 5. Component analysis of FaSei. LL: *lazy loading*, LC: *layer-wise caching*, and FP: *function placement*

As shown in Fig. 5, when we disable LL, LC, and FP in the scenario with 5 applications, the overall ACT increases to $3.16\times$, $6.7\times$, and $2.59\times$, respectively. In the scenario with 10 applications, the overall ACT increases to $1.9\times$, $3.4\times$, and $1.55\times$, respectively. This result verifies the effectiveness of the proposed LL, LC, and FP. Disabling LC results in the most significant and stable performance degradation. This is because cold start time dominates in DL-driven applications. We can see from Fig. 5(a) that disabling LL, LC, and FP leads to greater performance degradation compared to Fig. 5(b). This stems from the intrinsic efficiency of the proposed FaSei, where the its advantages become more pronounced under lower system load conditions.

## V. Related Work

Many existing studies have focused on the cold start problem. Some approaches mitigate the cold start problem through full container caching or keeping the container alive [7], [16]–[20]. A fully initialized container will occupy hundreds or thousands of MBs in memory due to the large DL models. However, due to the limited resources of the SEC, it is not feasible to cache all containers in edge servers. Further, other approaches employ Partial caching or container-sharing techniques [22]–[25], [37] to reduce the cold start time with a smaller memory footprint. Part of them [22], [23], [25] use the idea of Zygote to pre-import a portion of common container packages. Before executing a function, the remaining necessary packages are loaded from scratch to avoid the time-consuming cold startup process. Pagurus [24] comprises an intra-function manager for replacing an idle warm container with a container that other functions can use to reduce cold start time. RainbowCake [37] proposes a layer-wise sharing-aware algorithm to mitigate cold starts by proactively pre-warm and adaptively keep-alive layer-wise containers. The core idea of container-sharing approaches is to reduce the cold start time to some extent by sharing common packages at the first or second stages of cold start. However, these methods inevitably require the user-specified model file to be fully loaded before execution, generating a long model loading time.

Only a few works [15], [26] focus on the issue of long cold start-up time caused by loading DL models. These approaches propose model tensor sharing [26] and model transformation [15] techniques, reusing duplicated tensors or similar model structures to reduce the cold start time associated with model loading. However, not all models have reusable tensors and similar model structures. When non-reuse models are invoked, these methods still do not reduce the cold start time. Unlike existing approaches, we focus on consistently reducing model loading time. To achieve this, we propose a method that combines model lazy loading and layer-wise caching, reducing ACT in resource-limited SEC environments.

## VI. Conclusion

In this paper, we propose FaSei, a fast serverless edge inference method with synergistic model lazy loading and layer-wise caching, which can efficiently mitigate the cold start time caused by the time-consuming model loading process of DL-driven applications in SEC. FaSei jointly optimizes model lazy loading, layer-wise caching, and function placement to minimize ACT. The joint optimization problem is formulated as an INLP problem, which is proven to be an NP-hard problem. We first convert the INLP problem into an ILP problem. Then we design an approximation algorithm and prove its theoretical performance guarantee. Finally, we conduct extensive experiments to evaluate the performance of the proposed FaSei. The results demonstrate that FaSei achieves up to a $6.7\times$ speedup in ACT reduction.

## Acknowledgment

The authors also thank the Big Data Computing Center of Southeast University for providing the experiment environment and computing facility.

## REFERENCES

[1] H. Wu, Y. Yu, J. Deng, S. Ibrahim, S. Wu, H. Fan, Z. Cheng, and H. Jin, "Streambox: A lightweight GPU sandbox for serverless inference workflow," in *Proc. of USENIX ATC*, 2024.

[2] Y. Fu, L. Xue, Y. Huang, A. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Low-latency serverless inference for large language models," in *Proc. of USENIX OSDI*, 2024.

[3] K. Zhao, Z. Zhou, L. Jiao, S. Cai, F. Xu, and X. Chen, "Taming serverless cold start of cloud model inference with edge computing," *IEEE Transactions on Mobile Computing*, vol. 23, no. 8, pp. 8111–8128, 2024.

[4] F. Ttncolu and G. Dn, "Joint resource management and pricing for task offloading in serverless edge computing," *IEEE Transactions on Mobile Computing*, vol. 23, no. 6, pp. 7438–7452, 2024.

[5] F. Ttncolu, S. Joilo, and G. Dn, "Online learning for rate-adaptive task offloading under latency constraints in serverless edge computing," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 695–709, 2023.

[6] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *Proc. of ACM ACSW*, 2021.

[7] L. Pan, L. Wang, S. Chen, and F. Liu, "Retention-aware container caching for serverless edge computing," in *Proc. of IEEE INFOCOM*, 2022.

[8] X. Shang, Y. Mao, Y. Liu, Y. Huang, Z. Liu, and Y. Yang, "Online container scheduling for data-intensive applications in serverless edge computing," in *Proc. of IEEE INFOCOM*, 2023.

[9] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "*FaaSLight* : General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–29, 2023.

[10] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proc. of USENIX ATC*, 2018.

[11] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proc. of ACM ASPLOS*, 2021.

[12] X. Cai, Q. Sang, C. Hu, Y. Gong, K. Suo, X. Zhou, and D. Cheng, "Incendio: Priority-based scheduling for alleviating cold start in serverless computing," *IEEE Transactions on Computers*, vol. 73, no. 7, pp. 1780–1794, 2024.

[13] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. che Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *ArXiv*, vol. abs/1902.03383, 2019.

[14] Z. Zhou, Y. Zhang, and C. Delimitrou, "Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. of ACM ASPLOS*. ACM, 2023.

[15] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, R. Wattenhofer, and Y. Yu, "Optimus: Warming serverless ml inference via inter-function model transformation," in *Proc. of ACM EuroSys*, 2024.

[16] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in *Proc. of ACM SoCC*, 2021.

[17] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proc. of ACM ASPLOS*, 2022.

[18] M. Shahrad, R. Fonseca, . Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. of USENIX ATC*, 2020.

[19] Cold starts in aws lambda. [Online]. Available: https://mikhail.io/serverless/coldstarts/ aws/

[20] Cold starts in azure functions. [Online]. Available: https://mikhail.io/serverless/ coldstarts/azure/

[21] K. Xiao, S. Yang, F. Li, L. Zhu, X. Chen, and X. Fu, "Making serverless not so cold in edge clouds: A cost-effective online approach," *IEEE Transactions on Mobile Computing*, pp. 1–14, 2024.

[22] Y. Li, D. Zeng, L. Gu, M. Ou, and Q. Chen, "On efficient zygote container planning toward fast function startup in serverless edge cloud," in *Proc. of IEEE INFOCOM*, 2023.

[23] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *Proc. of USENIX ATC*, 2018.

[24] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. of USENIX ATC*, 2022.

[25] Y. Li, D. Zeng, L. Gu, M. Ou, Q. L. Y. Chen, L. Gu, Z. Qu, L. Tian, and D. Zeng, "On efficient zygote container planning and task scheduling for edge native application acceleration," in *Proc. of IEEE INFOCOM*, 2024.

[26] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *Proc. of USENIX ATC*, 2022.

[27] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: Synergistic progressive inference of neural networks over device and cloud," in *Proc. of ACM MobiCom*, 2020.

[28] F. Dong, H. Wang, D. Shen, Z. Huang, Q. He, J. Zhang, L. Wen, and T. Zhang, "Multi-exit dnn inference acceleration based on multi-dimensional optimization for edge intelligence," *IEEE Transactions on Mobile Computing*, vol. 22, no. 9, pp. 5389–5405, 2023.

[29] Q. Pei, Y. Yuan, H. Hu, Q. Chen, and F. Liu, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. of ACM SoCC*, 2023.

[30] Z. Huang, F. Dong, D. Shen, J. Zhang, H. Wang, G. Cai, and Q. He, "Enabling low latency edge intelligence based on multi-exit dnns in the wild," in *Proc. of IEEE ICDCS*, 2021, pp. 729–739.

[31] X. Guo, F. Dong, D. Shen, Z. Huang, and J. Zhang, "Resource-efficient dnn inference with early exiting in serverless edge computing," *IEEE Transactions on Mobile Computing*, pp. 1–17, 2024.

[32] Z. Xu, L. Zhou, W. Liang, Q. Xia, W. Xu, W. Ren, H. Ren, and P. Zhou, "Stateful serverless application placement in mec with function and state dependencies," *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 2701–2716, 2023.

[33] Gurobi optimization. [Online]. Available: https://www.gurobi.com/

[34] Google or-tools. [Online]. Available: https://developers.google.cn/optimization/reference/

[35] M. Mitzenmacher and E. Upfal., "Probability and computing: Randomized algorithms and probabilistic analysis." *Bulletin of Symbolic Logic*, vol. 12, no. 2, pp. 304–308, 2006.

[36] The weaver computer engineering research group. [Online]. Available: https://github.com/deater/performance_results

[37] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proc. of ACM ASPLOS*, 2024.

[38] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proc. of ACM ASPLOS*, 2020.