Pre-Warming is Not Enough: Accelerating Serverless Inference With Opportunistic Pre-Loading

Yifan Sui*
Shanghai Jiao Tong University
Shanghai, China
suiyifan@sjtu.edu.cn

Hanfei Yu
Stevens Institute of Technology
Hoboken, USA
hyu42@stevens.edu

Yitao Hu
Tianjin University
Tianjin, China
yitao@tju.edu.cn

Jianxun Li†
Shanghai Jiao Tong University
Shanghai, China
lijx@sjtu.edu.cn

Hao Wang
Stevens Institute of Technology
Hoboken, USA
hwang9@stevens.edu

Abstract

Serverless computing has rapidly prospered as a new cloud computing paradigm with agile scalability, pay-as-you-go pricing, and ease-to-use features for Machine Learning (ML) inference tasks. Users package their ML code into lightweight serverless functions and execute them using containers. Unfortunately, a notorious problem, called cold-starts, hinders serverless computing from providing low-latency function executions. To mitigate cold-starts, pre-warming, which keeps containers warm predictively, has been widely accepted by academia and industry. However, pre-warming fails to eliminate the unique latency incurred by loading ML artifacts. We observed that for ML inference functions, the loading of libraries and models takes significantly more time than container warming. Consequently, pre-warming alone is not enough to mitigate the ML inference function's cold-starts.

This paper introduces InstaInfer, an opportunistic pre-loading technique to achieve instant inference by eliminating the latency associated with loading ML artifacts, thereby achieving minimal time cost in function execution. InstaInfer fully utilizes the memory of warmed containers to pre-load the function's libraries and model, striking a balance between maximum acceleration and resource wastage. We design InstaInfer to be transparent to providers and compatible with existing pre-warming solutions. Experiments on OpenWhisk with real-world workloads show that InstaInfer reduces up to 93% loading latency and achieves up to 8× speedup compared to state-of-the-art pre-warming solutions.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless Computing, Cloud Computing, Cold-Start, Machine Learning

ACM Reference Format:
Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference With Opportunistic Pre-Loading. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3698038.3698509

1 Introduction

With the increasing popularity of machine learning (ML) applications, *e.g.*, image recognition and large language models (LLMs), their resource demands are booming.[1] This makes it imperative to develop performance- and cost-efficient computing architectures to serve large-scale ML inference queries. Serverless computing, as a new cloud paradigm, has gained immense popularity for serving ML inferences due to its agile scalability, pay-as-you-go pricing, and ease-of-deployment. Many ML inference products proposed from

*This work was performed when Yifan Sui was a remote intern student advised by Dr. Hao Wang at the IntelliSys Lab of Stevens Institute of Technology.
†Corresponding author

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698509

[1]Facebook alone serves over 200 trillions of inference queries daily [45].

academia and industry have been shifted to serverless architectures, such as Amazon Alexa [1], Azure RAG Chatbot [11], Nuclio [57], and ServerlessLLM [28].

ML inference applications are packaged as lightweight serverless functions invoked by users on-demand, executed in containers.[2] When an invocation arrives yet no available initialized (also known as "warmed") containers, it has to wait for a container to be launched from scratch—the notorious *cold-starts* [41]. Existing works [4, 16, 24, 30, 48, 72, 73, 77] have been extensively proposed to mitigate cold-starts in serverless computing. The predominant approach is referred to as "*pre-warming*" [16, 30, 48, 72]: creating the container and setting up the runtime in advance, while keeping the container alive after serving a query.[3] Thus, the warmed containers can avoid the cold-starts.

A serverless function typically goes through three stages: 1) container warming, 2) loading dependencies such as Python libraries, and 3) serving the query. For serverless workloads, the container warming dominates the cold-start, while the time cost to load dependencies is negligible. Thus, pre-warming methods suit well for these functions. However, we observed that for ML inference functions, the time spent on loading dependencies—which falls outside the scope of pre-warming strategies—is considerably significant.

Fig. 1 shows a real-world experiment of serving eight popular ML inference functions with invocation patterns following 4-hour industrial traces [72], with state-of-the-art pre-warming methods [30, 48, 72]. Loading the ML artifacts, including large libraries (*e.g.*, PyTorch) and model files (*e.g.*, BERT [23]) from disk into memory, and transferring the model into a GPU, accounts for 70% of the whole latency before the inference is actually executed. Such loading latency cannot be simply mitigated by pre-warming—we argue that *pre-warming is not enough* for accelerating serverless ML inferences.

A few recent studies also noticed this issue and proposed to pre-load ML models [35, 46, 63], allow user-defined warm-up triggers [52], and enable snapshots [8, 80]. However, they cannot completely mitigate the ML artifacts loading stage. Some solutions [35, 46, 63] ignored the library loading, some [8, 80] are incompatible with GPUs, and some [8, 52, 80] introduced additional constraints and delays.

To fully accelerate ML inference functions and achieve a minimal end-to-end latency, we aim to take a step further beyond pre-warming—pre-loading the ML artifacts into containers and GPU instances in advance. Therefore, upon an upcoming invocation, the function can jointly avoid the container warming and ML artifact loading stages to execute inference immediately.

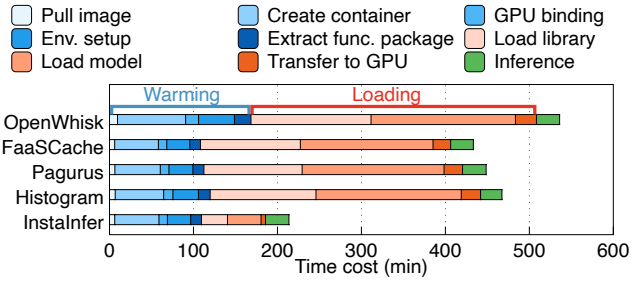

Figure 1: Cumulative time cost and breakdown of real-world serverless inference invocations driven by Azure traces [72]. The blue bars indicate the container warming stage, and the orange bars indicate the ML artifact loading stage.

However, two challenges remain to be addressed in achieving our goals: **1) Pre-loading is memory costly.** For the whole workload, higher acceleration performance means pre-loading more functions, leading to huge memory cost due to the large size of libraries and model files. **2) Pre-loading must avoid any extra function startup overheads.** Serverless functions usually have critical latency requirements (sub-second level) [72]. Pre-loading libraries and ML artifacts should be lightweight and transparent to avoid incurring any additional overheads.

This paper proposes InstaInfer, an opportunistic pre-loading system for serverless inference tasks to tackle these challenges. To balance the trade-off between minimizing loading latency and avoiding memory wastage, InstaInfer pre-loads functions only in existing warmed containers and GPU instances created by the platform, rather than proactively reserving memory.[4] To consistently provide optimal function acceleration, InstaInfer efficiently utilizes idle resources by dynamically loading and offloading functions. Besides, InstaInfer is compatible with existing pre-warming and keep-alive schemes by avoiding interfering with the container creation or removal policies.

We summarize InstaInfer's key contributions as follows:

- We observe the bottleneck of loading ML artifacts in serverless inference systems and propose the opportunistic ML model pre-loading technique to achieve minimal function startup latency.
- We design a pre-loading scheduler that accelerates the cluster-wide workload, which is compatible with existing pre-warming solutions.

[2]The term "container" here denotes virtual environments that execute function invocations in serverless computing, such as Docker containers and Firecracker MicroVMs.

[3]In the context of this paper, we use the term "pre-warming" to encompass both the techniques of pre-warm and keep-alive.

[4]The warmed containers include both pre-warmed and kept-alive containers

- We implement InstaInfer atop OpenWhisk, deploy it on an AWS EC2 cluster, and evaluate it using industrial traces and popular inference functions. Extensive experiments show that InstaInfer reduces the end-to-end function latency by 87% compared to start-of-the-art solutions.

2 Motivation and Background

2.1 Dissecting Serverless Inference

We carefully profile real-world serverless inference invocations and summarize their lifecycle into three stages: 1) container warming, 2) ML artifact (*e.g.*, libraries and models) loading, and 3) ML inference. Fig. 2 shows a dissection of a serverless inference process invoking a SeBS benchmark function [21] running the ResNet152 model.

Container warming. Upon an inference request to the model, the serverless platform begins to prepare and warm up the container, including pulling the base runtime image to create the container instance, initializing and bounding a GPU to the container, and configuring the required runtime environment. The configuration process involves setting up networks (*e.g.*, VPC), security configurations (*e.g.*, configuring firewalls, establishing secure connections), setting environmental variables (*e.g.*, model path, log level, and API key of remote storage), and deploying user custom configurations (*e.g.*, timeout and concurrency settings). Then, the container retrieves and unzips the function package uploaded by the developer. The package contains the ResNet152 model's *binary* ".pth" file, associated Python scripts, and dependent libraries.

ML artifact loading. After the container is warmed up, it starts to load ML artifacts (*e.g.*, ML library and model files) into CPU and GPU memory. Specifically, each library undergoes a initialization process to be loaded into memory. Then, the ML inference model, *i.e.*, a pre-trained ResNet152 model, stored in the *binary* ".pth" format, is read and deserialized into the container's CPU memory to reconstruct the model structure and weight parameters. The process of reading and deserializing models is I/O- and CPU-intensive. Finally, if a GPU is attached to the container, the model will be transferred from the CPU memory to the GPU memory.

Inference. After the warming and loading stages, the function executes the inference on the incoming user data with the loaded ResNet152 model on the GPU. When the user receives the returned inference results, the function will be either terminated or kept alive based on the serverless platform's policy.

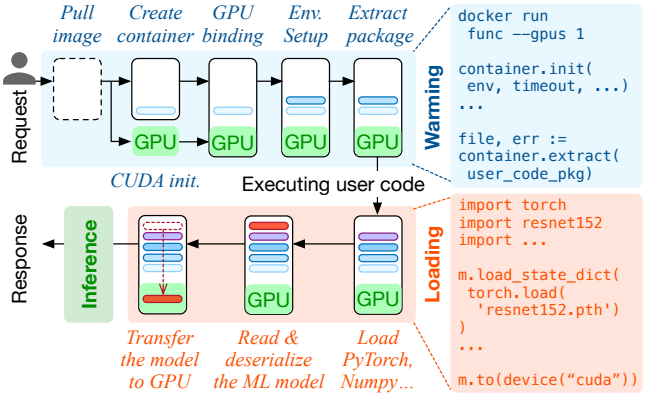

Figure 2: The life cycle of a serverless inference function with the ResNet152 model.

2.2 Container Warming *vs.* ML Artifact Loading

As Fig. 2 shows, a major indicator to distinguish the two stages, *i.e.*, container warming and ML artifact loading, is whether the container starts executing user code. General serverless workloads share the **container warming** stage, known as the "cold-start" issues. These issues have prompted extensive research on mitigating the latency introduced by "cold-starts," resulting in various solutions such as container caching [13, 16–18, 30, 32, 48, 49, 55, 61, 67, 68, 72, 77, 85] and sharing [4, 26, 48, 51, 59, 71, 85], snapshotting [8, 17, 24, 69, 80, 82], and virtualization refactoring [3, 8, 24, 31, 69, 73, 75].

However, the **ML artifact loading** stage is specific to serverless ML workloads due to the lengthy loading time of increasingly larger neural network models and their dependent libraries. General serverless workloads (*e.g.*, web serving and video processing) also have this loading stage but typically take much less time than the warming stage. Fig. 1 shows that the loading stage has dominated the end-to-end latency of serverless ML inference requests, yet it is overlooked by the aforementioned "cold-start" solutions, which are designed for general serverless workloads. Therefore, we argue that pre-warming is not enough for serverless inference functions.

2.3 The Necessity of Pre-loading

To further demonstrate that pre-warming alone is insufficient for eliminating inference functions' cold-starts, we select the eight most popular ML models based on their GitHub popularity. We conduct an experiment using real-world workloads driven by 4-hour industrial invocation traces from Azure [72]. Four NVIDIA A10 GPUs are used for inference. The Azure trace records the timing and frequency of real-world function invocations over the four hours. We swipe the whole Azure trace and randomly select eight function traces to build the

workload. Each trace is mapped to one benchmark function and drives the invocations in the experiment. The detailed experimental setup is described in Sec 7.2.

We implement OpenWhisk's default keep-alive policy and three state-of-the-art pre-warming methods, including Histogram [72], FaaSCache [30], and Pagurus [48], inside OpenWhisk as baselines. These strategies are compared against our proposed method, InstaInfer, which focuses on pre-loading. We report the total time spent on warming, loading, and inference stages for the entire workload for each method.

As shown in Fig. 1, existing pre-warming methods mitigate the warming latency over OpenWhisk. However, loading ML artifacts dominates the overall latency before inference with over 68% of the time, while only 25% is spent on warming and just 6% for inference. Thus, existing approaches severely overlooked the pre-loading opportunity for serverless inference tasks. In contrast, InstaInfer reduces the time for the entire workload by over 55%, demonstrating that pre-loading significantly reduces the overall latency.

Although the loading stage can be accelerated through other methods like using snapshot [8, 80], compressed memory [70], and RDMA [83] to minimize the I/O overhead associated with reading library and model files, these methods cannot enhance the library initialization and model deserialization stages. Consequently, they are insufficient for accelerating inference function.

2.4 Existing Solutions' Limitations

Current works have attempted to eliminate functions' cold-starts in three directions: *container pre-warming* [16, 30, 48, 72, 73, 77], *snapshot* [8, 17, 24, 69, 80, 82], and *model pre-loading* [35, 46, 63].

Pre-warming, the most mainstream method for mitigating cold-starts, assumes that functions start execution immediately after warming. Thus, the warming stage is identified as the primary bottleneck. It predictively initializes the container before request arrivals and keeps the container alive after function executions. However, for inference functions, the unique loading delay prevents pre-warming methods from fully mitigating the whole latency.

Snapshot methods capture functions' completed states as checkpoints on disk. When requests arrive, the snapshots are restored into the process and start execution. For inference functions, snapshots can freeze the state with loaded models and libraries to skip the loading stage, hence outperforming pre-warming. However, the large size of model and library files introduces high latency when the snapshot is loaded from the disk. Moreover, these solutions rely on Linux's memory mapping, which is incompatible with GPUs due to the difficulty in capturing and restoring the GPU memory and context as they are separate from CPU memory.

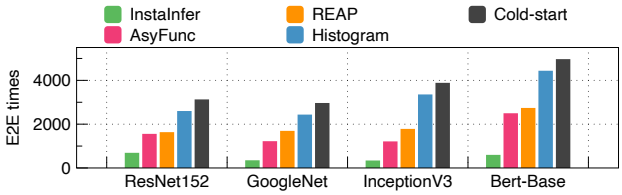

Figure 3: Time cost of executing inference functions.

Naive model pre-loading methods address the model loading bottleneck by either sharing common layers among different models [35, 46] or pre-loading part of the layers post-warming [63] However, they largely ignore crucial stages such as loading libraries and transferring models to the GPU. Additionally, the assumptions of layer similarity across models severely limit their effectiveness, hindering function acceleration across the whole workload.

Several works in other fields propose to employ pre-loading for acceleration. For example, [33, 93] focus on reducing the data fetching latency in databases. [39, 60] pre-load applications on mobile devices. [12, 74] pre-load information on network devices. However, these approaches do not cover the loading stage of inference tasks.

In conclusion, none of the existing works can eliminate the inference function's loading delay. To further motivate the need for pre-loading, we evaluate the latency of different inference functions with all types of cold-start mitigation schemes. Detailed evaluation setup is in Section 7.2. We implement Histogram [72], REAP [80], and AsyFunc [63] as baselines to represent pre-warming, snapshot, and naive model pre-loading, respectively. Fig. 3 shows that InstaInfer outperforms all other baselines by achieving minimal loading latency via its pre-loading.

2.5 The Opportunity of Pre-loading

A straightforward idea for realizing pre-loading is to load all inference functions in advance, which is infeasible due to excessive CPU and GPU memory requirements. Therefore, an ideal solution must seek a balance in reducing loading latency and resource costs. Fortunately, the existence of idle containers created by providers and the over-allocation phenomenon of functions [27, 32, 66, 72, 86, 87, 92, 92, 94] present an opportunity for pre-loading without extra resource costs.

Serverless providers like Microsoft Azure, AWS, and IBM usually keep large volumes of idle containers on standby to serve incoming requests [9, 16, 72]. We only leverage those existing idle containers for pre-loading, avoiding any extra containers and additional resource costs.

Furthermore, due to the fixed proportion between function's computation ability and memory size [10], numerous studies [27, 32, 66, 72, 92] have demonstrated that for optimal execution speed and handling peak workload, inference functions tend to over-provision memory to hold the libraries and

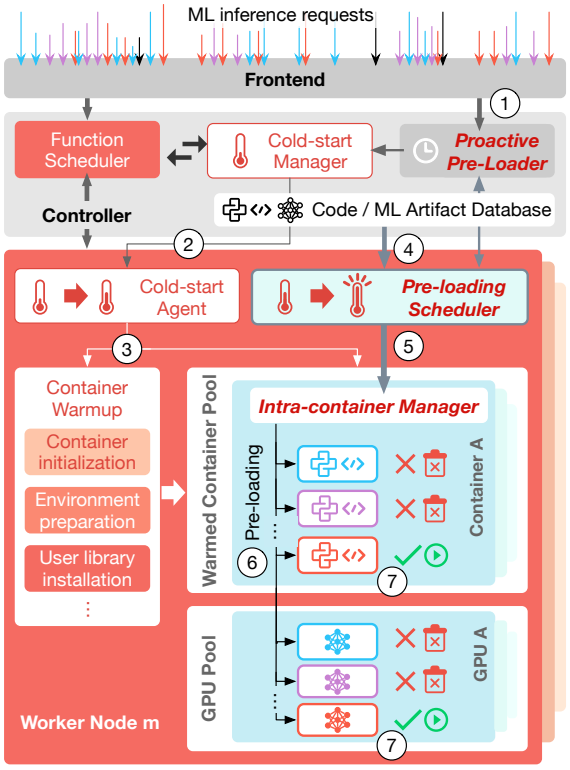

Figure 4: System overview. Boxes with *red bold italic* names are new components introduced by InstaInfer.

models. Therefore, the vast memory gap between containers' running and idle states presents another opportunity for our *opportunistic pre-loading*.

3 An Overview of InstaInfer

3.1 Objectives & Challenges

InstaInfer aims to achieve the following objectives:

- ***Instant inference***: Minimizing the overall end-to-end (E2E) latency of ML inference invocations.
- ***Zero wastage***: Utilizing only the idle capacities in existing containers and GPU instances to pre-load functions.
- ***Transparent to providers***: Pre-loading should avoid conflicts with the platform's inherent pre-warming mechanism.

To achieve the above objectives, we seek answers to the three challenging questions:

How to maximize the acceleration performance with limited idle containers and GPU instances? With only idle containers and GPU instances, we cannot pre-load all functions simultaneously. We must identify and select functions with a high potential for latency improvement and accurately assign them to each container instance.

How to avoid extra resource overheads when pre-loading functions? Holding libraries and models in containers can be memory-costly. We must seek a balance between memory waste and more pre-loading for optimal acceleration.

How to enable pre-loading without incurring additional function startup overheads? Serverless functions typically have critical latency requirements. For example, over 50% of functions on Azure Functions execute in less than one second [72]. We must design the pre-loading process in a lightweight and transparent manner to avoid any extra function startup overheads.

3.2 InstaInfer's System Architecture

We introduce the design of InstaInfer, an opportunistic pre-loading framework to mitigate the loading stage of inference functions. To achieve optimal acceleration within resource constraints, we design a secure instance-sharing mechanism that allows multiple functions to be pre-loaded simultaneously into a single container and share a GPU. InstaInfer includes three principal components: Proactive Pre-Loader, Pre-Loading Scheduler, and Intra-Container Manager.

Proactive Pre-Loader leverages the prediction model of the platform's pre-warming mechanism to forecast function invocation arrivals. The prediction results are then used to determine when to pre-load each function. To achieve cluster-wide acceleration, when receiving a request, it routes the request to the worker node that has pre-loaded the function.

Pre-Loading Scheduler runs on each worker node and assigns functions that need pre-loading to proper containers and GPUs. To maintain optimal acceleration over time, it dynamically makes pre-loading and offloading decisions based on the worker node's container creation and removal events triggered by the platform's pre-warm and keep-alive policies.

Intra-Container Manager independently operates the loading and offloading executions for each function. We design a three-tier security protection mechanism to ensure the security and privacy of each pre-loaded function that shares the same container.

3.3 InstaInfer's Workflow

Fig. 4 shows the workflow and architecture of InstaInfer. Upon the arrival of an ML inference function invocation, InstaInfer follows a five-step workflow:

Stage 1: The Proactive Pre-Loader records the arrival of each inference function's requests. It then predicts the arrival time of the next invocation to determine the optimal moments for loading and offloading each function (Step ① in Fig. 4).

Stage 2: The Proactive Pre-Loader selects a worker node with enough available resources and sends the prediction result to the node's Pre-Loading Scheduler. The scheduler then pre-loads the function in a suitable idle container, extracting the function's code, and unzipping ML artifacts from the platform's database (Step ④).

Stage 3: Concurrently, each request activates the platform's cold-start manager, which prompts the cold-start agent to control the creation and removal of containers based on the pre-warming mechanism (Steps ② and ③). The events of container removal and creation trigger the Pre-Loading Scheduler to make pre-load and offload decisions, which are asynchronous with Stage 2.

Stage 4: When the request arrives, the Proactive Pre-loader routes the request to a worker node that has pre-loaded the corresponding function. Then, the node's Pre-Loading Scheduler selects an idle container that pre-loads the function and an idle GPU that pre-loads the function's model. The request is then sent to the corresponding container's Intra-Container Manager (Step ⑤).

Stage 5: Once receiving the request, the Intra-Container Manager immediately calls the corresponding function's pre-loading process (Step ⑥) and off-loads all other pre-loaded function states (Step ⑦). We ensure that only one function can use the container during inference to guarantee security and privacy. Meanwhile, the Pre-Loading Scheduler selects other idle containers and GPUs to migrate the off-loaded functions to serve future invocations.

4 Proactive Pre-Loader

Because one container has limited CPU and GPU memory, not all functions can be pre-loaded concurrently. Pre-loading a function too early preempts the loading of other functions while doing this too late misses serving function invocations. Therefore, to achieve optimal acceleration, we design a Proactive Pre-Loader that decides when to pre-load a function based on its invocation arrival prediction. We offload the function to make room for pre-loading other functions if mispredictions occur.

4.1 Function Invocation Prediction

A straightforward approach is to load all functions and never offload them. However, due to the limited memory capacity, pre-loading all functions is infeasible. In contrast, we design InstaInfer to opportunistically pre-load a function right before the invocation arrival and offload the function to allow other pre-loadings if mispredicted.

Existing pre-warming approaches typically hold a predictor to forecast invocation arrivals (*e.g.*, Histogram in [48, 72], ARIMA in [72], Poisson Distribution in [85], Variable Order Markov Model in [13]). InstaInfer employs the platform's inherent prediction model to maintain transparency for serverless providers, avoiding introducing extra operational costs such as building new models.

4.2 Function Pre-Loading and Offloading

To effectively manage pre-loading and offloading of a function, denoted as f, we define two thresholds: a probability $P_{load}(f)$ for pre-loading and a probability $P_{offload}(f)$ for offloading. As the invocation's arrival probability increases, the function is immediately pre-loaded if the probability reaches $P_{load}(f)$. Conversely, if the function remains pre-loaded without being invoked for an extended period, such that the probability exceeds $P_{offload}(f)$, InstaInfer identifies that the prediction is incorrect and offloads the function to free up resources for pre-loading other functions.

Invocation patterns can vary over time [72, 90], and using outdated data severely degrades the prediction accuracy. To enhance pre-loading accuracy, we use a sliding window to capture each function's temporal shifts and align predictions with the latest data. It is compatible with various prediction models as we only adjust the temporal scope without altering the underlying model.

We take the Poisson Distribution model of RainbowCake [85] as an example to show how to compute optimal timings for loading and offloading functions. Let W denote the window size and T_w denote the duration between the last and first invocations within the window. We can compute the request arrival rate as $\lambda_f = \frac{W}{T_w}$. Thus, the probability distribution of the arrival time for the next request is: $F(t;\lambda_f) = 1 - e^{-\lambda_f t}, t \geq 0$.

The future timestamp to load and offload function f, $T_{load}(f)$ and $T_{offload}(f)$ are given by

$$T_{load}(f) = -\frac{1}{\lambda_f}\ln(1 - P_{load}(f))$$

$$T_{offload}(f) = -\frac{1}{\lambda_f}\ln(1 - P_{offload}(f))$$

We set the default $P_{load}(f)$ and $P_{offload}(f)$ to be 6% and 94%, respectively. These values are derived from a sensitivity analysis detailed in Section 7.11.

5 Pre-Loading Scheduler

We design a Pre-Loading Scheduler that dynamically selects and assigns functions to appropriate instances for optimal acceleration. To optimize performance over time, the scheduler adaptively adjusts the pre-loading policy to changes.

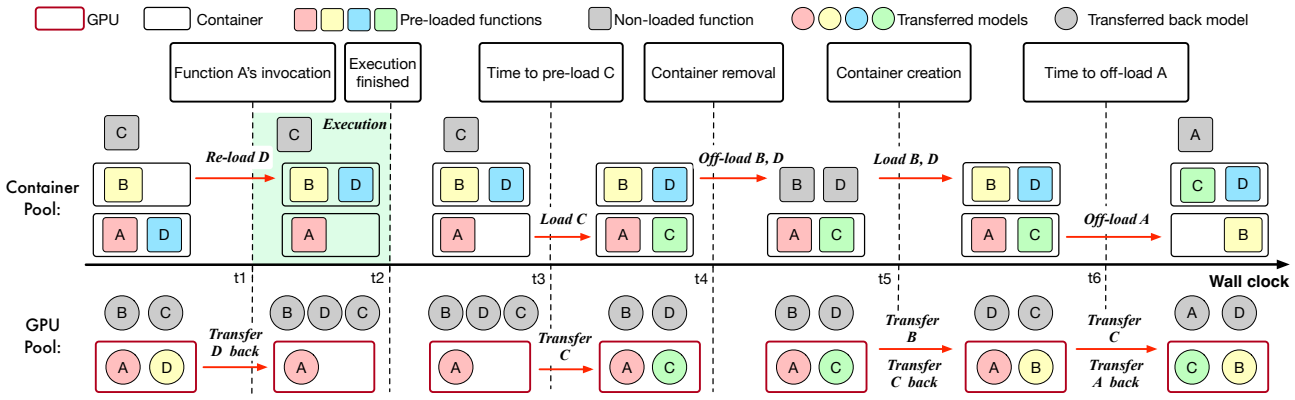

Figure 5: The scheduler's operation after detecting each event.

5.1 Latency-Aware Function Mapping

The simplest way to load functions is one-to-one mapping, where each instance holds only one pre-loaded function. However, this method cannot fully utilize all idle memory to pre-load more functions for further acceleration. To strike a balance between maximum acceleration and avoiding additional costs, we propose an instance-sharing mechanism that allows multiple functions to be pre-loaded simultaneously into a single container until its idle memory runs out while their models share the same GPU.

To select an appropriate container for each function to pre-load, we propose a *Latency-Aware Bin-Packing Policy*. Our goal is to maximize the acceleration of the entire workload, *i.e.*, to maximize the expected value of the saved loading latency among all selected functions. As function loading latency and container capacity are known, this problem fits well with the multiple knapsack bin-packing, wherein containers and functions are treated as bins and items. A bin's capacity is the container memory limit, while an item's weight is the memory cost for loading the function. The item's value is the expected latency saved by pre-loading (calculated as the product of function arrival probability and the loading latency). The objective is to maximize the overall value of the assigned items.

The Latency-Aware Bin-Packing Policy takes functions and idle containers as inputs, using dynamic programming to optimize assignments. The policy computes maximum latency savings $DP[i][j]$ for i functions in j containers by determining whether to place a function based on container capacity and the latency saved. The DP table is updated as: $DP[i][j] = \max(DP[i-1][j], DP[i-1][j-1] + \text{latency\_savings}(i))$. The algorithm iterates over all functions and containers to fill the DP table, with the optimal configuration found at $DP[n][m]$. A backtracking method is used to determine the function-to-container assignments that yield this optimal latency savings.

Besides library and model loading, transferring the model from container CPU memory to GPU memory also introduces non-negligible overhead due to IO and CUDA operations such as memory allocation, especially for large models. For further acceleration, the model of the pre-loaded function can be pre-transferred to GPU. As the GPU pool's capacity is usually smaller than the container memory pool, only part of the models can be kept on GPUs. To optimally determine which model should be kept on GPU, we use the same bin-packing policy wherein GPUs are treated as bins and models as items. The item's value is the expected latency to save, which is calculated as the product of the function's arrival probability and the transfer overhead.

5.2 Optimal Pre-loading Over Time

Due to time-varying workloads, a series of events will cause a fixed bin-packing policy to be sub-optimal: pre-loading or offloading a function, invocation arrivals, container creations, and container removals. We describe how our scheduler reacts to these events to maintain optimal acceleration over time as follows.

As shown in Fig. 5, Functions A, B, and D are pre-loaded on containers, while models of Function A and D are transferred to GPU. In the first case at $t1$, when Function A's invocation arrives, the scheduler first forwards the request to the GPU container that loads Functions A and D. Immediately, Function D is re-assigned to another container to ensure Function A execution performance. Since no GPUs are available, Function D's model is transferred from the GPU back to the container memory. In $t2$, after execution, Function A follows the platform's keep-alive mechanism and remains in the GPU container. Note that since each function has a unique resource configuration, the scheduler adjusts the container's resource limitations immediately upon receiving the invocation to match the function's configuration. The second

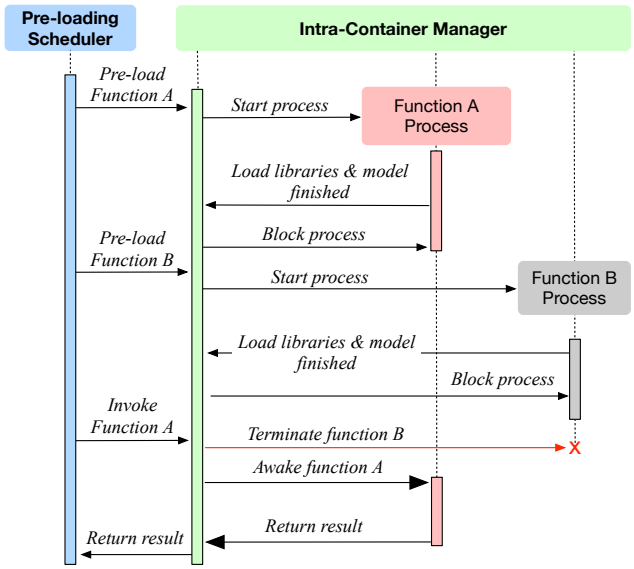

Figure 6: Timeline of InstaInfer's pre-loading.

case is function pre-loading. As shown in $t3$, the scheduler selects a container along with its GPUs that have enough space to load Function C. The third case is container removals. In $t4$, when terminating the container that loads Functions B and D, the scheduler is enforced to offload models of B and D. The fourth case is the container creations. In $t5$, once detecting a new idle container is available, the scheduler preloads Functions B and D inside the new container. The last kind of event is function offloading. The scheduler offloads Function A from both the container and its associated GPU directly, as shown in $t6$. Subsequently, Function C's model is transferred to the GPU to utilize the newly freed resources. The event-driven scheduler dynamically optimizes the bin-packing policy over time while ensuring compatibility with the platform's inherent pre-warming mechanism.

6 Intra-Container Manager

The Intra-Container Manager interfaces with the scheduler to control the process-level execution of functions, including loading, off-loading, and model transfer. Besides, for functions in the same container, it ensures no resource conflicts, and maintains security.

6.1 Pre-Loading Management

As each container holds multiple function's pre-loading processes, the design principle follows three steps: waiting for future invocations and forwarding them to corresponding processes, terminating all processes irrelevant to the incoming invocation, and guaranteeing each function's security and privacy. Upon receiving a pre-loading message from the scheduler, the manager executes the function code to load the library and model. It then transfers the model to the container's corresponding GPU based on the scheduler's decision. After loading, the process enters a blocked state, awaiting future invocations.

The manager's workflow is shown in Fig. 6. After pre-loading Functions A and B, upon receiving Function A's invocation, the manager forwards the request to Function A process's input pipeline, awakening the process to start inference and return the result. To avoid memory preemption and to guarantee function isolation requirements, the arrival of Function A's invocation prompts the immediate termination of all other pre-loading processes and the clearing of their memory allocations. This design ensures that the invoked function runs in a clean and isolated environment.

Similarly, while receiving the off-loading message from the scheduler, the manager terminates the corresponding function's process and erases all related data to protect user privacy. While functions are served as black-box, user code only needs slight changes to expose the model and library files to InstaInfer. We offer two modification options with different objectives:

```
# Original
model.load_state_dict(torch.load(model_path))
inference()...


# InstaInfer
model.load_state_dict(InstaInfer.load(model_path))
sys.stdin.readline()  # wait for request
inference()...
```

Maximum transparency. As the following Python code snippet shows, developers only need to modify two lines of code: First, replace the model loading line (*torch.load*) with the InstaInfer API to expose the model file's path. Second, add the *sys.stdin.readline()* line after loading the model for listening invocations. The function process will be resumed upon receiving requests.

Maximum privacy. If non-intrusive pre-loading is preferred, developers can simply implement a *LOAD* function similar to Azure Warmup Trigger [52] to hold the pre-loading content. The manager calls the *LOAD* API to perform pre-loading without accessing any function-specific data.

6.2 Privacy & Security Guarantee

As multiple functions' code and data are stored in the same container, it's necessary to guarantee the privacy and security of each function. InstaInfer provides a three-layer security protection mechanism. In the user layer, only functions belonging to the same user can be pre-loaded on one container. In the process layer, as shown in Fig. 6, when Function

A's invocation arrives, all other functions in the same container are off-loaded. Their data and code are deleted immediately. In the OS layer, each function's pre-loading process and data are allocated with a unique non-root user managed by Linux privilege domain and privilege control. Meanwhile, the isolation is enhanced with jail techniques [42] such as chroot jails [19]. These designs ensures that a function's process is restricted from accessing the data of other processes, both in memory and on disk. The OS-level isolation also avoids library version conflicts across functions, as the libraries for each function are isolated and stored under the path of its specific Linux user. Furthermore, for the strictest security guarantee that completely forbids container sharing and only allows a container to pre-load one function, InstaInfer still significantly outperforms existing pre-warming methods (Sec. 7.10).

7 Evaluation

7.1 Implementation

We implement a prototype of InstaInfer using Apache OpenWhisk [9]. We implement the Proactive Pre-Loader and Pre-Loading Scheduler as OpenWhisk components using 4K lines of Scala code and implement the Intra-Container Manager in each container's proxy using 2K lines of Golang code. We use PyTorch [62] as the ML environment, although InstaInfer is compatible with any other ML frameworks (*e.g.*, TensorFlow).

Proactive Pre-Loader. We implement the Proactive Pre-Loader in OpenWhisk's load balancer module where all invocations pass by. The Proactive Pre-Loader records the timestamp of invocations, thereby updating each function's prediction.

Pre-Loading Scheduler. OpenWhisk runs a container pool module in each node to manage each container's creation and removal. We implement the scheduler in this module so that the scheduler can acquire all the information it needs for pre-loading. The scheduler sends loading and off-loading messages to the Intra-Container Manager through HTTP requests. To make sure containers' resource limitations match the invoked function's configuration, the scheduler specifies limits using the `--memory`, `--cpu`, and `--gpus` flag when running Docker container.

Intra-Container Manager. We implement the manager in each container's proxy, which is used to communicate with OpenWhisk. The manager is written in Golang. We modify the Action Proxy module to receive the message from the scheduler. We modify the Executor module to execute loading and off-loading. Each pre-loaded function runs as an independent process.

GPU support. As all functions run in Docker containers, we apply the NVIDIA container toolkit [58] that can let the container use the CUDA devices without any additional configuration. To improve GPU resource utilization, we use NVIDIA MPS [22] to partition a GPU for multiple functions and control the GPU limitation of each function.

7.2 Experiment Settings

We describe the experimental settings for evaluating InstaInfer and state-of-the-art baselines.

Testbed: We evaluate InstaInfer on three OpenWhisk clusters: 1) **Single-node CPU cluster** on an AWS m5.16xlarge EC2 instance with 64 Intel Xeon Platinum-8175 CPU cores and 256 GB memory. We perform the E2E latency evaluation, comparisons with snapshot-based solutions, ablation study, sensitivity analysis, and scalability tests on this cluster. 2) **Single-node GPU server** on an AWS `g5.12xlarge` EC2 instance with 48 CPU cores, 196 GB of memory, and 4 NVIDIA A10 GPUs. We conduct the E2E latency and memory cost evaluation on this cluster. 3) **Multi-node cluster** that includes one controller node and four worker nodes, each an AWS m5.8xlarge EC2 instance with 32 CPU cores and 128 GB of memory. We perform E2E latency evaluation, large-scale evaluation of 1000 functions, and prediction evaluation on this cluster.

Workloads: We select the inference function of SeBS benchmark [21] to load each model. For simplicity, each function only runs one model. To optimize subsequent requests and avoid re-loading if warm containers have cached the function process, we follow the optimization approach of AWS Lambda [7]. We place the model and library loading code within the "INIT" structure and the inference code within the "Handler" structure.

To approximate the real-world invocation patterns, we sample the invocations from the Azure Function traces [72], which are collected in production environments. We scan the 14-day Azure invocation trace files and randomly select eight different 4-hour traces that satisfy the Coefficient of Variation (CoV) requirement for each benchmark function. Each trace is then mapped to an inference function, which drives the invocations during the evaluation. For generality, we define three patterns based on the CoV: Predictable (CoV <1), Normal (1<CoV <4), and Bursty (CoV >4).

Models and Libraries: We use PyTorch [62] as the ML framework. We collect eight most popular ML models in computer vision (CV) and natural language processing (NLP) as evaluation benchmarks based on the number of stars on GitHub: AlexNet [44], Inception_V3 [79], ResNet18 [34], ResNet50, ResNet152, VGG19 [76], GoogleNet [78], and Bert-Base [23]. The model size varies from 45MB to 549MB, providing sufficient diversity for evaluating InstaInfer's efficiency. We expand the function type to 1000 for further evaluation in Section 7.8.

Table 1: The average E2E latency, warming+loading latency, and pre-loading rate of baselines.

Metrics	Avg. E2E (ms) (Speedup ×)			Avg. warming+loading (ms) (Speedup ×)			Pre-loading Rate (%)		
Workload	Predictable	Normal	Bursty	Predictable	Normal	Bursty	Predictable	Normal	Bursty
InstaInfer+Histogram	538 (5.6)	707 (4.7)	814 (4)	295 (9.4)	462 (6.6)	567 (5.3)	79	66	48
Histogram	2642 (1.14)	2661 (1.24)	2630 (1.24)	2397 (1.15)	2409 (1.27)	2387 (1.26)	-	-	-
InstaInfer+Pagurus	468 (6.4)	552 (6)	618 (5.3)	223 (12.4)	309 (9.8)	376 (8)	85	78	71
Pagurus	2553 (1.18)	3017 (1.1)	2624 (1.3)	2304 (1.2)	2771 (1.1)	2382 (1.26)	-	-	-
InstaInfer+FaaSCache	826 (3.6)	955 (3.5)	1165 (2.8)	581 (4.7)	709 (4.3)	917 (3.3)	63	51	45
FaaSCache	2537 (1.19)	2715 (1.2)	2690 (1.21)	2292 (1.2)	2469 (1.24)	2445 (1.24)	-	-	-
OpenWhisk	3012 (N/A)	3309 (N/A)	3274 (N/A)	2767 (N/A)	3059 (N/A)	3025 (N/A)	-	-	-

InstaInfer+* Settings: As InstaInfer can be easily integrated with pre-warming solutions, InstaInfer+* indicates integration with three solutions: Histogram [72], Pagurus [48], and FaaSCache [30]. InstaInfer pre-loads functions in the warmed containers created by these solutions.

Baselines: we compare InstaInfer with the state-of-the-art baselines that mitigate cold-starts in serverless computing: 1) **OpenWhisk** [9], the default keep-alive policy of OpenWhisk that keeps each container alive for a fixed 10 minutes after invocation. 2) **Histogram Policy**, a histogram-based container caching approach to dynamically determine when to pre-warm the container and how long the container is kept alive by predicting the inter-arrival time of function invocations. We implemented the Histogram Policy inside OpenWhisk. 3) **FaaSCache** proposes a Greedy-Dual keep-alive caching policy to keep functions alive. Our evaluation reused FaaSCache's open-sourced code repository [29] in OpenWhisk. 4) **Pagurus** avoids cold start by "lending" other functions' idle containers to the function being invoked.[5] 5) **REAP** [80] is a snapshot-based cold start mitigation method that stores function completion states as snapshots on disk. 6) **Azure Function with warmup trigger** [52] allows pre-loading user-defined content while scaling up new instances.

Evaluation Metrics: **1) End-to-End (E2E) latency:** the total time of an invocation from being triggered to returning the results. **2) Warming+Loading latency:** the time period before the inference is actually executed, including both container warming and ML artifacts loading. **3) Pre-loading rate:** the ratio of invocations whose function has already been pre-loaded to the total invocations. **4) Speedup:** the acceleration performance against baselines. **5) Memory cost:** the platform's CPU and GPU memory consumption for running the whole workload.

7.3 Reducing E2E Latency

We evaluate InstaInfer+* and baselines on the single-node cluster. Fig. 7 shows that integrating InstaInfer with the baseline solutions reduces up to 86% E2E latency and 93% warming+loading latency compared with the pre-warming baselines and vanilla OpenWhisk, as InstaInfer effectively mitigates the latency with library and model pre-loading.

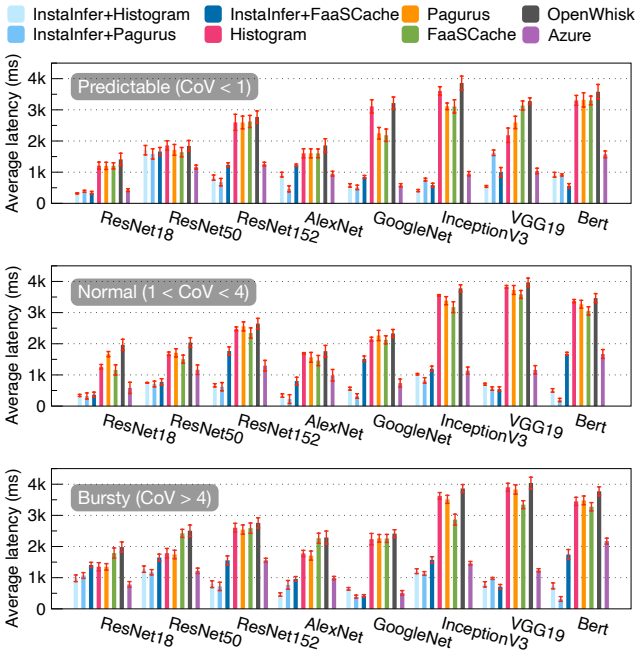

Figure 7: Average E2E latency of InstaInfer+* and baselines running the Predictable, Normal, and Bursty workloads.

The Azure Function baseline utilizes the warmup trigger [52] to pre-load user-defined contents, including libraries and models. Deviating from the traditional on-demand serverless products, warmup trigger is only available on the Premium plan [53], which keeps at least one "always-on" container and scales dynamically. For fair comparisons, we select the "EP2" configuration with two "always-on" containers, each with 4 vCPUs and 7 GB memory, totaling at least 64 vCPUs, compared to 48 vCPUs in InstaInfer.

Fig. 7 shows that InstaInfer outperforms Azure Function when serving most of the functions. Despite Azure's minimal warming latency due to "always-on" containers, it exhibited three main drawbacks compared with InstaInfer: 1) The function's library files are stored on Azure Files [54]. During loading, reading many small files incurs heavy overhead (over 10 seconds). 2) Warmup triggers only work during

[5] Pagurus's original implementation [47] is not for OpenWhisk. We reproduced Pagurus in OpenWhisk and tuned its performance to the best for a fair comparison.

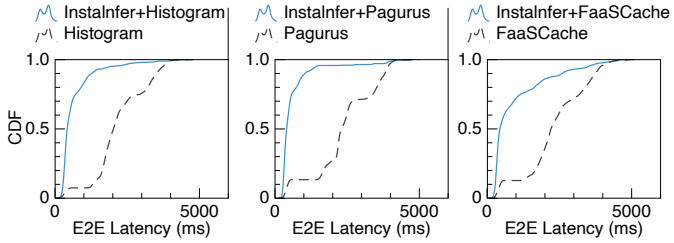

Figure 8: CDF of InstaInfer+* and baselines running the Normal workload.

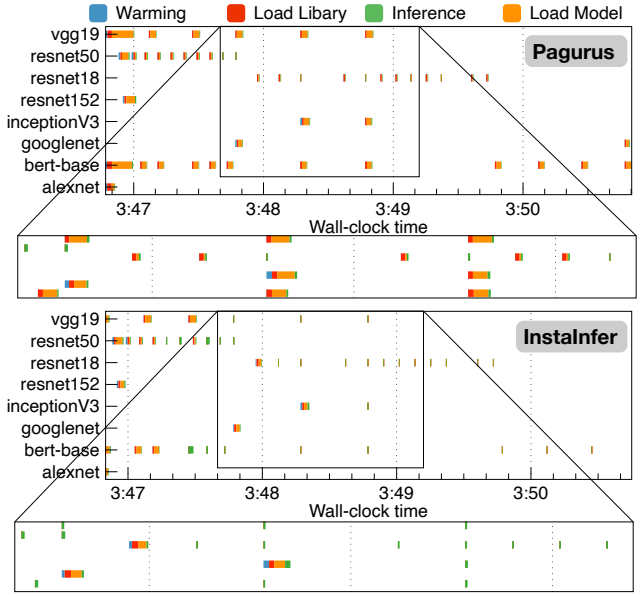

Figure 9: E2E latency breakdown of individual invocations served by Pagurus and InstaInfer+Pagurus.

scaling and never proactively pre-load functions in "always-on" containers, losing the opportunity to mitigate loading latency. 3) Unlike traditional serverless products that charge per use, the Premium plan has fixed hourly or monthly fees, leading to over 20× higher expense (Sec. 7.5).

Table 1 presents the average E2E latency, warming+loading latency, speedup, and pre-load rate of each baseline. InstaInfer+* outperforms each corresponding baseline on each metric. InstaInfer+Pagurus achieves the best performance due to having more idle containers for pre-loading. This is because Pagurus removes fewer containers and keeps more warmed containers over other baselines.

To further explore E2E latency reduction, we show the E2E latency's cumulative distribution function (CDF) of running the Normal workload for InstaInfer and each baselines in Fig. 8. The results show that InstaInfer can effectively accelerate the workload without increasing the tail latency.

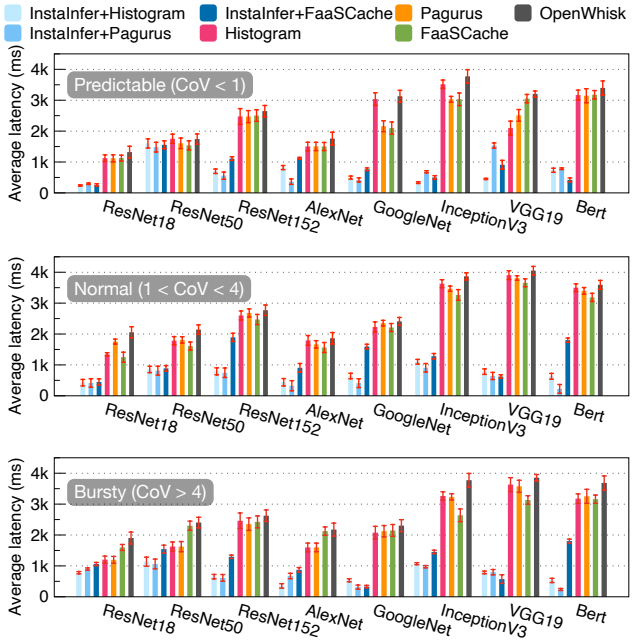

Figure 10: Average E2E latency of InstaInfer+* and baselines running each workload on GPUs.

To show InstaInfer's acceleration effect more intuitively, we present a time breakdown of the E2E latency of Pagurus and InstaInfer+Pagurus running a "Normal" workload in Fig. 9. Pagurus is selected in this case since it outperforms Histogram and FaaSCache. Fig. 9 shows that InstaInfer+Pagurus eliminates the latency of not just the warming stage but also the library and model loading stage for most invocations.

Note that in Pagurus's timeline in Fig. 9, several functions are invoked multiple times within a minute and are required to load everything from scratch due to two main reasons: First, if the request concurrency of a function exceeds the number of cached containers, additional warmed containers must be spawned to serve the extra requests. Second, to share the container among multiple functions, Pagurus transforms a dedicated container into a shareable one, which clears the cached states inside the container. Thus, if a request is served by a shared container, it must re-load the ML artifacts even if it's already warm-started.

7.4 InstaInfer GPU Evaluation

To show the benefits of opportunistic pre-loading in both CPU memory and GPU memory, we evaluate the E2E latency of workloads with InstaInfer in the GPU cluster with 4 NVIDIA A10 GPUs. As shown in Fig. 10, integrating InstaInfer with each baseline can significantly reduce at most

Table 2: Multi-node cluster's average E2E latency, warming+loading latency, and pre-loading rate of baselines.

Metrics	Avg. E2E (ms) (Speedup ×)			Avg. warming+loading (ms) (Speedup ×)			Pre-loading Rate (%)		
Workload	Predictable	Normal	Bursty	Predictable	Normal	Bursty	Predictable	Normal	Bursty
InstaInfer+Histogram	559 (5.6)	712 (4.4)	903 (3.8)	310 (9.2)	461 (6.9)	656 (4.9)	78	66	52
Histogram	2703 (1.16)	2729 (1.24)	2861 (1.28)	2452 (1.17)	2480 (1.3)	2614 (1.23)	-	-	-
InstaInfer+Pagurus	452 (7.7)	564 (6.2)	623 (5.6)	203 (14.1)	313 (10.3)	375 (8.6)	86	79	70
Pagurus	2493 (1.25)	2917 (1.2)	2624 (1.3)	2203 (1.3)	2663 (1.2)	2377 (1.35)	-	-	-
InstaInfer+FaaSCache	821 (3.8)	968 (3.6)	1043 (3.6)	576 (5)	725 (4.5)	811 (4)	61	46	42
FaaSCache	2526 (1.23)	2751 (1.27)	2723 (1.27)	2289 (1.26)	2508 (1.3)	2476 (1.3)	-	-	-
OpenWhisk	3124 (N/A)	3496 (N/A)	3459 (N/A)	2879 (N/A)	3247 (N/A)	3216 (N/A)	-	-	-

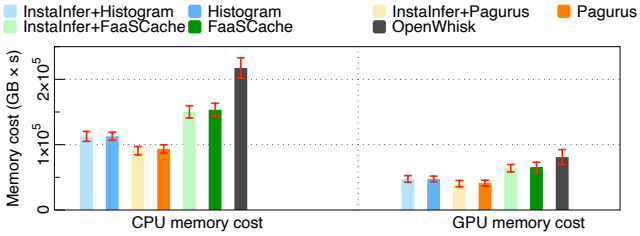

Figure 11: Average memory cost of InstaInfer+* and baselines running the same workload.

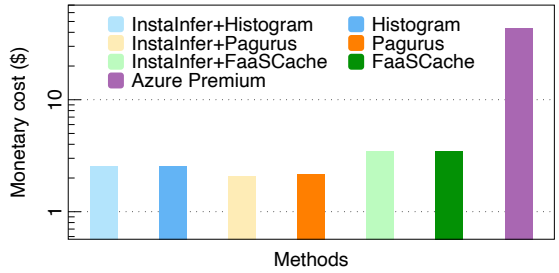

Figure 12: Monetary cost of InstaInfer+* and other baselines running the same workload.

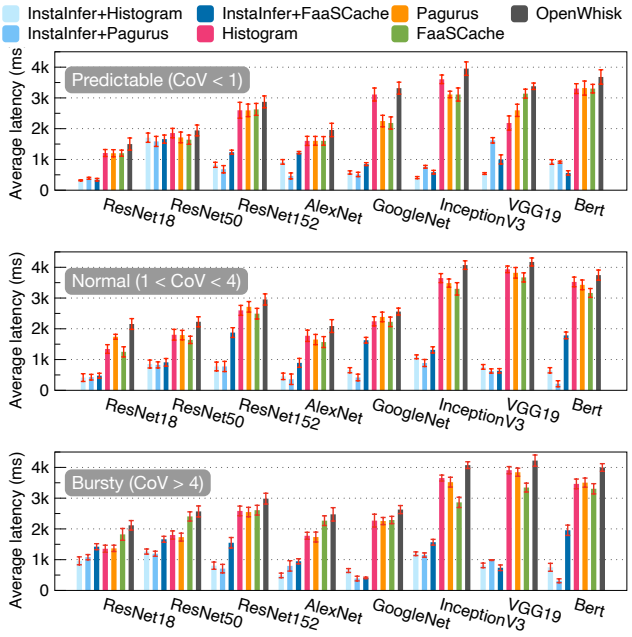

Figure 13: Average E2E latency of InstaInfer+* and other baselines running on the multi-node cluster.

93% average E2E latency for each inference function. Compared with CPU-based InstaInfer in Section 7.3, InstaInfer with GPU pre-loading further improves the function execution time cost as it mitigates the CUDA runtime initialization and model swapping latency.

7.5 Memory and Monetary Cost

We evaluate the monetary cost of InstaInfer, baseline pre-warming methods, and naive pre-loading while running the same Azure trace workload. In the evaluation, InstaInfer is combined with each baseline. In the OpenWhisk Pre-loading baseline, each container can only hold one pre-loaded function. To achieve the same acceleration performance as InstaInfer, more containers are created proactively for pre-loading. Shown in Fig. 11, the container and GPU memory consumption of InstaInfer+* are nearly identical to those of corresponding baselines alone. That's because InstaInfer only reuses the idle container created by the baseline method and does not proactively create new containers. Consequently, InstaInfer does not incur additional resource costs. In contrast, to achieve comparable acceleration performance, OpenWhisk Pre-loading creates more containers than InstaInfer, resulting in at most 2.4× the memory cost and 2× the GPU cost compared to InstaInfer.

Then we evaluate the monetary cost of running the above 4-hour workload using Azure Function pricing model [2]. Fig. 12 shows that the monetary cost of InstaInfer+* is nearly identical to that of the corresponding baseline alone. Although Azure Premium Plan achieves lower E2E latency for several functions according to Fig. 7 than InstaInfer, its expense is 20 times higher than other methods.

7.6 Multi-Node Evaluation

We evaluate the scalability of InstaInfer by conducting experiments on the multi-node cluster. We evaluate the E2E latency using the same benchmarks, metrics, baselines,

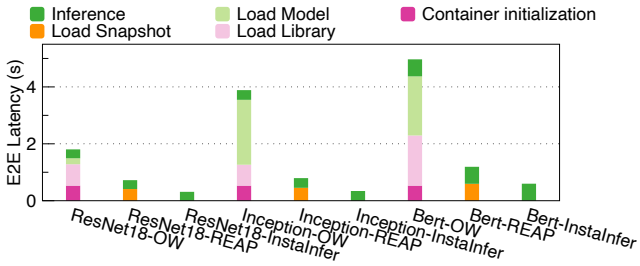

Figure 14: E2E latency of InstaInfer, REAP, and OpenWhisk running benchmark functions with different model sizes.

and workloads from Sec. 7.3. Fig. 13 shows that integrating InstaInfer with baselines reduces up to 87% E2E latency. The performance evaluated on the multi-node cluster is similar to the results observed from the single-node cluster. This consistency suggests that InstaInfer efficiently maintains low loading latency for a variety of workloads in a distributed cluster. Table 2 details the average E2E latency, warming+loading latency, speedup, and pre-load rate for each baseline. The data shows InstaInfer+* consistently outperforms existing baselines across all the metrics.

7.7 Comparisons with Snapshot Methods

To mitigate cold start, some approaches [8, 80] capture the function's complete state as a snapshot and store the snapshot on disk. For ML inference functions, as the snapshot can store the state after loading the libraries and model, it can also address eliminating the loading delay. Thus, we conduct an evaluation between InstaInfer and REAP [80], a snapshot-based serverless method.

We evaluated the E2E latency of three benchmark ML inference functions with small (ResNet18), medium (Inception_v3), large (Bert-Base) models respectively in InstaInfer, REAP, and vanilla OpenWhisk in the same setup. Fig. 14 shows REAP outperforms OpenWhisk. InstaInfer further enhances execution by 1.5 to 2.5× over REAP.

The reason for InstaInfer outperforming REAP is that InstaInfer does not need to load and restore the snapshot from disk to memory. As REAP's snapshots are all stored in disks, when a request arrives, a snapshot must be read into memory and restored to process, introducing additional latency. Based on the experiment result, the latency is high for inference functions (300–600ms) due to the large size of model and library files. In contrast, InstaInfer keeps functions in memory and achieves negligible latency (5–14 ms in Sec. 7.12).

7.8 Large-Scale Evaluation

To further evaluate the performance of InstaInfer in a more realistic scenario, we extend the workload to 1000 functions on the multi-node cluster. According to Azure[72], the top 18.6% functions make up 99.6% calls. Thus, we selected 50 often-used functions' traces, 150 ones with a once-per-minute call rate, and 800 rarely-called ones. All functions are created based on the eight benchmark models. We give each function a unique identifier (such as ResNet50-1, âĂę, ResNet50-125) to create 125 different functions that run the same model. Since InstaInfer treats the function code as black-box, all functions created are uniquely different.

Table 3: Average E2E latency in large-scale evaluation.

Method	Avg. E2E (ms) (Speedup ×)	Warm + Load (ms) (Speedup ×)
InstaInfer+Pagurus	1482 (2.49)	1184 (2.86)
Pagurus	3201 (1.15)	2896 (1.17)
OpenWhisk	3695 (N/A)	3397 (N/A)

Table 4: Comparison of different prediction methods under varying workloads, metrics including pre-loading rate (%) and speedup (×).

Workload	Poisson	Histogram	RF	ARIMA
Predictable	67 (2.93)	61 (2.65)	50 (1.86)	62 (2.67)
Normal	56 (2.32)	51 (1.93)	47 (1.75)	51 (1.94)
Bursty	42 (1.58)	46 (1.79)	43 (1.59)	40 (1.5)

We evaluate the E2E and warming+loading latency of InstaInfer+Pagurus, Pagurus, and vanilla OpenWhisk using the same workload. The result is shown in Table 3. Besides, we evaluate the pre-loading rate of InstaInfer. For the 50 functions that are frequently invoked, the pre-load rate is 73%. For the 150 less-frequently invoked functions, the pre-load rate is 28%. For the 800 rarely invoked functions, the pre-load rate is less than 1%. Thus, InstaInfer can effectively pre-load the frequently invoked functions and accelerate the workload in large-scale scenarios.

7.9 Prediction Performance Evaluation

To evaluate the robustness of InstaInfer, we choose four prediction models: Poisson distribution, Histogram policy-based prediction [72], Random Forests (RF)[15], and Auto-Regressive Integrated Moving Average (ARIMA) modeling[14]. Each model is used to decide when to load and offload a function. We randomly select 200 function traces from predictable, normal, and bursty workloads, respectively. As shown in Table 4, Poisson achieves the best performance in predictable and normal workloads, whereas Histogram performs best in bursty workloads. InstaInfer pre-loads over 40% functions and speeds up workloads by over 1.5×.

7.10 Ablation Study

We conduct an ablation experiment on the single-node cluster to evaluate the effectiveness of the Proactive Pre-Loader

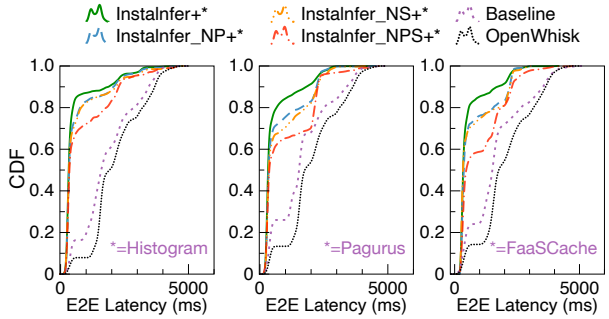

Figure 15: The CDF of E2E latency for ablation of InstaInfer+* and baselines.

and Pre-Loading Scheduler. Three variants of InstaInfer are evaluated and compared with Histogram Policy, Pagurus, and FaaSCache:

- **InstaInfer_NP**: InstaInfer without the Proactive Pre-Loader. This variant lacks the Proactive Pre-Loader, so it does not predict the arrival probabilities of the function. Thus, this variant never determines pre-loading and off-loading proactively, only reacting to container creation, container removal, and invocation arrival.
- **InstaInfer_NS**: InstaInfer without the Scheduler. This variant cannot make optimal assignments and dynamically schedule loading and unloading. For InstaInfer_NS, a function is only pre-loaded under two situations: 1) when receiving the pre-load message from the Proactive Pre-Loader and 2) when a container is idle, its corresponding function will be loaded (*i.e.*, one-to-one mapping).
- **InstaInfer_NPS**: InstaInfer without either the Proactive Pre-Loader or Scheduler. Each container only pre-loads its own function's libraries and models.

Fig. 15 shows the CDF of E2E inference latency under 2-hour "Normal" traces randomly selected from Azure. Regardless of the pre-warming method used, InstaInfer always outperforms other variants due to its full utilization of both the Proactive Pre-Loader and Scheduler. The synergy between these two components ensures the maximum loading latency reduction despite dynamic changes in invocation pattern and the number of idle containers.

On average, InstaInfer accelerates the workload by 1.16-1.28×, 1.21-1.49×, and 1.48-1.73× when compared with InstaInfer-NP, InstaInfer-NS, and InstaInfer-NPS.

7.11 Sensitivity Analysis

We conduct an experiment to evaluate the impact of two InstaInfer hyper-parameters: P_{load}, which decides when to load libraries and models, and the size of the Proactive Pre-Loader's sliding window, used to adapt to recent invocation changes. Fig. 16 shows their impact on the average E2E latency of a workload from Azure Trace.

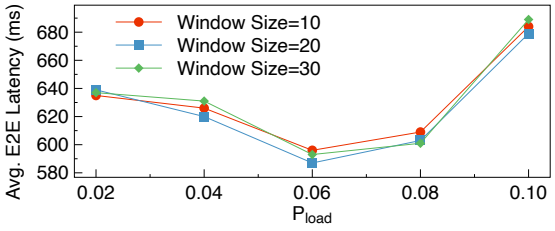

Figure 16: The average E2E latency with different P_{load} and sliding window size.

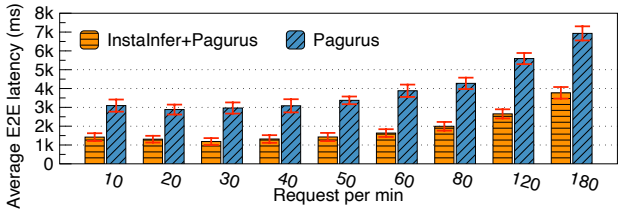

Figure 17: Average E2E latency *vs.* increasing workloads.

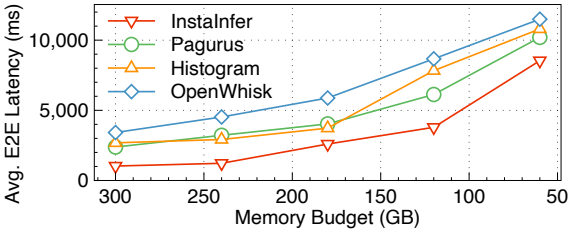

Figure 18: Robustness to limited memory budgets.

As observed, the performance of InstaInfer is not sensitive to the size of the Proactive Pre-Loader's sliding window. Meanwhile, we observed that the value of P_{load} converges to 0.06. Furthermore, the optimal value of P_{load} is not affected by the sliding window size. Although a lower P_{load} means loading a model earlier, leading to a higher hit rate for future invocations. However, pre-loading a function too early risks wasting the available resources, which might be utilized for loading other functions, leading to a sub-optimal acceleration. We set InstaInfer's P_{load} to be 0.06 to achieve the optimal acceleration.

7.12 Scalability and Overhead

To evaluate the scalability of InstaInfer, InstaInfer + Pagurus is given increasingly heavier workloads, varying from 10 to 180 requests per minute. The performance is shown in Fig. 17. InstaInfer consistently outperforms Pagurus across different scales. Then we evaluate the performance of InstaInfer against other baselines under constrained resource budgets by varying the container pool's size. As Fig. 18 shows, InstaInfer consistently outperforms other baselines under different memory budgets, showing stronger robustness.

Next, we report the latency and resource overhead of each component. The Proactive Pre-Loader introduces less than 3ms additional latency under the heaviest workload. The Intra-Container Manager introduces 2ms to 11ms latency overhead, which is caused by the memory preemption of clearing the memory of other pre-loading processes when invocation arrives. This latency varies based on the memory footprint of the to-be-offloaded function. As the scheduler's pre-loading & off-loading decision is asynchronous with serving the invocation, it does not cause latency overhead. Compared with the saved latency (1500-5000ms), the additional latency (5ms-14ms) is negligible. The overhead will be lower when handling fewer invocations.

Under the heaviest workload, the Proactive Pre-Loader consumes less than 0.3 CPU core and 72MB memory; the scheduler consumes 0.3 CPU core and 135MB memory; the Intra-Container Manager consumes 0.1 CPU core and 9MB memory. The overall resource overhead of all InstaInfer's components is negligible compared to the workloads.

8 Related Work

Serverless inference. Motivated by serverless computing's flexibility and cost-efficiency, a few studies proposed to enable ML inference via serverless computing [6, 20, 25, 35, 36, 38, 40, 46, 50, 89]. However, they ignore the ML artifacts loading latency, which extensively inflates the E2E latency. Some works improve inference functions' throughput by dynamically batching requests [5, 84, 91], which is orthogonal to InstaInfer. InstaInfer's Proactive Pre-Loader treats the batched requests as a single call and forwards them to a container. AsyFunc [63] mitigates bursts by pre-loading resource-intensive layers of a model while reusing others from a warmed container, dependent on the availability of warmed containers. Thus, it does not address the cold start problem. Moreover, it targets model loading overhead, which is only 52% of ML artifact loading time as observed in Fig. 1, leaving half of the overall latency sub-optimal. Tetris [46] and Optimus [35] share identical layers across models to address model loading bottlenecks but ignore library loading and GPU transfer overheads. Their effectiveness depends on layer similarity, limiting universality across diverse models. In contrast, InstaInfer accelerates any model.

Cold-start mitigation. Many studies attempt to address cold-start issues, which can be classified into four major categories: 1) Pre-warming [13, 16–18, 30, 32, 48, 49, 55, 61, 67, 68, 72, 77] that predictively pre-warms container in advance [13, 18, 32, 55, 72, 77] and keeps them warmed [16, 17, 30, 48, 49, 61, 67, 68, 72]. 2) Virtualization Refactoring [3, 8, 24, 31, 69, 73, 75] that use new virtualization technique to accelerate warming. 3) Container Sharing [4, 26, 48, 51, 59, 71] that shares container among functions. 4) Snapshot based methods [8, 17, 24, 69, 80, 82] that stores snapshots of functions. Among them, pre-warming, virtualization refactoring, and container sharing focus on container-level speedup for general functions, overlooking the unique loading stage for ML inference functions. Snapshot methods capture inference function states, including loaded libraries and models. However, these snapshot files are large, containing extensive model and library data, leading to a 100âĂŞ1000ms startup overhead as shown in our evaluation (Fig. 14) and the REAP experiment results [80]. Furthermore, these techniques rely on Linux's memory mapping mechanism and are not compatible with GPUs due to difficulties in capturing and restoring separate GPU memory and contexts.

Pre-loading in serverless. Some works [7, 37, 52] allow user-defined pre-loading primitives when starting a new instance. Azure warmup trigger [52] pre-loads the user-defined primitives during instance scaling. However, it only works out during scaling up, failing to tackle the cold start problem. For pre-warmed containers, the trigger does not pre-load components. AWS Lambda static initialization [7] allows components that execute only once during the first invocation to speed up subsequent operations. However, for the first invocation, even if the container has been created, the components cannot be pre-loaded. [37] enables executing user-defined primitives once a container is pre-warmed. However, as a naive pre-loading approach, it falls short of achieving optimal performance due to underutilized idle space. Furthermore, none of these methods is compatible with GPUs.

Function data caching. Some studies [43, 56, 64] cache ephemeral data of functions in local storage or cloud server, while others [65, 81] keep data in containers. Pheromone [88] uses multiple cache mechanisms based on developers' configuration. InstaInfer focuses on pre-loading libraries and models into memory, which is orthogonal to these data caching techniques.

9 Conclusion

This paper proposed InstaInfer, a pre-loading technique for serverless inference that alleviates the ML artifacts loading overhead of ML inference functions by opportunistically pre-loading their libraries and models rather than popular cold-start mitigation approaches. InstaInfer comprises a Proactive Pre-Loader to estimate when to load each function, a Pre-Loading Scheduler to assign to-be-loaded functions to suitable idle containers and GPUs, and an Intra-Container Manager for controlling the loading & off-loading of each function. Extensive experiments with real-world traces showed that InstaInfer reduces startup latency by up to 93% and accelerates the overall workload 8×.

Acknowledgments

We thank our shepherd Dr. Qian Li and anonymous reviewers for their valuable feedback. The work of Yifan Sui and Jianxun Li was supported in part by National Natural Science Foundation of China under grant 61673265. The work of Hanfei Yu and Hao Wang was supported in part by NSF 2153502, 2403247, 2403398, and the AWS Cloud Credit for Research program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] 2023. Alexa Skills - Serverless Applications Lens. https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html. Accessed: 2024-07-07.

[2] 2024. Pricing - Microsoft Azure Function. https://azure.microsoft.com/en-us/pricing/details/functions/. Accessed: 2024-07-12.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. {SAND}: Towards {High-Performance} Serverless Computing. In *Proc. 2018 Usenix Annual Technical Conference (USENIX ATC)*. 923–935.

[5] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: Machine Learning Inference Serving on Serverless Platforms With Adaptive Batching. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–15.

[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. the VLDB Endowment* 15, 10 (2022).

[7] Amazon Web Services. 2023. Optimizing static initialization - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/operatorguide/static-initialization.html Accessed on: 2024-06-12.

[8] Lixiang Ao, George Porter, and Geoffrey M Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*.

[9] Apache OpenWhisk. [n.d.]. [n. d.]. https://openwhisk.apache.org.

[10] AWS Lambda. 2024. Configure Lambda function memory. https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html/. Accessed: 2024-07-07.

[11] Azure Samples. 2024. Serverless AI Chat with RAG using LangChain.js. https://learn.microsoft.com/en-us/samples/azure-samples/serverless-chat-langchainjs/serverless-chat-langchainjs/. Accessed: 2024-07-07.

[12] Amotz Bar-Noy, Richard E Ladner, and Tami Tamir. 2008. Optimal delay for media-on-demand with pre-loading and pre-buffering. *Theoretical Computer Science* 399, 1-2 (2008), 3–11.

[13] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*.

[14] George EP Box and David A Pierce. 1970. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association* 65, 332 (1970), 1509–1526.

[15] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in {AWS} Lambda. In *Proc. 2023 USENIX Annual Technical Conference (USENIX ATC)*. 315–328.

[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proc. the Fifteenth European Conference on Computer Systems (EuroSys)*. 1–15.

[18] Xinquan Cai, Qianlong Sang, Chuang Hu, Yili Gong, Kun Suo, Xiaobo Zhou, and Dazhao Cheng. 2024. Incendio: Priority-Based Scheduling for Alleviating Cold Start in Serverless Computing. *IEEE Trans. Comput.* 73, 7 (2024), 1780–1794.

[19] Bill Cheswick. 1992. An Evening with Berferd in which a cracker is Lured, Endured, and Studied. In *Proc. Winter USENIX Conference, San Francisco*. 20–24.

[20] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. 2022. Sla-driven ML Inference Framework for Clouds with Heterogeneous Accelerators. *Proc. Machine Learning and Systems (MLSys)* 4 (2022), 20–32.

[21] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. Sebs: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proc. the 22nd International Middleware Conference (Middleware)*. 64–78.

[22] NVIDIA Corporation. 2024. NVIDIA Multi-Process Service. Software available from NVIDIA. https://docs.nvidia.com/deploy/mps/index.html Accessed: 2024-05-30.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[24] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proc. the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 467–481.

[25] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a Diet. In *Proc. the 11th ACM Symposium on Cloud Computing (SoCC)*. 45–59.

[26] Tarek Elgamal. 2018. Costless: Optimizing Cost of Serverless Computing Through Function Fusion and Placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*.

[27] Jonatan Enes, Roberto R Expósito, and Juan Touriño. 2020. Real-time resource scaling platform for big data workloads on serverless environments. *Future Generation Computer Systems* 105 (2020), 361–379.

[28] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Locality-Enhanced Serverless Inference for Large Language Models. *arXiv preprint arXiv:2401.14351* (2024).

[29] Alexander Fuerst. 2021. GitHub—aFuerst/openwhisk-caching. https://github.com/aFuerst/openwhisk-caching. [Accessed 26-10-2023].

[30] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 386–400.

[31] Google. 2018. gVisor. https://gvisor.dev/.

[32] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T Kandemir, and Chita R Das. 2020. Fifer: Tackling Underutilization in the Serverless Era. In *The 21st International Middleware Conference (Middleware)*.

[33] Ajay K Gupta and Udai Shanker. 2020. OMCPR: Optimal mobility aware cache data pre-fetching and replacement policy using spatial K-anonymity for LBS. *Wireless Personal Communications* 114, 2 (2020), 949–973.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. the IEEE conference on computer vision and pattern recognition (CVPR)*. 770–778.

[35] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proc. the Nineteenth European Conference on Computer Systems (EuroSys)*. 1039âĂŞ1053.

[36] Haichuan Hu, Fangming Liu, Qiangyu Pei, Yongjie Yuan, Zichen Xu, and Lin Wang. 2024. λGrapher: A Resource-Efficient Serverless System for GNN Serving through Graph Sharing. In *ACM on Web Conference 2024 (WWW)*. 2826âĂŞ2835.

[37] Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, and Eric Rozner. 2021. Proactive Serverless Function Resource Management. In *Proc. the 2020 Sixth International Workshop on Serverless Computing (WoSC)*. 61âĂŞ66.

[38] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. 2021. Amps-inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency. In *Proc. the 50th International Conference on Parallel Processing (ICPP)*. 1–12.

[39] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. 2020. Inspection and characterization of app file usage in mobile devices. *ACM Transactions on Storage* 16, 4 (2020), 1–25.

[40] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proc. the 2021 International Conference on Management of Data (SIGMOD)*. 857–871.

[41] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[42] Poul-Henning Kamp and Robert NM Watson. 2000. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, Vol. 43. 116.

[43] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proc. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 427–444.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems (NeurIPS)* 25 (2012).

[45] Kevin Lee, Vijay Rao, and William Arnold. 2019. Accelerating Facebook's Infrastructure with Application-Specific Hardware. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/. Accessed: 2024-07-07.

[46] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-Efficient Serverless inference through tensor sharing. In *Proc. 2022 USENIX Annual Technical Conference (USENIX ATC)*.

[47] Zijun Li. [n. d.]. GitHub—lzjzx1122/Pagurus: Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. https://github.com/lzjzx1122/Pagurus/tree/master. [Accessed 26-10-2023].

[48] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through {Inter-Function} Container Sharing. In *Proc. 2022 USENIX Annual Technical Conference (USENIX ATC)*. 69–84.

[49] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proc. the 11th Workshop on Programming Languages and Operating Systems (PLOS)*.

[50] Yushi Liu, Shixuan Sun, Zijun Li, Quan Chen, Sen Gao, Bingsheng He, Chao Li, and Minyi Guo. 2024. FaaSGraph: Enabling Scalable, Efficient, and Cost-Effective Graph Processing with Serverless Computing. In *Proc. the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 385âĂŞ400.

[51] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. {ORION} and the Three Rights: Sizing, Bundling, and Prewarming for Serverless {DAGs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[52] Microsoft. 2023. Azure Functions warmup trigger. https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-warmup?tabs=isolated-process%2Cnodejs-v4&pivots=programming-language-python Accessed: 2024-06-12.

[53] Microsoft. 2024. Azure Functions Premium plan. https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal Accessed: 2024-07-12.

[54] Microsoft. 2024. Storage considerations for Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/storage-considerations?tabs=azure-cli. Accessed: 2024-07-01.

[55] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[56] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proc. the Sixteenth European Conference on Computer Systems (EuroSys)*. 228–244.

[57] Nuclio. 2024. Nuclio: Serverless Platform for Automated Data Science. https://nuclio.io/ Accessed: 2024-07-12.

[58] NVIDIA Corporation. 2024. NVIDIA Container Toolkit. Software available from NVIDIA. https://github.com/NVIDIA/nvidia-container-toolkit Accessed: 2024-05-30.

[59] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proc. the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*.

[60] Yi Ouyang, Bin Guo, Qianru Wang, Yunji Liang, and Zhiwen Yu. 2022. Learning dynamic app usage graph for next mobile app recommendation. *IEEE Transactions on mobile Computing* (2022).

[61] Li Pan, Lin Wang, Shutong Chen, and Fangming Liu. 2022. Retention-Aware Container Caching for Serverless Edge Computing. *Proc. of IEEE Conference on Computer Communications (INFOCOM)* (2022).

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8024–8035.

[63] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proc. the ACM Symposium on Cloud Computing (SoCC)*. 324–340.

[64] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proc. 16th USENIX symposium on networked systems design and implementation (NSDI)*. 193–206.

[65] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache For Serverless Applications. In *Proc. the ACM symposium on cloud computing (SoCC)*. 122–137.

[66] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. {INFaaS}: Automated Model-less Inference Serving. In *Proc. 2021 USENIX Annual Technical Conference (USENIX ATC)*. 397–411.

[67] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[68] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming Serverless Functions Better With Heterogeneity. In *Proc. the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 753–767.

[69] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory Deduplication for Serverless Computing with Medes. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*.

[70] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with medes. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*. 714–729.

[71] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2022. FUSIONIZE: Improving Serverless Application Performance Through Feedback-driven Function Fusion. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*.

[72] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. 2020 USENIX Annual Technical Conference (USENIX ATC)*. 205–218.

[73] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proc. the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 121–135.

[74] Umair Siddiqi, Timothy Martin, Sam Van Den Eijnden, Ahmed Shamli, Gary Grewal, Sadiq Sait, and Shawki Areibi. 2022. Faster fpga routing by forecasting and pre-loading congestion information. In *Proc. the 2022 ACM/IEEE Workshop on Machine Learning for CAD*. 15–20.

[75] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proc. the 21st International Middleware Conference (Middleware)*. 1–13.

[76] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556* (2014).

[77] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. SpecFaaS: Accelerating Serverless Applications with Speculative Function Execution. In *Proc. 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 814–827.

[78] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Proc. the IEEE conference on computer vision and pattern recognition (CVPR)*. 1–9.

[79] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proc. the IEEE conference on computer vision and pattern recognition (CVPR)*. 2818–2826.

[80] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proc. the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[81] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. 2020. {InfiniCache}: Exploiting Ephemeral Serverless Functions to Build a {Cost-Effective} Memory Cache. In *Proc. 18th USENIX conference on file and storage technologies (FAST)*. 267–281.

[82] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proc. the Seventeenth European Conference on Computer Systems (EuroSys)*. 1–16.

[83] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast {RDMA-codesigned} Remote Fork for Serverless Computing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 497–517.

[84] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proc. the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 768–781.

[85] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proc. the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 335–350.

[86] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2023. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proc. the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. 181–194.

[87] Hanfei Yu, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. 2022. Accelerating serverless computing by harvesting idle resources. In *Proc. the ACM Web Conference (WWW)*. 1741–1751.

[88] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, not the Function: Rethinking Function Orchestration in Serverless Computing. In *Proc. 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1489–1504.

[89] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *arXiv preprint arXiv:2306.03622* (2023).

[90] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proc. the 11th ACM Symposium on Cloud Computing (SoCC)*. 30–44.

[91] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. {MArk}: Exploiting Cloud Services for {Cost-Effective},{SLO-Aware} Machine Learning Inference Serving. In *Proc. 2019 USENIX Annual Technical Conference (USENIX ATC)*. 1049–1062.

[92] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*.

[93] Yi Zhou, Shubbhi Taneja, Chaowei Zhang, and Xiao Qin. 2018. GreenDB: Energy-efficient prefetching and caching in database clusters. *IEEE Transactions on Parallel and Distributed Systems* 30, 5 (2018), 1091–1104.

[94] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. 2022. Aquatope: Qos-and-Uncertainty-Aware Resource Management for Multi-Stage Serverless Workflows. In *Proc. the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1–14.