Rocket: Warming Serverless Inference via Hierarchical ML Artifact Pre-loading and Sharing

Xiaofei Yue†, Song Yang†, Fan Li†, Youqi Li†, Yu Wang‡
† School of Computer Science and Technology, Beijing Institute of Technology, China
‡ Department of Computer and Information Sciences, Temple University, USA
Email:{xfyue, S.Yang, fli, liyouqi}@bit.edu.cn, wangyu@temple.edu

***Abstract*—Serverless computing is a promising method to serve Machine Learning (ML) inference via on-demand functions. Due to the time- and memory-consuming ML library and model (*i.e.*, ML artifact) loading, serverless inference endures notable startup overhead and memory waste issues. In this paper, we advocate for *hierarchical ML artifact pre-loading and sharing* to balance loading and memory efficiency. Building on this, we propose Rocket, a serverless ML inference system that accelerates function startup while reducing memory waste. Rocket dynamically pre-loads partial, shared ML artifacts, each implying a hierarchy of trade-offs between the loading latency and memory usage. Specifically, with a dual-timescale invocation prediction, Rocket first estimates the pre-loading timing for each function, and then schedules them via a sharing-aware agglomerative clustering to improve ML artifact sharing efficiency. In particular, Rocket learns to make the online hierarchical pre-loading decision for function containers based on a lightweight contextual bandit algorithm. Finally, we implement Rocket and evaluate it with realistic workloads. Experimental results display that Rocket outperforms existing solutions by up to 38.7% on startup latency and up to 43.8% on memory saving.**

I. Introduction

The flourishing of Machine Learning (ML) has propelled a substantial surge in resource demands [1], especially for user-facing inference workloads. Facebook, for example, serves up to 200 trillion inference requests daily [2]. Given the dynamic requests at scale, traditional server-driven inference platforms [3], [4], however, not only entail fussy management, but also hold resource inefficiency. As a revolutionary cloud paradigm, serverless computing [5] fits well with ML inference due to its ease of use, agile scalability and cost efficiency [6]. Coupled with the booming of Large Language Models (LLM) [7], [8], many inference offerings (*e.g.*, AWS SageMaker [9] and Azure ACI [10]) have shifted toward serverless architectures.

The serving of serverless inference proceeds via functions, each of which runs in a container hosting a pre-trained model. These functions are invoked on-demand, but require a lengthy *cold-start* [5] before formal inference, which includes: (1) container warming (*i.e.*, creation and initialization), (2) ML library (*e.g.*, PyTorch) importing and (3) model loading, as illustrated in Fig. 1. Despite recent advances in cold-start mitigation via container keep-alive [11], [12] or pre-warm [13], [14] policies,

Song Yang is the corresponding author.

This work is partially supported by the National Natural Science Foundation of China (No. 62472028, No. 62172038, and No. 62372045), the Beijing Natural Science Foundation (No. 4232033 and No. L254058), the National Key R&D Program of China (No. 2023YFB3107300), and the Beijing Institute of Technology Research Fund Program for Young Scholars.

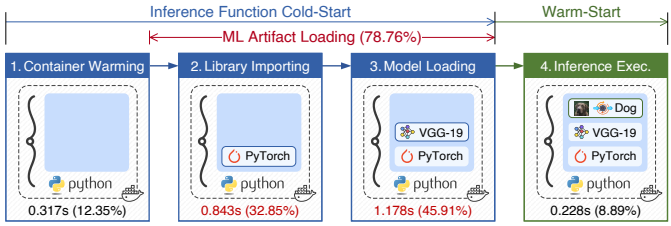


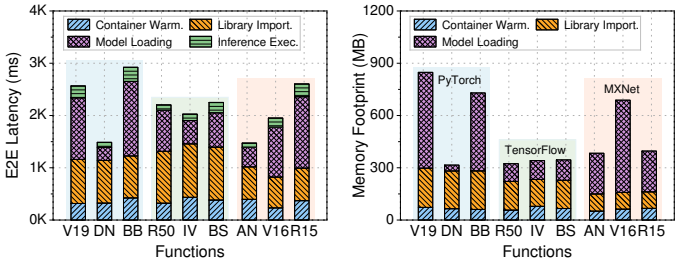
Fig. 1. The running timeline of a serverless ML inference function.

they gain little for inference tasks. Concretely, ML models and libraries (*i.e.*, ML artifacts [15]) are typically memory-hungry. Keeping ML artifacts alive with containers together will incur large memory waste, as 87% of functions are invoked at most once per minute [5]. Alternatively, pre-warming the container alone ignores the *ML artifact loading*[1], a process unique to inference functions that has become the primary bottleneck. For instance, this process accounts for up to 78.76% of the total latency in Fig. 1. In summary, it is essential to accelerate ML artifact loading while reducing memory waste.

In serverless inference, recent efforts have sought to address similar issues using *(1) partial model pre-loading and sharing*, which pre-loads common model tensors [16], [17] or operators [18], and share them across different functions. This partially speeds up model loading with low memory cost, and such potential sharing further amplifies the benefits while facilitating coping with request burstiness [19]. Nevertheless, the notable library importing latency is largely ignored. Therefore, *(2) ML artifact pre-loading* [15] is proposed to pre-load entire ML artifacts after container warming, thereby maximizing end-to-end loading efficiency. However, more duplicate ML artifacts are forced to load for extended periods, leading to poor memory efficiency [20]. In our view, these two approaches are complementary, with the potential to mitigate their respective drawbacks in ML artifact loading and memory waste.

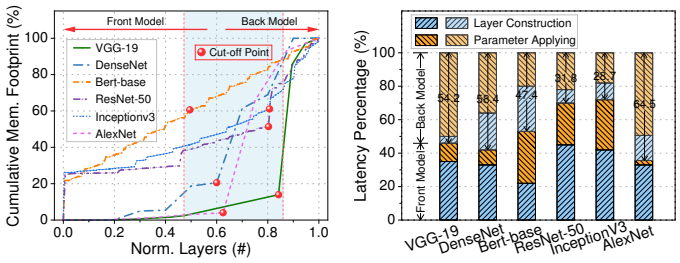
In this paper, we advocate for *hierarchical ML artifact pre-loading and sharing*. That is, we dynamically pre-load partial ML artifacts, each implying a hierarchy of trade-offs between the loading latency and memory waste, into warm containers. Also, these containers hold the potential to be shared across functions to amplify gains. Nonetheless, there are three challenges to achieve this goal. *First*, given the memory-hungry ML artifacts, we need to judiciously decide the timing of pre-

[1]Similar to [15], the library importing and model loading are jointly referred to as the ML artifact loading (*i.e.*, steps 2&3 in Fig. 1) in this paper.



(a) E2E Latency. (b) Memory footprint.

Fig. 2. The (a) End-to-End (E2E) latency and the (b) memory footprint breakdowns of nine functions (see Section VI-A) with varying ML artifacts.



(a) "Cut-off" points. (b) Loading latency.

Fig. 3. The (a) performance "cut-off" points and the (b) corresponding loading latency breakdowns of six typical models.

loading. Yet, function requests are highly dynamic and bursty [5], [21], making it hard for fine-grained invocation prediction. *Second*, once mis-predicted due to the bursty invocations, a full cold-start occurs if no shared container is located. To alleviate this, it is essential to adjust function distribution across servers timely to improve the sharing efficiency. *Third*, the hierarchical pre-loading proceeds accompanied by dynamic invocation and sharing behaviors [12]. The ensuing uncertainty makes it challenging to identify the impact of pre-loading decisions.

To meet these challenges, we present Rocket, a hierarchical ML artifact pre-loading and sharing system for serverless inference. It treats each container as a *state machine*, and enables state transitions by pre-loading partial ML artifacts, to speed up function startup while reducing memory waste. This is because each state exhibits unique advantages in loading, memory and sharing. More specifically, Rocket granularly estimates future invocations via Dual-scale Invocation Prediction (DIP), which captures periodicity at a large timescale and burstiness at a small timescale. Then, Rocket proposes a sharing-aware function grouping and scheduling algorithm to co-locate functions with high degrees of sharing using agglomerative clustering [22]. Finally, Rocket customizes contextual bandit [23], a lightweight online algorithm, to make hierarchical pre-loading decisions without unreliable offline profiling.

To sum up, this paper makes the following contributions:

- We identify various trade-offs between memory and loading efficiency, and strike them via hierarchical pre-loading and sharing to adapt to bursty invocation patterns.
- We design Rocket, which learns efficient pre-loading decisions online atop the predicted invocation pattern of DIP and sharing-aware function scheduling by group.
- We implement a prototype of Rocket, which outperforms the state-of-the-art solutions in function startup latency and memory saving via extensive experiments.

II. Background and Motivation

A. Serverless Inference

In the cloud-native era, serverless computing offers a compelling way to serve ML inference, known as *serverless inference* [3], [24]. A case study [25] has verified its advantages in performance and cost. In serverless inference, each pre-trained model is packaged into a function, with the system seamlessly handling its resource provisioning. The function is capable of auto-scaling by quickly spawning instances to serve online requests timely, with billing based on the actual resource usage. Nonetheless, this event-driven pattern inevitably incurs a cold-start prior to formal inference, which prepares a container and loads the required ML artifact [15]. As depicted in Fig. 1, the inference cold-start consists of the following three steps:

(1) Container warming. Upon receiving an inference request, a container is instantiated from the built-in Python image, and initializes its runtime environment. Next, the container unzips the user package, including the inference code, model file and required ML library (*e.g.*, PyTorch).

(2) ML library importing. After the container warming, its function process is spawned to execute the inference code. To provide the necessary ML runtime, the ML library must first be loaded into memory from disk by executing the corresponding import statement (*e.g.*, "import torch").

(3) Model loading. After the library importing, the function process loads the pre-trained model by reading and deserializing the local model files (*e.g.*, with "*.pt*" format for PyTorch) saved with the model structure and parameters.

B. Observations

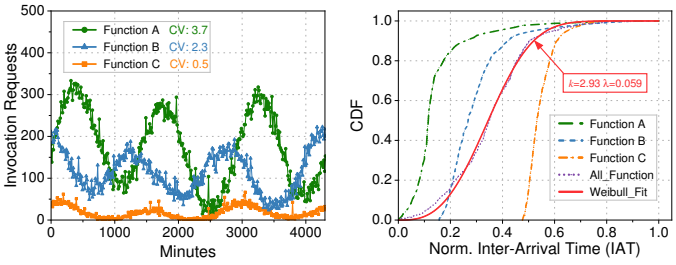
To identify the improvement opportunity of inference cold-starts, we conduct a preliminary exploration based on popular workloads (see Section VI-A for details).

Observation 1: *The ML artifact loading dominates the cold-start latency and memory footprint of inference functions.*

As shown in Fig. 2(a), ML library importing, not just model loading, accounts for 26.4%–58.4% of the cold-start latency, which is largely ignored by recent works [26], [18]. These two steps jointly dominate up to 78.8% of the total latency. In contrast, the container warming has a smaller share. This is because the ML artifact loading is I/O- and CPU-intensive, which involves retrieving and parsing many dynamic linking modules for ML libraries [27] and parameters in model files. These artifacts are maintained as inference runtime, leading to a notable memory footprint, as shown in Fig. 2(b). Overall, pre-warming container gains little for cold-start mitigation.

Observation 2: *ML models exhibit similar performance cut-off points, which facilitate partial pre-loading and sharing.*

Delving into ML models, Fig. 3(a) reveals that several top layers occupy vast memory. The cumulative usage rises sharply beyond a certain "*cut-off*" point, as these layers are high-dimensional with numerous parameters for abstract feature integration. With this point, we split each model into two slices:



(a) Invocation requests. (b) IAT distribution.

Fig. 4. A three-day (a) request trace and (b) IAT distribution of three typical serverless workloads in Azure Functions Trace [5].

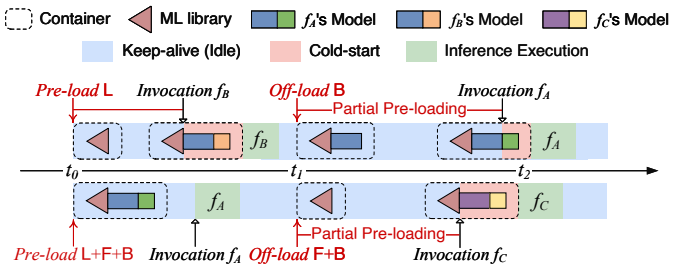


Fig. 5. An example of our hierarchical ML artifact pre-loading and sharing via three inference functions, f_A, f_B, and f_C. Specifically, they adopt the same ML library, and f_A and f_B hold variants sharing the same base model.

front model and *back model*. As shown in Fig. 3(b), however, the loading latency of heavier back models, with fewer layers, grows *sub-linearly w.r.t. parameter size*. This is because layer construction is time-consuming, whereas parameter applying involves fast in-memory mapping. This prompts us to only pre-load the front model, which is lighter but saves a comparable loading latency. An analysis [16] shows that lots of production models are drawn by pre-training or transfer learning for low training overhead while addressing data scarcity. That is, model variants share a backbone and differ in a few task-specific top layers. Even for models with less pronounced phenomena (*e.g.*, Bert [28]), the partial (*i.e.*, front model) pre-loading can still benefit from the sharing of model variants.

Observation 3: *Invocation requests of a function are bursty, but aggregated requests are relatively predictable.*

The Azure Trace [5] is widely-used for inference workloads [15], [16]. We select three functions with typical (*low, medium, high*) request rates, and plot them in Fig. 4(a). As shown, they reveal varying degrees of diurnal *periodicity* and are highly *bursty* in the short term. The burstiness is related to request rates, as shown by the Coefficient of Variation (CV) of Inter-Arrival Time (IAT) being 0.5, 2.3, and 3.7, respectively. In contrast, the arrival of aggregated requests is relatively stable across *all functions*, with a CV of only 1.4. Delving into the overall IAT, we find it can be approximated by a Weibull distribution [29], as shown in Fig. 4(b). Also, 87% of function requests are sparse [5], arriving at most once per minute. Thus, the aggregated requests tend to be much more predictable.

C. Hierarchical ML Artifact Pre-loading and Sharing

Due to the varied latency and memory footprints of the three steps, and the existence of model cut-off points **(Observation 1** and **Observation 2)**, there are different hierarchies of trade-offs between minimizing ML artifact loading latency and reducing memory waste. Moreover, these trade-offs should be flexible to transition, so as to accommodate the uncertainty implied in the dynamic and bursty invocation pattern **(Observation 3)**.

Key insight. We advocate for *hierarchical ML artifact pre-loading and sharing* to balance loading and memory efficiency. Concretely, we identify three key objects: ML library (L), front model (F) and back model (B), each mapping to a hierarchy of those trade-offs. Inspired by layer-wise containers in RainbowCake [12], we further treat a warm container as a state machine and flexibly *pre-load* (*off-load*) these objects for proactive state transition, instead of passive caching. This is because keeping ML artifacts alive is costly. To highlight the merits of our insight, we present an illustrative example in Fig. 5. Originally, two containers of function f_A have been reserved at time t_0, and we pre-load only L into the first container (Fig. 5 (top)). Then, it is shared to be compatible with and serve an unexpected invocation of f_B, which incurs a *partial cold-start* (*i.e.*, loading its model). At t_1, we off-load B from this container for partial pre-loading, facilitating reducing memory waste and sharing with a subsequent invocation of f_A. For another (Fig. 5 (bottom)), we pre-load the entire ML artifact at t_0, as a f_A's invocation arrives soon. Similarly, we off-load its model at t_1, without affecting its hit by an invocation of f_C. By doing so, we can switch among those trade-offs for long-term benefits.

III. System Overview

A. Overview

Rocket is a novel serverless inference system that achieves hierarchical ML artifact pre-loading and sharing. It is designed to speed up inference function startup with reduced memory waste. The high-level overview of Rocket is depicted in Fig. 6. The client issues inference requests specifying the input data and a model pre-trained from a type of base model. They are translated into invocations of the functions hosting the corresponding models. The goal of Rocket is to locate an available or compatible container to serve each of them.

Specifically, the ❶ Dual-scale Invocation Predictor retrieves recent request logs to build function-specific time series prediction models at a large timescale and group-specific analytical models at a small timescale (Section IV-A). Based on this, it estimates each function's invocation concurrency, which facilitates the proactive container pre-warming and the further pre-loading. Next, the ❷ Sharing-aware Function Scheduler periodically groups the functions with high degrees of sharing via agglomerative clustering, and greedily schedules pre-warmed containers by group to enhance ML artifact sharing efficiency (Section IV-B). The ❸ Online Hierarchical Pre-loader makes the ML artifact pre-loading decisions by managing the respective states (Section III-B) of reserved, idle containers (either in keep-alive or pre-warmed) for each group. It proceeds with a lightweight contextual bandit algorithm, which learns online with the objective of minimizing the unified startup and memory costs (Section IV-C). Following the pre-loading decision, finally, the ❹ Container Manager pre-loads the corresponding

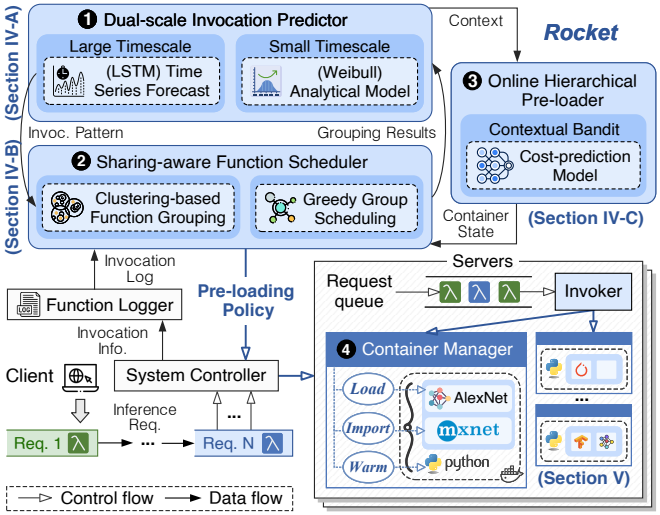


Fig. 6. The system overview of Rocket.

objects to each container, transitioning them to the designated states while enabling the sharing across functions (Section V).

B. Container Lifecycle and States

In Rocket, each idle container is considered as a finite state machine. Specifically, lower states imply greater compatibility and fewer memory footprints, but incur higher startup latency. The opposite holds for higher states. These states, ordered *from low to high*, are listed as follows:

- In the Warming state, a container has been fully instantiated from the built-in Python image, providing a base language runtime for general inference functions.
- In the Imported state, the Warming container has spawned and started its function process to import a specific *ML library*, which further offers an active ML runtime for any function using this ML library.
- In the P-loaded state, the Imported container has driven the function process to partially load a specific *front model*, which can be shared across functions that use model variants pre-trained from the same base model.
- In the F-loaded state, the P-loaded container has further loaded a corresponding *back model* to make up the full model in memory, which can immediately handle an invocation of only one inference function.

Fig. 7 depicts the lifecycle of a container spanning the above four states. This container can transition among adjacent states by pre-loading or offloading the corresponding object (*i.e.*, ML library, front model, or back model).

IV. System Design

A. Dual-scale Invocation Predictor

For efficient container pre-warming and further pre-loading, it is essential to judge whether a function will be invoked, and its invocation concurrency [13], within a time interval. Given the long-term diurnal periodicity and short-term burstiness of inference requests, we propose Dual-scale Invocation Prediction (DIP), which captures each function's invocation pattern by combining large and small timescale predictions.

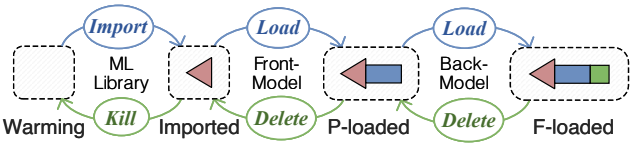


Fig. 7. The state transition process of an idle container in Rocket.

At a large-timescale, DIP uses time series forecasting (*i.e.*, LSTM [14]) to predict the invocation pattern for each function by learning from the diurnal periodicity of their requests (see Fig. 4(a)). For accuracy, we need to divide a series of coarse-grained time windows (*e.g.*, 1 minute), as in prior works [5], [14]. Based on the number of invocations from the past t time windows $\{Q_{i,1}, Q_{i,2}, \ldots, Q_{i,t}\}$ for function f_i, DIP estimates the expected quantity $\hat{Q}_{i,t+1}$ in the next window. To prevent under-estimation, we adopt a conservative, classification-based manner for the LSTM, instead of predicting a scalar value directly. The prediction space is discretized into segments, with the upper bound of each segment as the ground truth.

Nevertheless, such a large timescale prediction is not enough. Each invocation arrives at any time due to the dynamic IAT [13] (*i.e.*, burstiness). Warming $\hat{Q}_{i,t+1}$ containers at the start of time window $t+1$ is premature for some invocations. Despite the acceptability for most functions [14], the gains of reduced cold-start latency are easily negated by wasted resource costs, when applied to memory-hungry inference workloads.

At a small-timescale, thus, we refine each time window into T fine-grained timesteps (*e.g.*, 10s). Given the relatively stable IAT of aggregated requests (see Fig. 4(b)), DIP first estimates the next timestep's concurrency for function groups (the grouping algorithm is stated in Section IV-B). Given a group $g = \{f_1, f_2, \ldots\}$, DIP fits a Weibull distribution [29] to its IAT records for the latest T timesteps, *i.e.*, $\text{IAT}_g \sim \text{Weibull}(k_g, \lambda_g)$, where k_g and λ_g are the shape and scale parameters, respectively. Then, invocation arrivals for timestep $T+1$ can be simulated based on this Weibull distribution. Specifically, each drawn IAT sample is assigned to function $f_i \in g$ with probability p_i, until the cumulative IAT exceeds the timestep length. Note that p_i is approximated as the estimated share of invocations for f_i in the *belonging time window*. For example, $p_i = \frac{\hat{Q}_{i,t'}}{\sum_{f_j \in g} \hat{Q}_{j,t'}}$ if timestep $T+1$ falls within time window t'. Finally, DIP estimates the concurrency $\hat{q}_{i,T+1}$ of function f_i at timestep $T+1$ from the sampled IATs by aggregating them into a short-term peak workload estimate.

B. Sharing-aware Function Scheduler

The function distribution across servers has a vital influence for Rocket. For instance, if two co-located functions adopt the same ML library and even base model, they have more sharing opportunities. Otherwise, their invocations could be routed to search for compatible containers, resulting in poor efficiency. Despite the high similarity in ML artifacts, it is still non-trivial to find idle shared containers once their invocation patterns are also similar [19]. Given this, Rocket proposes a *sharing-aware grouping and scheduling algorithm* that groups functions and schedules their containers to improve sharing gains.

Algorithm 1: Clustering-based function grouping.

Input : Inference functions, $\{f_1, f_2, \ldots\}$;
Function DoS matrix, $\{DoS(f_i, f_j)\}_{\forall f_i \neq f_j}$;
Output: Function sharing dendrogram, $\mathcal{D}$;

// Executes every T timesteps
1 **Initialize**: $\mathcal{G} \leftarrow \{\{f_1\}, \{f_2\}, \ldots\}, \mathcal{D} \leftarrow \{\}$;
2 **foreach** group pair $\langle g_i, g_j \rangle \in \mathcal{G}$ **do**
3 $\quad DoS[g_i, g_j] \leftarrow DoS(f_i, f_j)$;
4 **while** $|\mathcal{G}| > 1$ **do**
$\quad$ // Merge groups with the highest DoS
5 $\quad (i', j') \leftarrow \arg\max_{i \neq j} DoS[g_i, g_j]$;
6 $\quad g_k \leftarrow g_{i'} \cup g_{j'}$;
7 $\quad$ Remove $g_{i'}$ and $g_{j'}$ from $\mathcal{G}$, add g_k to $\mathcal{G}$;
8 $\quad \mathcal{D}$.append $(g_{i'}, g_{j'}, DoS[g_{i'}, g_{j'}])$;
$\quad$ // Update DoS for the merged group g_k
9 $\quad$ **foreach** group $g \in \mathcal{G}$ **and** $g \neq g_k$ **do**
10 $\quad\quad DoS[g_k, g] \leftarrow \frac{1}{|g_k||g|} \sum_{f_i \in g_k, f_j \in g} DoS(f_i, f_j)$;
11 **return** $\mathcal{D}$ // Find the final dendrogram $\mathcal{D}$

1) Clustering-based Function Grouping: Rocket jointly considers two metrics: (*i*) *ML artifact similarity* and (*ii*) *invocation complementarity*, to group functions in the cluster. First, the ML artifact similarity $\mathcal{S}(f_i, f_j)$ of two inference functions f_i and f_j is expressed as:

$$\mathcal{S}(f_i, f_j) = \begin{cases} 0, & lib_i \neq lib_j \\ 0.5, & lib_i = lib_j \wedge m_i \neq m_j \\ 0.75, & lib_i = lib_j \wedge m_i = m_j \end{cases}, \quad (1)$$

where lib_i and m_i represent the ML library and base model of f_i, respectively. The rationale is that function containers with the same ML library are shareable in 2 out of the 4 states. This proportion rises to 75% if their base models are also the same. Moreover, based on the historical concurrency $\{q_{i,t}\}_{t \in T}$ and $\{q_{j,t}\}_{t \in T}$ over the most recent T timesteps, we can calculate their invocation complementarity with $\mathcal{C}(f_i, f_j) = (1 - r)/2$. Here, r denotes the Pearson correlation coefficient, given by:

$$r(f_i, f_j) = \frac{\sum_{t=1}^{T} (q_{i,t} - \bar{q_i})(q_{j,t} - \bar{q_j})}{\sqrt{\sum_{t=1}^{T} (q_{i,t} - \bar{q_i})^2} \cdot \sqrt{\sum_{t=1}^{T} (q_{j,t} - \bar{q_j})^2}}, \quad (2)$$

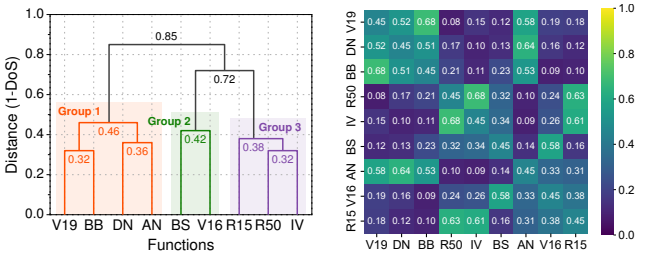
where $\bar{q_i}$ and $\bar{q_j}$ are the averages of $\{q_{i,t}\}$ and $\{q_{j,t}\}$, respectively. Then, we define the *Degree of Sharing* (DoS) of the two functions as $\alpha\mathcal{S}(f_i, f_j) + (1 - \alpha)\mathcal{C}(f_i, f_j)$, where $\alpha \in [0, 1]$ is a tunable weight that balances the two metrics.

Based upon DoS, Algorithm 1 tends to group functions with similar ML artifacts and distinct invocation patterns via *agglomerative clustering* [22]. The higher DoS of two functions, the closer in their distance. First, each function is initialized as a separate group, with the DoS matrix being the same as that of the corresponding function (lines 1–3). Next, we iteratively merge two groups with the highest DoS (lines 5–7) to construct the *dendrogram* (line 8), until only one group is left (line 4). After each merge, the DoS (*i.e.*, distance) between the merged group and each of the others is updated (lines 9–10). The DoS of two groups is the average DoS across all of their function pairs. We found that the number of groups remains constant due to the high contribution of ML libraries, despite variations in the functions within groups. Fig. 8 shows a dendrogram and the DoS over the nine functions in Section VI-A, and they are distinctly clustered into three groups. Here, the weight α is set to 0.6 with the largest inter-group distances.

Algorithm 2: Greedy function scheduling by group.

Input : Function groups, $\mathcal{G} = \{g_1, g_2, \ldots\}$;
Predicted function concurrency, $\{\hat{q}_{i,t}\}_{\forall f_i}$;
Output: Pre-warmed container scheduling of each function;

// Executes at each timestep t
1 **Initialize**: $\mathcal{A} \leftarrow \{\}$;
$\quad$ // Calculate the server affinity for each group
2 **foreach** server k in the cluster **do**
3 $\quad$ **foreach** function group $g_j \in \mathcal{G}$ **do**
4 $\quad\quad a_{j,k} \leftarrow \sum_{f_i \in g_j} f_i$.idleNumber$(k)$;
5 $\quad\quad \mathcal{A} \leftarrow \mathcal{A} \cup [g_j, k, a_{j,k}]$;
$\quad$ // Get the pre-warm plan of each function
6 **foreach** function f_i **do**
7 $\quad r_{i,t} \leftarrow$ Hierarchical_Pre_loader $(\hat{q}_{i,t})$;
8 $\quad r_{i,t} \leftarrow \max\left(r_{i,t} - \sum_k f_i\text{.idleNumber}(k), 0\right)$;
$\quad$ // Schedule functions by affinity
9 Sort $\mathcal{A}$ in descending order by $a_{j,k}$;
10 **foreach** affinity $a \in \mathcal{A}$ *and* function $f_i \in$ group $a[0]$ **do**
11 $\quad$ **while** $r_{i,t} > 0$ and isAvailableRes$(a[1])$ **do**
12 $\quad\quad$ Pre-warm a container of f_i on server $a[1]$;
13 $\quad\quad r_{i,t} \leftarrow r_{i,t} - 1$;



(a) Sharing dendrogram. (b) DoS between functions.

Fig. 8. The (a) grouping results and (b) DoS heatmap of 9 function workloads.

For each group g_j, we count the number of its idle containers (in keep-alive) on each server k as their *affinity* $a_{j,k}$ (lines 1–5). Then, we make *pre-warm plan* for each function based on the predicted concurrency of DIP (lines 6–8). Specifically, we decide the number of reserved containers $r_{i,t}$ for function f_i through the Hierarchical Pre-loader (see Section IV-C, and $r_{i,t} = \sum_{j>0} x_{i,j,t}$). Therefore, the number of containers to be pre-warmed is derived by subtracting the existing ones. Then, the group with the highest affinity is prioritized for scheduling on each server (lines 9–11). That is, we pre-warm this group's containers until the available resources are depleted or the pre-warm plan is fulfilled (lines 12–14). By doing so, the requests of each group can be scheduled among a few servers for load-balancing. This improves the likelihood of finding compatible containers, when no F-loaded container is available. We next will demonstrate how to determine the states of the reserved containers for efficient hierarchical pre-loading.

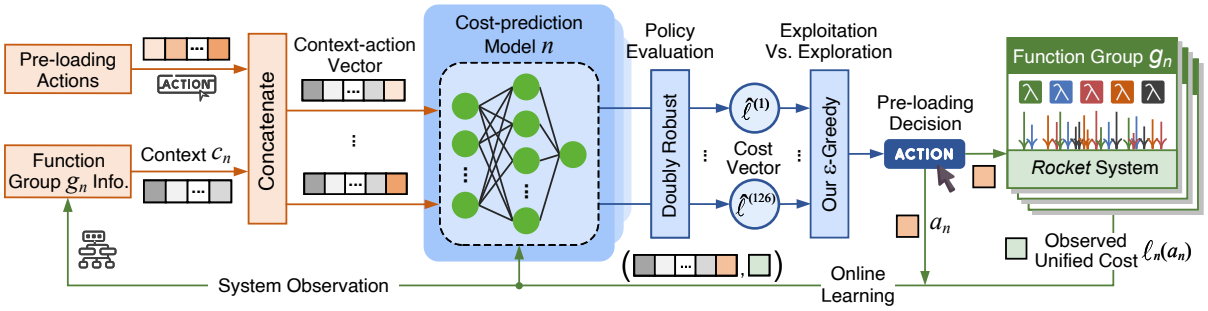


Fig. 9. The workflow of our contextual bandit algorithm.

C. Online Hierarchical Pre-loader

Based on the predicted invocation pattern, we can pre-warm enough containers proactively for each function to serve future invocations. Given the potential mis-estimation due to request burstiness, Rocket needs to make hierarchical pre-loading decisions by managing their states (see Fig. 7). The objective is to jointly minimize the (1) *startup overhead* (*i.e.*, the latency required to transition from the current state to F-loaded state), and (2) *memory waste* (*i.e.*, the cumulative memory consumption during idle periods) for these containers.

1) Problem Formulation: Suppose that a total of N inference functions are to be served by Rocket. At timestep t, each function $f_i, i \in [1, N]$ reserves at most $\hat{q}_{i,t}$ idle containers (*i.e.*, the predicted concurrency of DIP), each of which is either pre-warmed or kept alive. We need to decide their hierarchical pre-loading plan $x_{i,j,t}$ (*i.e.*, a variable that represents the number of containers in the j-th state for function f_i at t). Therefore, this problem can be formulated as:

$$\min_{X} \quad \sum_{t=1}^{\mathcal{T}} \sum_{i=1}^{N} \sum_{j=1}^{M} x_{i,j,t} (C_{i,j}^{start} + \gamma C_{i,j}^{mem}) \tag{3}$$

$$\sum\nolimits_{j} x_{i,j,t} \leq \hat{q}_{i,t}, \forall i \in [1, N], t \in \{1, \mathcal{T}\}, \tag{4}$$

$$x_{i,j,t} \in \mathbb{Z}^{+} \cup \{0\}, \forall i \in [1, N], j \in [1, M], t \in \{1, \mathcal{T}\}. \tag{5}$$

The objective (3) is to minimize the total *unified cost* of the reserved containers over $\mathcal{T}$ timesteps, where $C_{i,j}^{start}$ and $C_{i,j}^{mem}$ are the total startup overhead and memory waste, respectively, of a function f_i's container at the j-th state when handling all invocations. The weight $\gamma \in (0, 1]$ is a conversion factor while gauging the priority in minimizing memory cost. Constraint (4) ensures that the number of reserved containers does not exceed the concurrency with conservative estimates, and constraint (5) refers to the domain constraint.

If we use $j = 0$ to represent no reservation, then constraint (4) can be rearranged as $\sum_{j \in [0,M]} x_{i,j,t} = \hat{q}_{i,t}$. For a single timestep, the problem (3)–(5) is equivalent to an $(M+1)$-way integer $(\hat{q}_{i,t})$ partitioning problem, which is known to be NP-complete [30]. Across $\mathcal{T}$ timesteps, the *online invocation and sharing behavior* is highly uncertain, which renders it tough to explicitly quantify (3), especially during the bursty invocation periods. Thus, we next resort to Contextual Bandits (CB) [23], [31], a well-known lightweight online learning algorithm, to facilitate the hierarchical pre-loading for Rocket.

2) Contextual Bandit-based Pre-loading: Recall that Rocket aims to transition the DIP-reserved containers to designated states at the start of each timestep via hierarchical pre-loading. This time-stepped way to prediction and pre-loading simplifies the problem (3)–(5) as a *receding-horizon optimization*, as in prior works [13], [14], [32], which largely localizes the impact of an action to the current step [31]. In contrast to widely-used reinforcement learning [33], [34], the CB is more lightweight and better suitable for our problem. At the same time, it adapts naturally to the non-stationary invocation patterns using online updates with fewer real-time samples [23].

The sharing of ML artifacts yields inter-function dependencies, which are confined within groups (see Section IV-B). Due to the lower DoS and sharing efficiency between groups, they are considered mutually independent. As such, Rocket trains multiple CBs, each responsible for the hierarchical pre-loading of a disjoint set of functions (*i.e.*, a group). With the observed *context* of group g_n (*e.g.*, invocation concurrency), CB n takes an action (*e.g.*, $x_{i,j,t}$) at each timestep t. It then encounters an immediate cost (*i.e.*, negative reward) feedback by the chosen action, with the goal being to minimize cumulative cost (*e.g.*, objective (3)). This problem can be reduced to a cost-sensitive classification through *counterfactual estimates* [35], which has recently been witnessed to vast success in real-world systems [23], [31]. That is, the CB trains a *cost-prediction model* using standard supervised learning. Building on this, the workflow of our CB algorithm is shown in Fig. 9, which learns an action with the lowest cost for each group at each timestep's context.

Context. A naive method is to utilize the predicted concurrency of intra-group-g_n functions as the context of CB n. However, their impact has been implied in the reserved container number. Thus, we define the context $c_{n,t}$ as the mean Requests Per Second (RPS) of each function, coupled with its cold-start latency and memory breakdowns, given by,

$$c_{n,t} = \left\{ \text{latency}_i, \text{mem}_i, \frac{1}{\hat{q}_{i,t}} \sum\nolimits_{k} \text{RPS}_{i,t}^{(k)} \right\}_{\forall f_i \in \text{group } g_n}, \tag{6}$$

where $\text{RPS}_{i,t}^{(k)} = (\text{IAT}_{i,t}^{(k)})^{-1}, k \in [1, \hat{q}_{i,t}]$, is the k-th invocation's transient RPS for f_i, equal to the inverse of the corresponding IAT. If a function exhibits a large RPS, its containers may tend to be pre-loaded to a higher state, as it is more likely to reduce startup overhead with low memory waste.

Action. Considering intra-group dependencies, our problem becomes partitioning a set of integers $\{\hat{q}_{i,t}\}_{f_i \in g_n}$ into five dis-

tinguishable addends (*i.e.*, states) simultaneously at t. Despite the finite container states, CB n still needs to find the best action $a_{n,t}$ from $\prod_{f_i \in g_n} \binom{\hat{q}_{i,t}+4}{4}$ solutions. For example, even with 9 functions, each including only one container, yielding up to 1,953,125 choices. Coupled with the time-varying action space, it is infeasible for CBs to learn effectively.

To reduce and fix the action space, we divide the containers of intra-group-g_n functions evenly into K classes, and output an action (*i.e.*, container state) for each class. The invocation complementarity within g_n enables each function to maintain a distinct number of containers in different, fine-grained classes. Hence, this partitioned decision-making manner does not miss the optimal action. In this way, the action space is reduced to $\binom{K+4}{4} = 126$ (with $K = 5$ producing the lowest unified cost based on the sensitivity experiment in Section VI-B).

Cost function. The total unified cost of intra-group-g_n functions serves as the per-step cost of CB n. Given that each invocation may arrive at any time in t, leading to distinct $C^{mem}_{i,j}$ for f_i's containers, (3) cannot be directly used as the cost function by counting $x_{i,j,t}$. Hence, we calculate the memory cost of each container by measuring its idle time. If it is not hit, the idle time is the entire timestep. Also, the startup overhead remains relatively stable and can be easily profiled. Despite the existing prediction errors in DIP, the unified costs of containers undergoing a full cold-start are also incorporated as a penalty for the pre-loading decision in terms of sharing.

Exploitation and exploration. Each CB n learns online by updating its cost-prediction model on a collection of samples $\{c_{n,t}, a_{n,t}, \ell_n(a_{n,t})\}$, where $\ell_n(a_{n,t})$ is the observed cost of taking action $a_{n,t}$, serving as the label. With this model, CB n estimates a cost vector $(\hat{\ell}^{(1)}_{n,t+1}, \ldots, \hat{\ell}^{(k)}_{n,t+1}, \ldots, \hat{\ell}^{(126)}_{n,t+1}) \in \mathbb{R}^{126}$ based on the next step's context $c_{n,t+1}$, where $\hat{\ell}^{(k)}_{n,t+1}$ denotes the estimated cost of the k-th action, which further biased via *doubly robust policy evaluation* [35]. It then *exploits* the k^*-th action as $a_{n,t+1}$, where $k^* = \text{argmin}_k \hat{\ell}^{(k)}_{n,t+1}$, with a small probability of ϵ to *explore* a random action (*i.e.*, ϵ-greedy [23]), preventing getting stuck in a local optimum.

However, the performance drop due to improper exploration is irreversible for Rocket, potentially offsetting the long-term benefits of hierarchical pre-loading. Inspired by [31], we limit exploration to only the *neighbors of the best action*. For instance, if the best action is $(3, 2, 4, 0, 1)$, denoting the selected state orders for the five classes of containers, its neighbors include: $(4, 2, 4, \ldots)$, $(3, 1, 4, \ldots)$, $(3, 2, 3, \ldots)$, and so on. This is feasible since container states are transitioned in order (see Fig. 7). Such conservative exploration can avoid frequent ML artifact loading or prolonged memory idleness.

V. Implementation

Rocket is built atop OpenWhisk [36], an open-source serverless system, offering warm inference services with various ML runtimes. All components of Rocket, apart from the Container Manager, are co-located with OpenWhisk's Controller.

Dual-scale Predictor. The Predictor uses a sliding window to cache recent invocation information for Weibull distribution updates. Coupled with the well-trained LSTM models, it sends the predicted concurrency to the Function Scheduler and IATs to the Hierarchical Pre-loader as context.

Function Scheduler. The Scheduler gets historical invocation records and implements the agglomerative clustering via sklearn.cluster library [37]. The group scheduling plan, along with the container states of the Pre-loader, is asynchronously sent to the corresponding Invokers. We modify the Invoker to activate a specified number of containers via the standard pre-warming event. In addition, the container states are translated into pre-loading requests sent to the Container Manager.

Hierarchical Pre-loader. The Pre-loader implements CBs using the Vowpal Wabbit (VW) library [23]. Its built-in cost-prediction model is a lightweight *shallow neural network*. We set the number of hidden neurons and the parameter ϵ to 8 and 0.1, respectively. VW optimizes the model parameters with an adaptive learning rate initialized at 0.5. It offers *feature hashing* to handle our variable-length effectively. Finally, the output decisions are forwarded to the Scheduler.

Container Manager. Each Manager corresponds to a container, which is co-located with OpenWhisk's Container Proxy. It accepts the pre-loading requests in the form of standard Run HTTP, augmented with a *"state"* parameter. The Manager can thus seamlessly drive the function process to transition containers to the specified state (*i.e.*, pre- or off-load ML artifacts). Note that the state between Imported and Warming is transitioned by killing and initializing this process. Moreover, we enable the cross-function container sharing using the Init requests. For instance, when a container of f_A is shared with f_B, the corresponding Manager accepts and handles the f_B's Init request to unzip its code and model files, thereby replacing the existing ML artifacts. This does not influence the running function process and loaded artifacts.

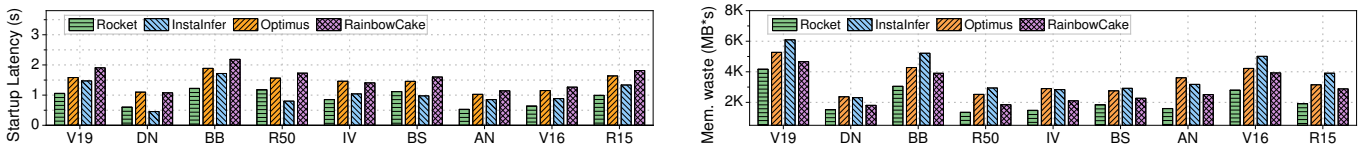
VI. Experimental Evaluation

A. Experimental Environment

Testbed setup. Our evaluations are conducted on an OpenWhisk cluster composed of one controller server and 6 invoker servers. The client is hosted on a separate server. Specifically, each server is equipped with two 8-core Intel Xeon E5-2650v2 CPUs @ 2.60 GHz, 64 GB of memory. Moreover, these servers are connected via a Dual port 10Gb Ethernet NIC.

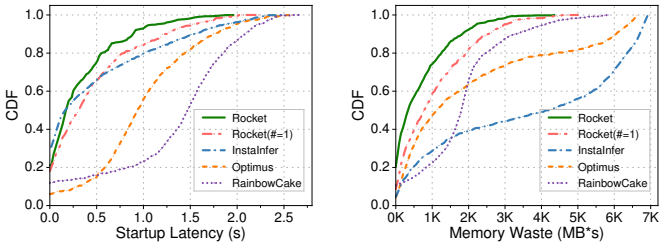
ML artifacts. We use nine widely-used ML models to serve as the base models, involving: 1) AlexNet [38], ResNet family [39], VGG family [40], InceptionV3 [41], and DenseNet [42], adopted from Imgclsmob [43], a computer-vision model repository; 2) Bert family [28] for natural language processing. The models are implemented using different ML libraries, *i.e.*, PyTorch [44], TensorFlow [45] and MXNet [46]. In general, the detailed ML artifacts are shown in Table I.

Invocation traces. To approximate the real-world invocation patterns, we adopt the Azure Function Trace [5], collected from an actual production environment over a 14-day period, to drive the client in generating user requests. Each ML artifact in Table I corresponds to an inference function. Given the heterogeneity shown in Fig. 4(a), we randomly sample a trace per function for various overall IATs.



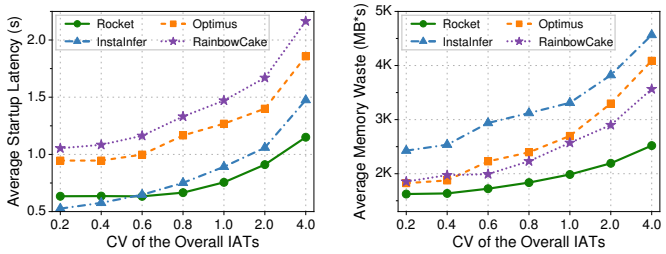
(a) Startup latency. (b) Memory waste.

Fig. 10. The overall performance of Rocket against the three baselines with nine inference functions.



(a) Latency distribution. (b) Memory distribution.

Fig. 11. The impact of model variants on overall performance.



(a) Average startup latency. (b) Average memory waste.

Fig. 12. The scalability of various methods with varying invocation patterns.

TABLE I
FUNCTIONS WITH VARIOUS ML LIBRARIES AND MODELS.

ML library	Model	Parameter
PyTorch	VGG-19 (V19)	143.7M
	DenseNet (DN)	7.98M
	Bert-Base (BB)	110M
TensorFlow	ResNet-50 (R50)	25.6M
	InceptionV3 (IV)	23.8M
	Bert-Small (BS)	28.5M
MXNet	AlexNet (AN)	62.4M
	VGG-16 (V16)	138.4M
	ResNet-152 (R15)	60.2M

Baselines. We evaluate Rocket against the following typical state-of-the-art baselines with default parameters.

- **RainbowCake** [12] is a layer-wise container caching and sharing solution for serverless computing, but keeps the entire ML artifacts alive in User containers.
- **InstaInfer** [15] is an ML artifact pre-loading solution for serverless inference, which dynamically pre-loads various ML artifacts to each idle container.
- **Optimus** [18] is a partial model pre-loading and sharing solution for serverless inference, which shares common operators among functions with model transformation.

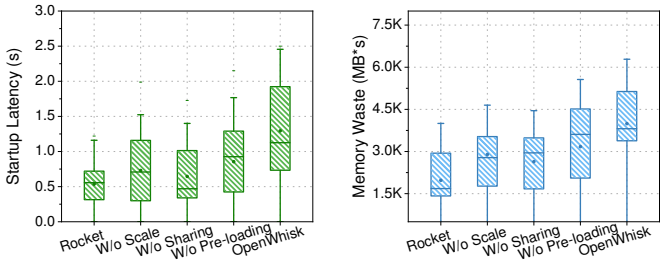
B. Performance Evaluation

Overall performance. We compare the overall performance of Rocket against the baselines in terms of startup latency and memory waste. These two metrics are averages across all invocations for each function. Note that InstaInfer [15] integrates multiple popular cold-start solutions, we thus present the best-performing results. Moreover, the weights γ in the unified cost and the number of classes K in the CB are set to 0.0024 and 5, respectively, which will be discussed later.

Startup latency. Fig. 10(a) illustrates that Rocket can significantly accelerate the nine inference functions with distinct ML artifacts. It reduces startup latency by 15.8%–38.7%, owing to the efficient hierarchical ML artifact pre-loading and sharing. This is because the baselines remain fixed pre-loading, despite the varying degrees of sharing. Concretely, Optimus mitigates loading overheads by reusing idle containers with in-memory model transformation. Neglected, each invocation still requires importing ML libraries, gaining little for functions with lighter models (*e.g.*, AN and DN). With proactive, full pre-loading, InstaInfer obtains comparable benefits to Rocket. Due to heavy ML artifacts, however, it is tough to find shareable containers with ample memory. In addition, RainbowCake pre-warms the container alone, limiting its performance to passive ML artifact retention. In contrast, Rocket flexibly pre-loads partial and shared ML artifacts, achieving robust startup acceleration.

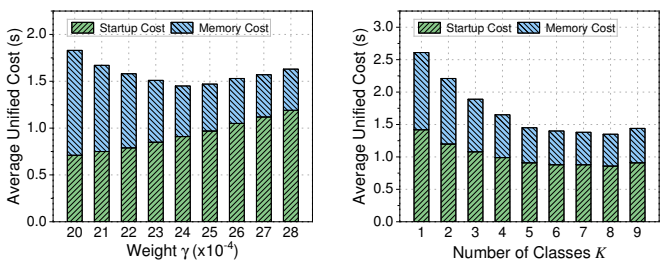
Memory waste. Fig. 10(b) shows the corresponding memory waste of Rocket against these baselines. As illustrated, Rocket outperforms them by 26.75%–43.8% in memory savings. Due to partial model sharing, Optimus shrinks the frequency of full cold-starts and corresponding memory footprints. However, its efficiency is related to the similarity among models, and thus the variance is relatively large. Further, InstaInfer loads more ML artifacts into each idle container for higher hit rates, but the gains are negated by the ensuing duplicate ML runtimes with the limited sharing. Despite only pre-warming lite, general containers, RainbowCake passively calculates a keep-alive time to off-load ML artifacts, which is not timely and thereby incurs non-negligible memory waste. By avoiding these issues, Rocket conducts proactive pre-loading, whose sharing can be built over finite ML libraries, at a fine time granularity.

The impact of model variants. Recall that most production models are pre-trained by fine-tuning a few of top layers [16]. To examine their impact on Rocket, we generate five variants for each base model. As illustrated in Fig. 11, the performance gap between Rocket and each baseline is further widened. This derives from the sharing of front models among variants from the same base model under the P-loaded state. The increased model similarity also benefits Optimus, due to the reduced operator edit distances. For the other baselines, they are unaware of model knowledge, leading to performance fluctuations with more workloads. Despite the infeasible sharing (*i.e.*, #=1), the partial model pre-loading still favors the trade-offs in our goal, due to the presence of "cut-off" points.



(a) Startup latency. (b) Memory waste.

Fig. 13. The ablation performance of various variants for Rocket.



(a) Parameter γ. (b) Parameter K.

Fig. 14. The sensitivity of parameters in Rocket on the unified cost.

Robustness evaluation. To assess the robustness of Rocket, we alter function invocation patterns with different burstiness. Following [12], we sample seven invocation trace sets. Their CVs of the overall IAT range from 0.2 to 4.0, which is higher for more bursty and volatile request arrivals. Note that the CV for each function within a set is not uniform. Fig. 12 displays the average startup latency and memory waste. As the CV increases, Optimus and InstaInfer exhibit apparent performance drop, primarily due to a reduced availability of idle containers during bursty periods. Similar to RainbowCake, thereby, they incur a higher frequency of function cold-starts. Owing to the agile pre-loading enabled by the dual-scale prediction, Rocket can robustly tolerate varying invocation patterns.

Ablation evaluation. To examine the effectiveness of each component in Rocket, we compare it with the following three variants and vanilla OpenWhisk [36]. (1) *W/o scale*: we predict invocation patterns and conduct the scheduling and pre-loading only at the large-timescale; (2) *W/o sharing*: we turn off the sharing-aware scheduler and dispatch requests through a hash-based load balancing strategy; (3) *W/o pre-loading*: we disable the hierarchical pre-loader and randomly decide the container states. The comparison results are illustrated in Fig. 13.

Fig. 13(a) illustrates that the average startup latency is increased by 32.4%, 25.2%, and 42.7%, respectively, without the three components. Notably, the random pre-loading brings the most severe performance drop. This is because Rocket either maintains excessive high-state containers, resulting in low hit rates and more cold-starts, or more low-state containers with higher startup latency. As demonstrated in Fig. 13(b), a similar performance drop also occurs in memory waste, with 30.7%, 23.6%, and 37.9%, respectively. Overall, the variants still outperform OpenWhisk, further confirming their effectiveness in balancing loading and memory efficiency.

Parameter sensitivity. We analyze the parameter sensitivity of Rocket to the goal (*i.e.*, unified cost (3)). (*i*) For *the weight γ, a conversion factor in* (3), we decide its range according to the latency-to-memory ratio of the cold-start steps to converge them. Fig. 14(a) demonstrates that Rocket achieves the further optimization until γ reaches 0.0024. As the priority of memory costs increases, the increased startup cost brings a rise in the unified cost. (*ii*) With *the number of classes K* growing, Fig. 14(b) shows that Rocket significantly decreases the unified cost by granularly deciding the container states. When K exceeds 5, we observe the diminishing marginal returns, with the potential risk of higher CB overheads.

VII. Related Work

Serverless inference. Due to the high elasticity and cost-efficiency, serverless inference has been widely studied from various aspects. Adaptive batching [47] is proposed to bundle requests of a function for improved throughput. Further, INFless [24] and SMIless [32] offer elastic resource provisioning for inference functions and workflows to ensure performance, and some works [20], [48], [49] enhance GPU efficiency with temporal-spatial sharing. They are orthogonal to Rocket and can be integrated to speed up inference. Similar to ours, other studies improve the loading and memory efficiency with pre-loading and sharing. For example, AsyFunc [26] pre-loads the resource-intensive model layers to cope with bursts. Tetris [16] and Optimus [18] share common tensors and operators across functions. Despite the decreased memory usage, they overlook the notable ML library importing overhead. The latest solution, InstaInfer [15], reuses idle containers by pre-loading multiple ML artifacts for instant inference. Yet, they are forced to live with a heavy load. In contrast, Rocket dynamically pre-loads the partial and shared ML artifacts, mitigating their drawbacks in function startup latency and memory waste.

Cold-start mitigation. Cold-start [5] is an inherent issue in serverless computing, as function startup is an order of magnitude slower than execution. Most mitigation efforts customize container caching strategies (*e.g.*, keep-alive [11], [19] and pre-warm [13], [14]) for functions. Some works speed up container warming using snapshot caching [50], novel virtualization [51] and fork [52] methods. RainbowCake [12] employs layer-wise container caching and sharing for reduced container warming latency and memory waste. When applied to inference workloads, nonetheless, the above studies are in-efficient due to the cold-start bottleneck shifting to the ML artifact loading.

VIII. Conclusion

This paper proposes Rocket, a hierarchical pre-loading and sharing system that reduces function startup latency and memory waste for serverless inference. It employs DIP to estimate the timing and number of containers to be pre-loaded for each function. Based on agglomerative clustering, these containers are scheduled by group to improve sharing efficiency. Finally, Rocket employs a lightweight contextual bandit algorithm to make the online hierarchical pre-loading decision for each function group. Experimental results demonstrate that Rocket can reduce by up to 38.7% on startup latency and up to 43.8% on memory waste, compared with the existing solutions.

REFERENCES

[1] Y. Lu, S. Bian, L. Chen, Y. He, Y. Hui, M. Lentz, B. Li, F. Liu, J. Li, Q. Liu, *et al.*, "Computing in the era of large generative models: From cloud-native to AI-native," *arXiv preprint arXiv:2401.12230*, 2024.

[2] K. Hazelwood, S. Bird, D. Brooks, *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proc. of the IEEE HPCA*, pp. 620–629, 2018.

[3] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for cost-effective and SLO-aware machine learning inference serving," in *Proc. of the USENIX ATC*, pp. 1049–1062, 2019.

[4] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *Proc. of the USENIX ATC*, pp. 397–411, 2021.

[5] M. Shahrad *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. of the USENIX ATC*, pp. 205–218, 2020.

[6] X. Yue *et al.*, "HyFaaS: Accelerating serverless workflows by unleashing hybrid resource elasticity," *IEEE Trans. Parallel Distributed Syst.*, vol. 37, no. 1, pp. 272–286, 2026.

[7] M. Liu, W. Wang, and C. Wu, "Optimizing distributed deployment of Mixture-of-Experts model inference in serverless computing," in *Proc. of the IEEE INFOCOM*, pp. 1–10, 2025.

[8] S. Zeng, M. Xie, S. Gao, Y. Chen, and Y. Lu, "Medusa: Accelerating serverless LLM inference with materialization," in *Proc. of the ACM SOSP*, pp. 653–668, 2025.

[9] "Amazon SageMaker," 2025. https://aws.amazon.com/pm/sagemaker/.

[10] "Azure Container Instances," 2025. https://azure.microsoft.com/en-us/products/container-instances/.

[11] A. Fuerst *et al.*, "FaasCache: keeping serverless computing alive with greedy-dual caching," in *Proc. of the ACM ASPLOS*, pp. 386–400, 2021.

[12] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, *et al.*, "RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing," in *Proc. of the ACM ASPLOS*, pp. 335–350, 2024.

[13] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proc. of the ACM ASPLOS*, pp. 753–767, 2022.

[14] Z. Zhou *et al.*, "Aquatope: QoS-and-uncertainty-aware resource management for multi-stage serverless workflows," in *Proc. of the ACM ASPLOS*, pp. 1–14, 2022.

[15] Y. Sui, H. Yu, Y. Hu, J. Li, and H. Wang, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *Proc. of the ACM SoCC*, pp. 178–195, 2024.

[16] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *Proc. of the USENIX ATC*, pp. 473–488, 2022.

[17] H. Hu, F. Liu, Q. Pei, Y. Yuan, *et al.*, "λgrapher: A resource-efficient serverless system for gnn serving through graph sharing," in *Proc. of the ACM WWW*, pp. 2826–2835, 2024.

[18] 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 the ACM EuroSys*, pp. 1039–1053, 2024.

[19] Z. Li, L. Guo, Q. Chen, J. Cheng, *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. of the USENIX ATC*, pp. 69–84, 2022.

[20] M. Yu, A. Wang, D. Chen, H. Yu, X. Luo, Z. Li, W. Wang, *et al.*, "Torpor: GPU-enabled serverless computing for low-latency, resource-efficient inference," *arXiv preprint arXiv:2306.03622*, 2025.

[21] X. Yue, S. Yang, L. Zhu, S. Trajanovski, F. Li, and X. Fu, "Exploiting wide-area resource elasticity with fine-grained orchestration for serverless analytics," *IEEE Trans. Netw.*, vol. 33, no. 1, 2025.

[22] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[23] A. Bietti, A. Agarwal, and J. Langford, "A contextual bandit bake-off," *Journal of Machine Learning Research*, vol. 22, no. 133, pp. 1–49, 2021.

[24] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "INFless: a native serverless system for low-latency, high-throughput inference," in *Proc. of the ACM ASPLOS*, pp. 768–781, 2022.

[25] Y. Wu, T. T. A. Dinh, G. Hu, M. Zhang, Y. M. Chee, and B. C. Ooi, "Serverless data science - are we there yet? a case study of model serving," in *Proc. of the ACM SIGMOD*, pp. 1866–1875, 2022.

[26] 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 the ACM SoCC*, pp. 324–340, 2023.

[27] "Delving into what happens when you 'import torch'," 2023. https://dev-discuss.pytorch.org/t/1589.

[28] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[29] A. Kızılersü, M. Kreer, and A. W. Thomas, "The Weibull distribution," *Significance*, vol. 15, no. 2, pp. 10–11, 2018.

[30] E. L. Schreiber *et al.*, "Optimal multi-way number partitioning," *Journal of the ACM*, vol. 65, no. 4, pp. 1–61, 2018.

[31] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan, "Autothrottle: A practical Bi-level approach to resource management for SLO-targeted microservices," in *Proc. of the USENIX NSDI*, pp. 149–165, 2024.

[32] C. Lu *et al.*, "SMIless: Serving DAG-based inference with dynamic invocations under serverless computing," in *Proc. of the ACM ASPLOS*, pp. 590–606, 2024.

[33] X. Yue, S. Yang, L. Zhu, S. Trajanovski, and X. Fu, "Demeter: Fine-grained function orchestration for geo-distributed serverless analytics," in *Proc. of the IEEE INFOCOM*, pp. 2498–2507, 2024.

[34] Y. Wang, P. Chen, H. Dou, Y. Zhang, G. Yu, Z. He, and H. Huang, "FaaSConf: QoS-aware hybrid resources configuration for serverless workflows," in *Proc. of the IEEE/ACM ASE*, pp. 957–969, 2024.

[35] A. Agarwal, D. Hsu, S. Kale, J. Langford, L. Li, and R. Schapire, "Taming the monster: A fast and simple algorithm for contextual bandits," in *Proc. of the PMLR ICML*, pp. 1638–1646, 2014.

[36] "Apache OpenWhisk," 2024. https://openwhisk.apache.org/.

[37] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2016.

[40] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[41] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. of the IEEE CVPR*, pp. 2818–2826, 2016.

[42] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proc. of the IEEE CVPR*, pp. 4700–4708, 2017.

[43] "Imgclsmob: A large-scale, high-performance, and open-source image classification model zoo," 2021. https://github.com/osmr/imgclsmob.

[44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[45] M. Abadi, P. Barham, J. Chen, *et al.*, "TensorFlow: A system for large-scale machine learning," *Proc. of the USENIX OSDI*, 2016.

[46] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, *et al.*, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[47] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "BATCH: Machine learning inference serving on serverless platforms with adaptive batching," in *Proc. of the IEEE SC*, pp. 1–15, 2020.

[48] J. Liu, Z. Cai, Y. Liu, H. Li, *et al.*, "SMore: Enhancing GPU utilization in deep learning clusters by serverless-based co-location scheduling," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[49] C. Lv, X. Shi, Z. Lei, J. Huang, W. Tan, *et al.*, "Dilu: Enabling GPU resourcing-on-demand for serverless DL serving via introspective elasticity," in *Proc. of the ACM ASPLOS*, pp. 311–325, 2025.

[50] L. Ao, G. Porter, and G. M. Voelker, "FaaSnap: FaaS made fast using snapshot-based VMs," in *Proc. of the ACM EuroSys*, pp. 730–746, 2022.

[51] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. of the USENIX NSDI*, pp. 419–434, 2020.

[52] D. Du, Q. Liu, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Serverless computing on heterogeneous computers," in *Proc. of the ACM ASPLOS*, pp. 797–813, 2022.