Accelerating ML Inference via Opportunistic Pre-Loading on Serverless Clusters

Yifan Sui, Hanfei Yu, Yitao Hu, *Member, IEEE*, Jianxun Li, *Senior Member, IEEE*,
and Hao Wang, *Member, IEEE*

***Abstract*—Serverless computing has emerged as a novel paradigm in cloud computing, characterized by its agile scalability, cost-effective pay-as-you-go billing, and user-friendly capabilities for Machine Learning (ML) inference tasks. Developers wrap their ML algorithms into serverless functions and run them in containers. However, the well-known cold-start problem significantly slows down the response time of functions. To address cold-starts, the technique of pre-warming, which proactively maintains containers in a warm state, has gained widespread adoption across both research and industry. Nevertheless, we observed that pre-warming does not address the distinct delays caused by the loading of ML artifacts. According to our analysis, in ML inference functions, the time required to load libraries and models significantly exceeds the time needed to warm containers. Thus, relying solely on pre-warming is insufficient for mitigating cold-starts. This paper presents *Tyche*, an opportunistic pre-loading approach designed to eliminate the latency associated with loading ML artifacts, enabling near-instant inference and minimizing function execution time. *Tyche* fully leverages the idle memory in warmed containers and GPUs to pre-load required libraries and models, striking an optimal balance between acceleration and resource efficiency. Additionally, *Tyche* is tailored for large-scale serverless platforms, incorporating cluster-wide scheduling and lightweight locality-aware load balancing to enhance performance. We design *Tyche* to be transparent to providers and compatible with existing pre-warming solutions. Experiments on OpenWhisk with real-world workloads show that *Tyche* reduces up to 93% loading latency and achieves up to 8× speedup compared to state-of-the-art pre-warming solutions. Compared with the state-of-the-art serverless pre-loading solution, *Tyche* also achieves up to 1.9× speedup.**

***Index Terms*—Serverless computing, cloud computing, cold start, machine learning.**

Received 19 December 2024; revised 27 October 2025; accepted 16 November 2025. Date of publication 28 November 2025; date of current version 22 December 2025. 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 under Grant 2527416 and Grant 2534241, and in part by the AWS Cloud Credit for Research program. An earlier version of this paper was presented in part at the 2024 ACM Symposium on Cloud Computing [DOI: 10.1145/3698038.3698509]. Recommended for acceptance by O. Ozkasap. *(Corresponding author: Jianxun Li.)*

Yifan Sui and Jianxun Li are with the Departments of Automation of the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: suiyifan@sjtu.edu.cn; lijx@sjtu.edu.cn).

Hanfei Yu and Hao Wang are with the Department of Electrical and Computer Engineering, Stevens Institute of Technology, Hoboken, NJ 07030 USA (e-mail: hyu42@stevens.edu; hwang9@stevens.edu).

Yitao Hu is with the Tianjin Key Laboratory of Advanced Networking of the Department of Intelligence and Computing, Tianjin University, Tianjin 300350, China (e-mail: yitao@tju.edu.cn).

Digital Object Identifier 10.1109/TPDS.2025.3638428

I. Introduction

With the growing adoption of machine learning (ML) applications, such as image recognition and large language models (LLMs), the demand for computational resources to support these applications is booming—Facebook alone serves over 200 trillion inference queries daily [2]. This rapid growth necessitates the development of computing architectures that are both performance- and cost-efficient to handle large-scale ML inference workloads. Serverless computing, a modern cloud paradigm, has become increasingly popular for serving ML inferences due to its flexibility in scaling, cost-effectiveness with pay-as-you-go pricing, and ease of deployment. Consequently, numerous ML inference solutions from both academia and industry have transitioned to serverless architectures, including Amazon Alexa [3], Azure RAG Chatbot [4], Nuclio [5], and ServerlessLLM [6].

In serverless platforms, ML inference applications are packaged as lightweight serverless functions, which are invoked on demand and executed in containers.[1] When an invocation occurs without any pre-initialized (known as "warmed") containers available, the system must initiate a new container from scratch, resulting in what is known as *cold-starts* [7]. To address this issue, extensive research has been proposed on mitigating cold-start [8], [9], [10], [11], [12], [13], [14], [15]. The most widely adopted solution is known as "*pre-warming*" [11], [13], [14], [15], which involves initializing containers and setting up their runtime environments in advance, then keeping these containers alive after serving requests.[2] By maintaining warmed containers, pre-warming methods can effectively avoid the container initialization latency.

The cold-start of a serverless function involves three distinct phases: 1) container warming, 2) dependency loading (e.g., Python libraries), and 3) query processing. Taking an inference function as an example, Fig. 2 shows the operations performed in each phase. For general serverless workloads, container warming represents the primary cold-start bottleneck, while the overhead from loading dependencies is negligible. Consequently, pre-warming strategies are well-suited for these workloads. However, our investigation reveals that for ML inference

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

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

1045-9219 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies.
Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

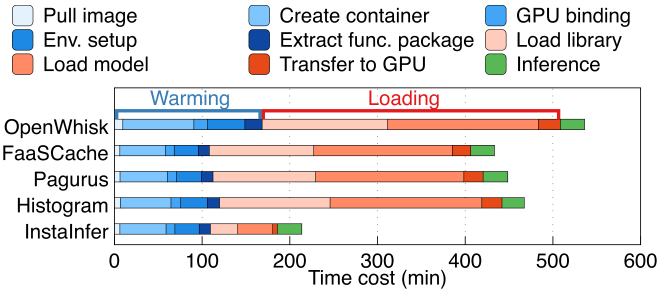
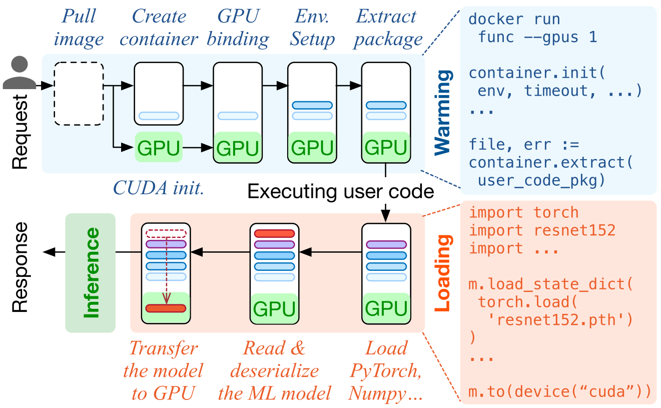


Fig. 1. Cumulative time cost and breakdown of real-world serverless inference invocations driven by Azure traces [14]. The blue bars indicate the container warming stage, and the orange bars indicate the ML artifact loading stage.



```
docker run
 func --gpus 1

container.init(
 env, timeout, ...)
...

file, err :=
container.extract(
 user_code_pkg)
```

```
import torch
import resnet152
import ...

m.load_state_dict(
 torch.load(
  'resnet152.pth')
)
...

m.to(device("cuda"))
```

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

functions, the time required for dependency loading—which falls outside the scope of pre-warming strategies—is notably significant.

Fig. 1 shows a real-world experiment of serving eight popular ML inference functions with invocation patterns following 4-hour industrial traces [14], with state-of-the-art pre-warming methods [13], [14], [15]. To be comprehensive and representative, these methods include all mainstream pre-warming approaches: proactive creation, container caching, and container sharing. We measure the total time cost of each operation in warming, loading, and inference phases on a 4-GPU serverless cluster.[3] Loading the ML artifacts, including large libraries (e.g., PyTorch) and model files (e.g., BERT [16]) 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 [17], [18], [19], allow user-defined warm-up triggers [20], and enable snapshots [21], [22]. However, they cannot completely mitigate the ML artifacts loading stage. Some model loading speedup solutions [17], [18], [19] ignored the library loading, which accounts for over 40% of loading latency, and they heavily reliance on layer similarity across diverse models. The snapshot-based solutions [21], [22] introduces additional disk IO overhead and are incompatible with GPUs due to reliance on Linux's memory mapping. The pre-loading solutions [20], [23], [24] introduced additional constraints and delays and are not compatible with Python runtime.

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.

However, three 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) [14]. Pre-loading libraries and ML artifacts should be lightweight and transparent to avoid incurring any additional overheads. **3) Pre-loading introduces additional scheduling burdens on the cluster.** In large-scale clusters, managing pre-loading on each worker node and directing requests to pre-loaded containers adds significant overhead to the scheduling process, increasing the response latency.

This paper proposes *Tyche*,[4] 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, *Tyche* pre-loads functions only in existing warmed containers and GPU instances created by the platform, rather than proactively reserving memory.[5] To consistently provide cluster-wide optimal function acceleration, we proposes a scheduling policy that can efficiently utilize idle resources by dynamically loading and offloading functions from both containers and GPUs. Besides, *Tyche* is compatible with existing pre-warming and keep-alive schemes by avoiding interfering with the container creation or removal policies.

We summarize *Tyche*'s key contributions as follows:

- We observe the bottleneck of loading ML artifacts in serverless clusters and propose the opportunistic ML model pre-loading technique to minimize function startup latency for the whole workload.
- We design a pre-loading scheduling policy that can make optimal pre-loading decisions across the entire cluster with minimal scheduling overhead, even in large-scale clusters. The scheduling policy is compatible with existing pre-warming solutions.
- We implement *Tyche* atop OpenWhisk, deploy it on an AWS EC2 cluster, and evaluate it using industrial traces and popular inference functions. Extensive experiments show that *Tyche* reduces the end-to-end function latency by 87% compared to state-of-the-art pre-warming and pre-loading solutions.

[3] We follow the same experimental setup in Section VII-C.

[4] The goddess of opportunities in Greek religion.

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

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

II. Motivation and Background

A. 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 [25] 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 binding 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 into CPU and GPU memory. Specifically, each library undergoes an 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.

Scheduling: Besides the three stages mentioned above, another implicit yet non-negligible stage not shown in Fig. 2 is the scheduling stage. In serverless clusters, to avoid cold starts, the scheduler typically traverses a large number of worker nodes to locate a warmed container corresponding to the invocation.

B. *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 [11], [12], [13], [14], [15], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35] and sharing [9], [13], [35], [36], [37], [38], [39], snapshotting [8], [21], [22], [30], [40], [41], and virtualization refactoring [8], [10], [22], [40], [42], [43], [44]. 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.

C. The Motivation of Pre-Loading

To demonstrate that pre-warming alone is insufficient for eliminating inference functions' cold-starts, we select the eight most popular ML models based on GitHub popularity. We conduct an experiment using real-world workloads driven by 4-hour industrial invocation traces from Azure [14]. Four NVIDIA A10 GPUs are used for inference. The Azure trace records the timing and frequency of real-world function invocations. We sweep the trace and randomly select eight function traces to build the workload, mapping each to one benchmark function. The detailed experimental setup is described in Section VII-B.

We implement OpenWhisk's default keep-alive policy and three state-of-the-art pre-warming methods, including Histogram [14], FaaSCache [15], and Pagurus [13], inside OpenWhisk as baselines. These strategies are compared against our proposed method, *Tyche*, 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 warming latency over OpenWhisk. However, loading ML artifacts dominates overall latency 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, *Tyche* reduces the time for the entire workload by over 55% .

Although the loading stage can be accelerated through snapshot [21], [22], compressed memory [45], and RDMA [46] to minimize I/O overhead, these methods cannot enhance library initialization and model deserialization stages, making them insufficient for accelerating inference functions.

The most relevant prior work, InstaInfer [1], proposed opportunistic pre-loading but suffers from decentralized scheduling where each worker makes independent decisions assuming uniform request distribution. This prevents globally optimal resource allocation. Tyche addresses this with centralized, cluster-wide scheduling that maintains a global view of idle resources and invocation patterns. To avoid centralization bottlenecks, Tyche employs consistent-hashing-based load balancing and greedy bin-packing for fast placement decisions, making it both faster and smarter than InstaInfer at scale.

D. 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

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

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 [14], [27], [47], [48], [49], [50], [51], [52] 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 [11], [14], [53]. 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 [54], numerous studies [14], [27], [47], [51], [52] have demonstrated that for optimal execution speed and handling peak workload, inference functions tend to over-provision memory to hold the libraries and models. Therefore, the vast memory gap between containers' running and idle states presents another opportunity for our *opportunistic pre-loading*.

III. An Overview of *Tyche*

A. *Objectives & Challenges*

Tyche 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 wastage 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 [14]. Thus, we must design the pre-loading process in a lightweight and transparent manner to avoid any extra function startup overheads.

How to minimize scheduling latency in large-scale clusters? In large-scale clusters, both managing pre-loading on each node and directing requests to the pre-loaded containers adds non-negligible scheduling overhead. Therefore, we should design a lightweight pre-loading scheduling and routing policy to minimize latency overhead in large-scale clusters.

B. Tyche*'s System Architecture*

We introduce the design of *Tyche*, 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. *Tyche* includes four principal components: Proactive Pre-Loader, Pre-Loading Scheduler, Pre-Loading Agent, and Intra-Container Manager.

The Proactive Pre-Loader handles prediction and request routing, determining *when* to pre-load functions. The Pre-Loading Scheduler makes strategic decisions about *where* to place pre-loaded functions across nodes. The Pre-Loading Agent executes these decisions on worker nodes, focusing on *what* functions to pre-load. Finally, the Intra-Container Manager oversees the actual function loading operations and determines *how* to allocate resources across the cluster. We introduce each component's functionality as follow:

Proactive Pre-Loader forecasts 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 optimal container.

Pre-Loading Scheduler makes centralized decisions on function pre-loading or off-loadings. The decision is based on both the prediction result and container availability. To optimize acceleration over time, it adjusts pre-loading and off-loading based on container lifecycle events driven by platform pre-warming policies.

Pre-Loading Agent on each worker node interfaces with the Pre-Loading Scheduler, sends commands to the Intra-Container Manager based on scheduling decisions, and reports container status changes on its worker node.

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

C. Tyche*'s Workflow*

Fig. 3 shows the workflow and architecture of *Tyche:* Before and upon the arrival of an ML inference function invocation, *Tyche* 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 (Step① in Fig. 3), and passes the prediction to the Pre-Loading Scheduler (Step②).

Stage 2: Concurrently, in the background, the platform's cold-start manager interacts with each worker node's cold start agent to control the creation and removal of containers based on the pre-warming mechanism (Step②③).

Stage 3: Based on the prediction and each worker node's idle resources, the Pre-Loading Scheduler determines which function to pre-load on each worker node's pre-warmed containers and GPUs, and passes the decision to the Pre-Loading Agent (Step④). The Pre-Loading Agent then loads each function in

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

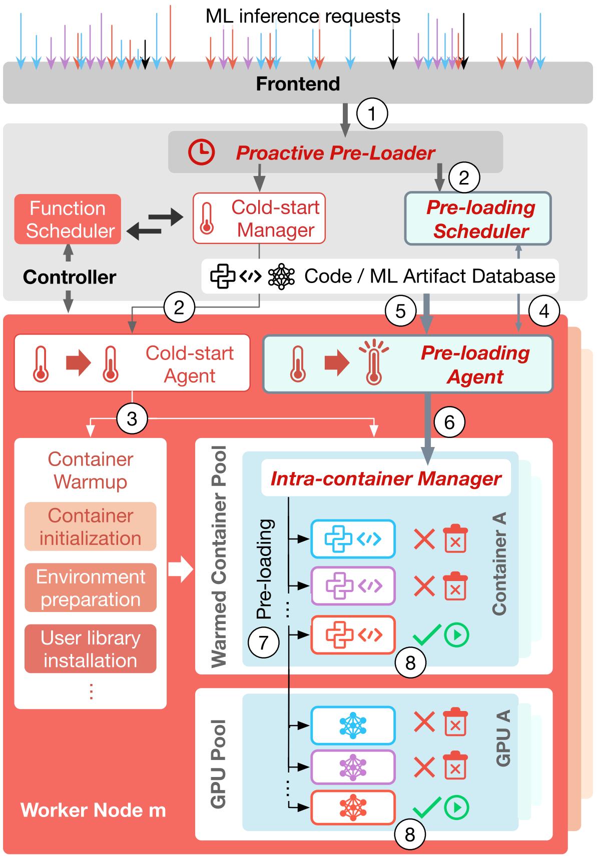


Fig. 3. System overview. Boxes with *red bold italic* names are new components introduced by *Tyche*.

a suitable idle container, extracting the function's code, and unzipping ML artifacts from the platform's database (Step⑤).

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 Agent selects an idle container that pre-loads the function and an idle GPU that pre-loads the function's model and kernel. The request is then sent to the corresponding container's Intra-Container Manager (Step⑥).

Stage 5: Once receiving the request, the Intra-Container Manager immediately calls the corresponding function's pre-loading process (Step⑦) and off-loads all other pre-loaded function states (Step⑧). 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.

IV. 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.

A. 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 *Tyche* 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 [13], [14], ARIMA in [14], Poisson Distribution in [35], Variable Order Markov Model in [33]). *Tyche* employs the platform's inherent prediction model to maintain transparency for serverless providers, avoiding introducing extra operational costs such as building new models.

B. 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 $P_{offload}(f)$, *Tyche* 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 [14], [55], 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 [35] 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

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

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

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 VII-L.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Consistent-Hashing Based Load Balancing.

```
   Input: Function $f$'s Request rate $\lambda_A$, execution
          duration $T_A$, resource cost $R_A$, nodes
          $s_1, s_2, \ldots, s_N$
   Output: Proper server selection for function A
1  Procedure
     SelectServerForFunction($\lambda_A, T_A, R_A, \{s_i\}$):
2      $S \leftarrow$ ConsistentHashingNodes($A, \{s_i\}$) // Get
        home server set
3      $S' \leftarrow$ SelectSubset($S, \lambda_A, T_A, R_A$)
4      $selected\_server \leftarrow$ null
5      $max\_capacity \leftarrow -\infty$
6      foreach $s \in S'$ do
7          if $s.Loaded(f) \& (s.Capacity() > R_A)$ then
8              if $s.GPU.Loaded(f)$ then
9                  $selected\_server \leftarrow s$;
10                 break
11             if $s.Capacity() > max\_capacity$ then
12                 $selected\_server \leftarrow s$;
                    $max\_capacity \leftarrow s$.Capacity();
13     return $selected\_server$
14 Function SelectSubset($S, \lambda_A, T_A, R_A$):
15     $S', resource\_sum \leftarrow 0$
16     $required\_resources \leftarrow \lambda_A \times T_A \times R_A$
17     foreach $s \in S$ do
18         if $resource\_sum \leq required\_resources$ then
19             $S'$.add($s$)
20             $resource\_sum \leftarrow$
                $resource\_sum + s$.Capacity()
21         else break
22     return $S'$
```

C. Consistent Hashing Function Balancing

In the large-scale cluster that contains many worker nodes, while each node contains numerous idle containers and GPUs, to minimize the overall E2E latency, we face a new challenge.

When a request arrives, instead of randomly routes the request to a worker node, the Proactive Pre-Loader should route the request to the container that has the optimal acceleration. Therefore, the most straightforward method is to let the Pre-Loader traverse all worker nodes and search all containers' pre-loaded functions. After finding all containers that have pre-loaded the function, due to the limited capacity of GPUs, only part of containers can transfer the model to GPU to achieve the best acceleration performance. Thus, the Pre-Loader must choose an optimal one from all these containers.

Although this solution is practical and efficient in single-node clusters, however, in the large-scale cluster that contains many numerous containers and pre-loaded functions, traversing all of them introduces high scheduling delay. In *Tyche*, we propose a lightweight function load balancing policy based on consistent hashing that can make a balance between maximizing inference acceleration and minimizing scheduling delay.

As most scheduling delay is caused by traversing all worker nodes' containers, the Proactive Pre-Loader only selects a small group of worker nodes and select the optimal container from these nodes. To avoid the situation that the non-selected nodes contain the optimal container, we need to make sure that for each inference function, only a group of worker nodes will pre-load it. Thus, we design a consistent-hashing based load balancing policy to achieve this goal.

Consistent hashing is primarily used to distribute requests evenly across a changing set of nodes. In *Tyche*, each function's request is mapped to a "home" worker node based on its hashing value. This mapping ensures that requests for the same function are directed to the same node. If the "home" node is overloaded or lacks the pre-loaded function, the Proactive Pre-Loader will determine the next node in sequence, following the hashing circle, until an available node with the requisite pre-loaded function is found. Therefore, even if there are numerous worker nodes or the node number are frequently changing, this method minimizes the changes in node selection, providing stability and small scheduling delay.

We further explain the load balancing policy in Algorithm 1. When request arrives, first, the Pre-Loader get the function's arrival rate, execution time, and resource cost information. Next, it uses consistent hashing to preliminarily select worker node set S. Next, to further reduce traversing latency, we select a subset S' from S (line 14). We estimate the overall resource cost of the function's all invocations (line 17) and make sure that S' contains enough remaining resource to serve all those functions.

Subsequently, we traverse all worker nodes' containers in S' to get the optimal container (line 6). If the GPU binding to a container has pre-loaded the model data, this container can provide the maximum acceleration. The Pre-Loader chooses this container directly (line 8-10). Otherwise, if none of GPU in S' has loaded the model, from all containers that has pre-loaded the function in CPU memory, the Pre-Loader choose the one whose worker node contains most remaining resource (line 11-13).

V. Pre-Loading Scheduling

The pre-loading scheduling contains two components, a centralized Pre-Loading Scheduler and a distributed Pre-Loading Agent. The Pre-Loading Scheduler 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. The Pre-Loading Agent uses the Pre-Loading Scheduler's decision to operates pre-loading and off-loading in each worker node.

A. 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

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

until its idle memory runs out while their models share the same GPU.

To select an appropriate container for each function to preload, 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.

As the Multiple knapsack Bin-Packing problem has already been proven NP-Hard [56], to get the optimal bin-packing assignments, we must exhaust a huge number of possible conditions, even with the help of efficient algorithms like Dynamic Programming. Although optimal, in multi-node cluster where there are numerous functions and containers, the high complexity of exhaustion prevents efficient pre-loading scheduling.

Thus, we propose a greedy-based bin-packing policy to make a balance between scheduling efficiency and acceleration performance. To reduce the complexity, we first sort functions by their value-to-weight ratio, and then attempting to place each function in the container where the container has enough capacity and where the function adds the most value relative to the container's remaining capacity. Through this policy, we can balance the speed and accuracy in scheduling. By sorting functions and dynamically adapting to container capacities, it minimizes the computation overhead while effectively optimizing resource utilization. Consequently, *Tyche* can pre-load more functions in the limited idle containers.

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.

B. Cluster-Wide Scheduling

As the scheduling decision is based on each function's request arrival probability, if the Pre-Loading Scheduler runs independently on each worker node, it will face an additional challenge that accurately estimate the request arrival distribution on each worker node.

According to *Tyche*'s routing policy, instead of equally (randomly) routing the request to each worker node, the Proactive Pre-Loader priorly chooses from the node that already pre-loaded the function and has the most remaining resource. Therefore, due to the uneven distribution, it's both difficult and time-costing to accurately estimate each node's request arrival probability.

To tackle this challenge, we design a cluster-wide Pre-Loading Scheduler. It takes into account the overall probability of incoming requests to the platform, rather than the probability of requests arriving at each individual worker node. Additionally, it comprehensively considers the availability of all idle containers and GPUs on every worker node to make globally optimized bin-packing decisions. While the Pre-Loading Agent, which runs independently on each worker node, only receives the Pre-Loading Scheduler's decisions and operates the pre-loading and off-loading.

C. 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 the cluster-wide optimal acceleration over time as follows.

As shown in Fig. 4, Functions A, B, and D are pre-loaded on worker 1's containers, while models of Function A and D are transferred to GPU (For simplicity, Fig. 4 only shows worker 1's container pool and GPU pool). In the first case at $t1$, when Function A's invocation arrives, as Function A is loaded on worker 1, the scheduler first forwards the request to worker 1's 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 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. As worker 2 has idle spaces, Function B, D are subsequently pre-loaded on worker 2. The fourth case is the container creations. In $t5$, once detecting a new idle container is available, the scheduler pre-loads 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.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

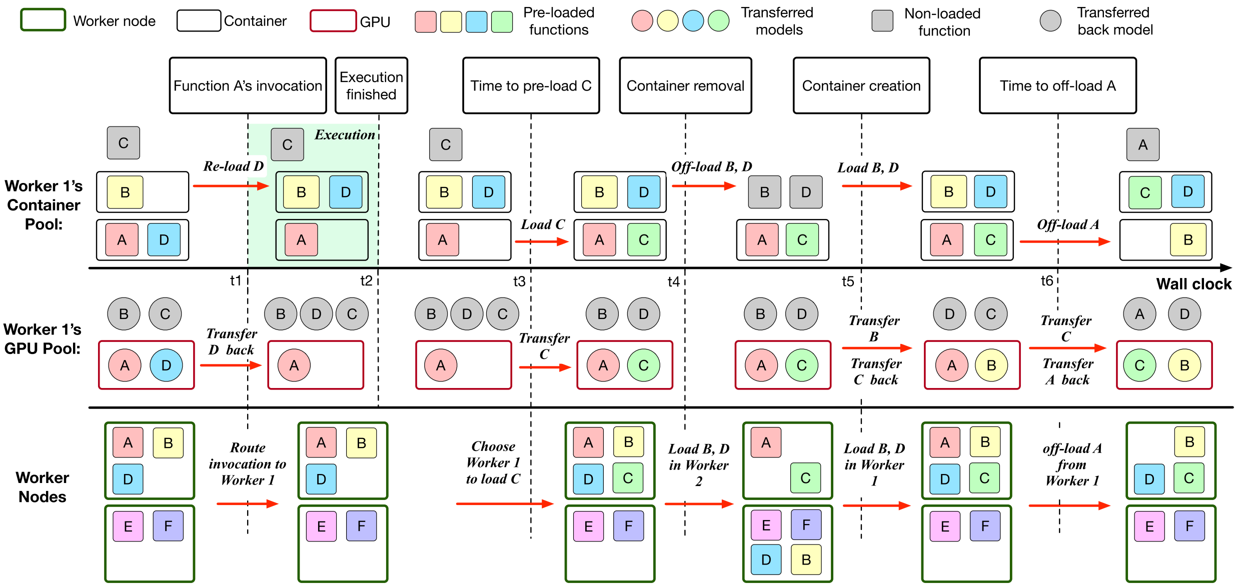


Fig. 4. The scheduler's operation after detecting each event.

VI. 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.

A. Pre-Loading Management

As each container holds multiple functions' 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.

We describe manager's workflow by an example. 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 *Tyche*. We offer two modification options with different objectives:

Code snippet 1: How to use *Tyche*

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

# Tyche
model = Tyche.load(model_path)
sys.stdin.readline() # wait for request
inference()...
```

Maximum transparency: As the following Python code snippet 1 shows, developers only need to modify two lines of code: First, replace the model loading line (*torch.load*) with the *Tyche* 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 [20] to hold the pre-loading content. The manager calls the *LOAD* API to perform pre-loading without accessing any function-specific data.

B. 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. *Tyche* 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, when a function'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.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

Meanwhile, the isolation is enhanced with jail technique [57]. 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 preload one function, *Tyche* still significantly outperforms existing pre-warming methods (Section VII-K).

VII. Evaluation

A. Implementation

We implement a prototype of *Tyche* using Apache OpenWhisk [53]. We implement the Proactive Pre-Loader, Pre-Loading Scheduler, and Pre-Loading Agent as OpenWhisk components using 4 K lines of Scala code and implement the Intra-Container Manager in each container's proxy using 2 K lines of Golang code. We use PyTorch as the ML environment, although *Tyche* 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: We implement the Pre-Loading Scheduler in OpenWhisk's load balancer module. Thereby it can receive the prediction result directly from the Proactive Pre-Loader. It interacts with the Pre-Loading Agent to get other information by a Redis distributed database.

Pre-Loading Agent: OpenWhisk runs a container pool module in each node to manage each container's creation and removal. We implement the agent in this module so that the agent can acquire all the information it needs for pre-loading. The agent sends loading and off-loading command 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 Pre-Loading Agent. We modify the Executor module to execute loading and off-loading. Each pre-loaded function runs as an independent Python process.

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

B. Experiment Settings

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

Testbed: We evaluate *Tyche* on four 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 CPU 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. 4) **Multi-node GPU clusters** that includes one controller node and 4/8/16 worker nodes. These four clusters are allocated with the same overall resources (256 CPU cores, 1024 GB of memory, and 16 NVIDIA A10 GPUs). We perform the pre-loading scheduling optimization evaluation and scheduling overhead evaluation on this cluster.

Workloads: We select the inference function of SeBS benchmark [25] 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 [23]. 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 [14], 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 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 [59], Inception_V3 [60], ResNet18 [61], ResNet50, ResNet152, VGG19 [62], GoogleNet [63], and Bert-Base [16]. The model size varies from 45 MB to 549 MB, providing sufficient diversity for evaluating *Tyche*'s efficiency. We expand the function type to 1000 for further evaluation in Section VII-I.

Tyche+$*$ Settings: As *Tyche* can be easily integrated with pre-warming solutions, Tyche+$*$ indicates integration with three solutions: Histogram [14], Pagurus [13], and FaaSCache [15]. *Tyche* pre-loads functions in the warmed containers created by these solutions.

Baselines: We compare *Tyche* with the state-of-the-art baselines that mitigate cold-starts in serverless computing: 1) **OpenWhisk** [53], 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

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

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 in OpenWhisk. 4) **Pagurus** avoids cold start by "lending" other functions' idle containers to the function being invoked.[6] 5) **REAP** [21] is a snapshot-based cold start mitigation method that stores function completion states as snapshots on disk. 6) **Azure Function with warmup trigger** [20] allows pre-loading user-defined content while scaling up new instances. 7) **InstaInfer** [1] is a serverless pre-loading solution for accelerating inference functions.[7]

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 & Monetary cost:** the platform's CPU and GPU memory consumption and the overall monetary cost for running the whole workload. **6) Scheduling overhead:** The overall scheduling time cost for each invocation is calculated as the E2E latency minus the function execution time and the warming latency.

C. Reducing E2E Latency

We evaluate TYCHE+∗ and baselines on the single-node cluster. Fig. 5 shows that integrating TYCHE 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 TYCHE effectively mitigates the latency with library and model pre-loading.

The Azure Function baseline utilizes the warmup trigger [20] 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 [65], 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 *Tyche*.

Fig. 5 shows that *Tyche* 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 *Tyche*: 1) The function's library files are stored on Azure Files. During loading, reading many small files incurs heavy overhead (over 10 seconds). 2) Warmup triggers only work during 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 (Section VII-D).

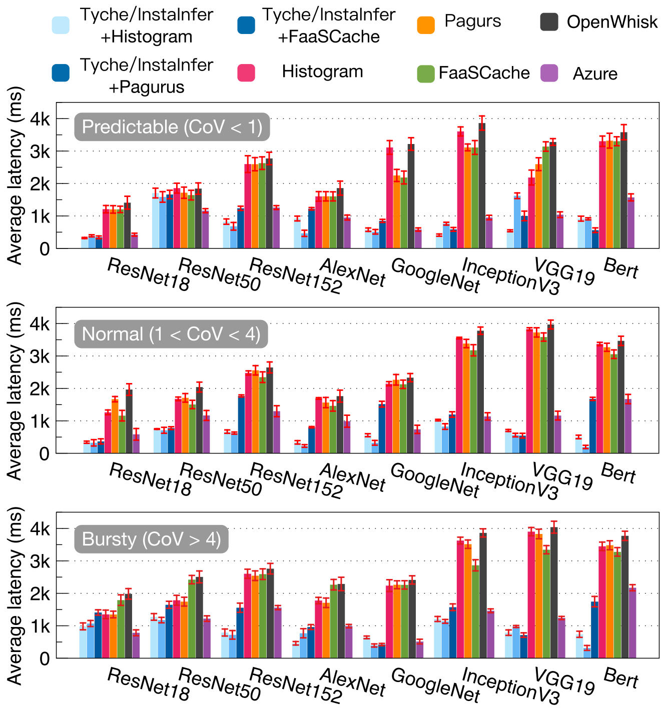


Fig. 5. Average E2E latency of TYCHE/INSTAINFER+∗ and baselines running the Predictable, Normal, and Bursty workloads.

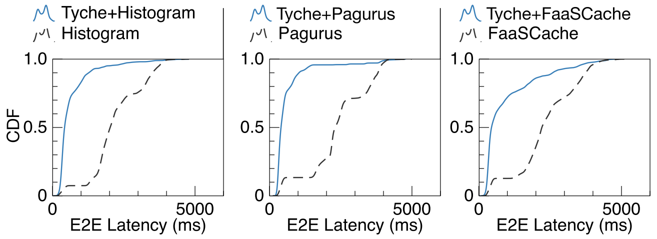


Fig. 6. CDF of TYCHE+∗ and baselines running the Normal workload.

Table I presents the average E2E latency, warming+loading latency, speedup, and pre-load rate of each baseline. TYCHE+∗ outperforms each corresponding baseline on each metric. TYCHE+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 *Tyche* and each baselines in Fig. 6. The results show that *Tyche* can effectively accelerate the workload without increasing the tail latency.

To show *Tyche*'s acceleration effect more intuitively, we present a time breakdown of the E2E latency of Pagurus and TYCHE+Pagurus running a "Normal" workload in Fig. 7. Pagurus is selected in this case since it outperforms Histogram and

[6] Pagurus's original implementation [64] is not for OpenWhisk. We reproduced Pagurus in OpenWhisk and tuned its performance to the best for a fair comparison.

[7] As Tyche extends InstaInfer to improve performance in large-scale clusters, we only evaluate InstaInfer in multi-node evaluations.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

TABLE I
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
TYCHE+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)	-	-	-
TYCHE+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)	-	-	-
TYCHE+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)	-	-	-

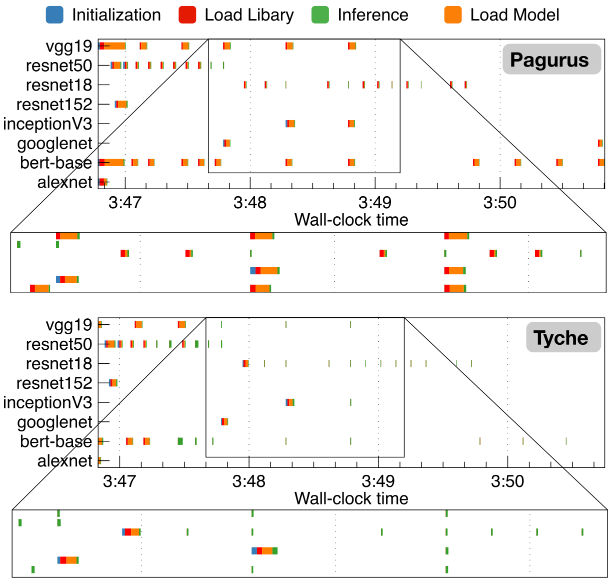


Fig. 7. E2E latency breakdown of individual invocations served by Pagurus and TYCHE+Pagurus.

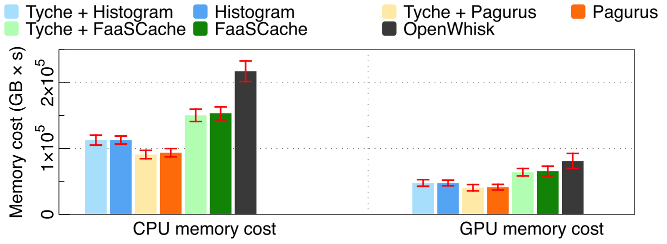


Fig. 8. Average memory cost of TYCHE+* and baselines running the same workload.

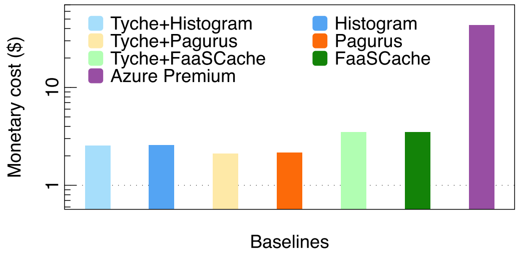


Fig. 9. Monetary cost of TYCHE+* and other baselines running the same workload.

FaaSCache. Fig. 7 shows that TYCHE+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. 7, 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.

D. Memory and Monetary Cost

We evaluate the monetary cost of *Tyche*, baseline pre-warming methods, and naive pre-loading while running the same Azure trace workload. In the evaluation, *Tyche* 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 *Tyche*, more containers are created proactively for pre-loading. Shown in Fig. 8, the container and GPU memory consumption of TYCHE+* are nearly identical to those of corresponding baselines alone. That's because *Tyche* only reuses the idle container created by the baseline method and does not proactively create new containers. Consequently, *Tyche* does not incur additional resource costs. In contrast, to achieve comparable acceleration performance, OpenWhisk Pre-loading creates more containers than *Tyche*, resulting in at most 2.4× the memory cost and 2× the GPU cost compared to *Tyche*.

Then we evaluate the monetary cost of running the above 4-hour workload using Azure Function pricing model [66]. Fig. 9 shows that the monetary cost of TYCHE+* is nearly identical to that of the corresponding baseline alone. Although Azure Premium Plan achieves lower E2E latency for several functions according to Fig. 5 than *Tyche*, its expense is 20 times higher than other methods.

E. Multi-Node Evaluation

We evaluate the scalability of *Tyche* by conducting experiments on the multi-node cluster. We evaluate the E2E latency using the same benchmarks, metrics, baselines, and workloads from Section VII-C. Fig. 10 shows that integrating *Tyche* with baselines reduces up to 90% E2E latency. The performance

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

TABLE II
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
TYCHE+Histogram	467 (6.7)	624 (5.6)	768 (4.5)	302 (9.5)	429 (7.6)	558 (5.8)	80	67	53
Histogram	2703 (1.16)	2729 (1.24)	2861 (1.28)	2452 (1.17)	2480 (1.3)	2614 (1.23)	-	-	-
TYCHE+Pagurus	392 (7.7)	589 (7.9)	542 (6.4)	186 (15.5)	266 (12.2)	326 (9.8)	84	81	72
Pagurus	2493 (1.25)	2917 (1.2)	2624 (1.3)	2203 (1.3)	2663 (1.2)	2377 (1.35)	-	-	-
TYCHE+FaaSCache	734 (4.3)	861 (4.1)	917 (3.8)	506(5.7)	616 (5.3)	722 (4.5)	62	48	43
FaaSCache	2526 (1.23)	2751 (1.27)	2723 (1.27)	2289 (1.26)	2508 (1.3)	2476 (1.3)	-	-	-
InstaInfer	821 (3.8)	968 (3.6)	1043 (3.6)	576 (5)	725 (4.5)	811 (4)	61	46	42
OpenWhisk	3124 (N/A)	3496 (N/A)	3459 (N/A)	2879 (N/A)	3247 (N/A)	3216 (N/A)	-	-	-

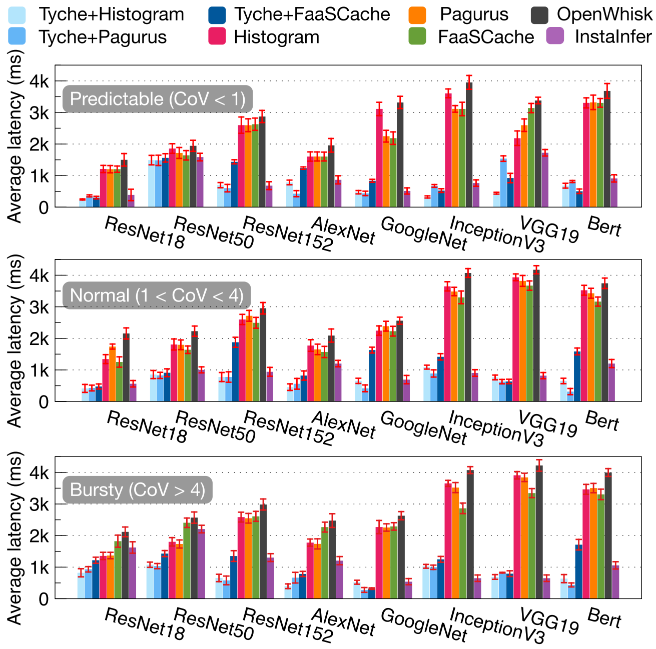
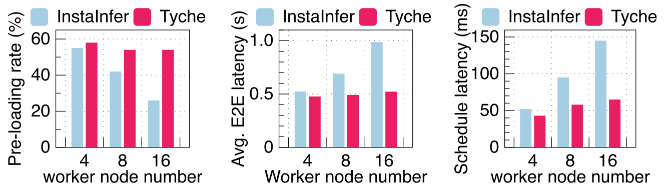


Fig. 10. Average E2E latency of TYCHE+∗ and other baselines running on the multi-node cluster.



(a) Pre-loading rate. (b) E2E latency. (c) Scheduling cost.

Fig. 11. Scheduling performance of TYCHE and INSTAINFER under the same workload.

evaluated on the multi-node cluster is similar to the results observed from the single-node cluster. This consistency suggests that *Tyche* efficiently maintains low loading latency for a variety of workloads in a distributed cluster. Table II details the average E2E latency, warming+loading latency, speedup, and pre-load rate for each baseline. The data shows TYCHE+∗ consistently outperforms existing baselines across all the metrics.

F. Pre-Loading Scheduling Optimization Evaluation

To evaluate whether the centralized Pre-Loading Scheduler + distributed Pre-Loading Agent design can continuously make the optimal bin-packing decision, we compare *Tyche* with InstaInfer, which operates pre-loading scheduling independently on each worker node. Its scheduler assumes that the request's arrival probability to each worker node is equal. Consequently, the scheduler directly utilizes the prediction result from the Proactive Pre-Loader to make bin-packing decisions and reacts to the node's container creation and removal events. We run the same 4-hour workload in three clusters : a 4-node cluster, a 8-node cluster, and a 16-node cluster separately. The total number of vCPU, memory, and GPU in each cluster is the same.

As shown in Fig 11(a) and (b), with the increase of cluster size, *Tyche* maintains a similar pre-loading rate and average E2E latency. In contrast, although the performance of InstaInfer is similar to *Tyche* in the 4-node cluster, when the cluster size increases, the pre-loading rate decreases significantly. Subsequently, its average E2E latency increases. In the 16-node cluster, *Tyche* further accelerates the workload 1.9× over InstaInfer.

This difference in performance is due to InstaInfer assuming that requests are evenly spread across all worker nodes. However, this assumption is incorrect because, in reality, requests are more likely to be sent to nodes that have pre-loaded the necessary functions. As a result, InstaInfer fails to achieve globally optimal scheduling: Some worker nodes end up pre-loading many functions that are never invoked, while others fail to pre-load sufficient function instances to meet all incoming requests. In contrast, the centralized scheduling design of *Tyche* can make decisions based on all worker nodes' idle instances and the overall arrival probability of each function. Therefore, *Tyche* can make decisions more wisely, optimally utilizing the limited resources to achieve a higher pre-loading hit rate and thereby achieves better acceleration.

G. Scheduling Overhead Evaluation

To evaluate whether the consistent hashing-based load balancing policy can reduce scheduling overhead, we compare *Tyche* with InstaInfer. We evaluate the scheduling overhead of the two solutions () in the same setup and workload as Section VII-D.

As shown in Fig. 11(c), *Tyche* benefits from a lightweight load balancing policy that avoids traversing all containers across all worker nodes, accelerates the overall scheduling process up to 2.3×, compared with InstaInfer. This efficiency arises primarily

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

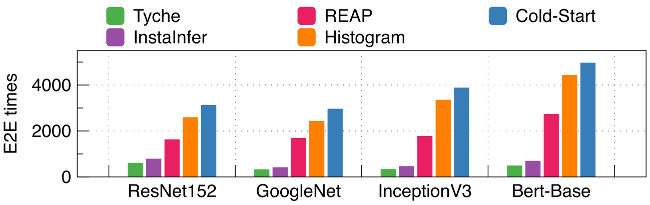


Fig. 12. E2E latency of TYCHE, REAP, and Histogram running benchmark functions with different model sizes.

because, on the one hand, the lightweight scheduling is less complex than that of InstaInfer. On the other hand, since the Proactive Pre-Loader communicates with each Scheduler via the Redis database, the smaller number of traversed nodes requires less information to communicate between the Pre-Loader and each node, significantly reducing the communication overhead.

We evaluated our greedy bin-packing policy against the optimal ILP solution using 1,000 functions and 100 idle containers. Tyche achieved near-optimal performance with only 1.4% optimality loss while delivering a 295,210× speedup, demonstrating effective scalability for large-scale serverless environments.

H. Comparisons With Snapshot Methods

To mitigate cold start, some approaches [21], [22] 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 eliminate the loading delay. Thus, we conduct an evaluation between *Tyche* and REAP [21], a snapshot-based serverless method.

We evaluated the E2E latency of four representative benchmark ML inference functions with small (GoogleNet), medium (Inception_v3), large (ResNet152) and extra-large (Bert-Base) models respectively in *Tyche*, REAP, AsyFunc, and Histogram in the same setup. Fig. 12 shows REAP outperforms Histogram, while *Tyche* further enhances execution by 1.5 to 2.5× over REAP.

The reason for *Tyche* outperforming REAP is that *Tyche* 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 (300–600 ms) due to the large size of model and library files. In contrast, *Tyche* keeps functions in memory and achieves negligible latency (5–14 ms in Section VII-M).

I. Large-Scale Evaluation

To further evaluate the performance of *Tyche* in a more realistic scenario, we extend the workload to 1000 functions on the multi-node cluster. According to Azure [14], 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, ..., ResNet50-125) to create 125 different functions that run the same model. Since *Tyche* treats the function code as black-box, all functions created are uniquely different.

TABLE III
AVERAGE E2E LATENCY IN LARGE-SCALE EVALUATION

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

TABLE IV
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 Tyche+Pagurus, Pagurus, and vanilla OpenWhisk using the same workload. The result is shown in Table III. Besides, we evaluate the pre-loading rate of *Tyche*. 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, *Tyche* can effectively pre-load the frequently invoked functions and accelerate the workload in large-scale scenarios.

J. Prediction Performance Evaluation

To evaluate the robustness of *Tyche*, we choose four prediction models: Poisson distribution, Histogram policy-based prediction [14], Random Forests (RF), and Auto-Regressive Integrated Moving Average (ARIMA) modeling. 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 IV, Poisson achieves the best performance in predictable and normal workloads, whereas Histogram performs best in bursty workloads. *Tyche* pre-loads over 40% of functions and speeds up workloads by over 1.5×.

K. Ablation Study

We conduct an ablation experiment on the single-node cluster to evaluate the effectiveness of the Proactive Pre-Loader and Pre-Loading Scheduling, including both the Pre-Loading Scheduler and Agent. Three variants of *Tyche* are evaluated and compared with Histogram Policy, Pagurus, and FaaSCache:

- *TYCHE_NP: Tyche* 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.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

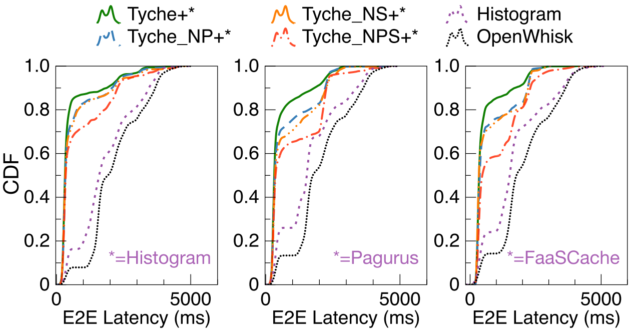


Fig. 13. The CDF of E2E latency for ablation of TYCHE+* and baselines.

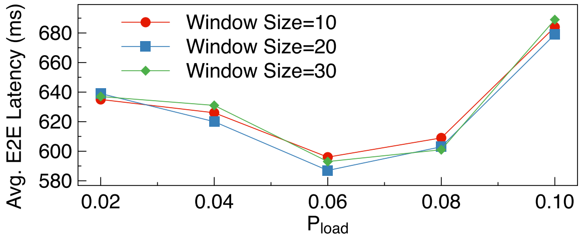


Fig. 14. The average E2E latency with different P_{load} and sliding window size.

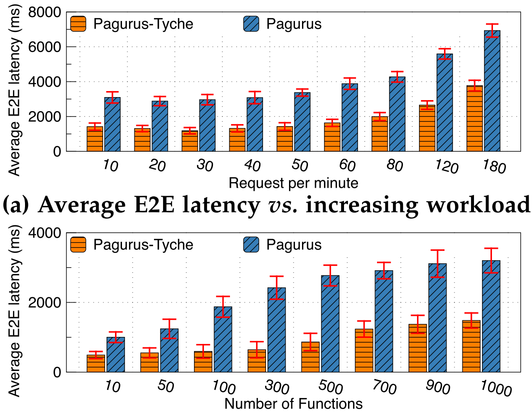
- TYCHE_NS: *Tyche* without Scheduling. This variant cannot make optimal assignments and dynamically schedule loading and unloading. For TYCHE_NS, a function is only pre-loaded under two situations: 1) when receiving the pre-load message from the Proactive Pre-Loader and 2) when a container is idle, its corresponding function will be loaded (i.e., one-to-one mapping).
- TYCHE_NPS: *Tyche* without either the Proactive Pre-Loader or Scheduler. Each container only pre-loads its own function's libraries and models.

Fig. 13 shows the CDF of E2E inference latency under 2-hour "Normal" traces randomly selected from Azure. Regardless of the pre-warming method used, *Tyche* 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, *Tyche* accelerates the workload by 1.16-1.28×, 1.21-1.49×, and 1.48-1.73× when compared with *Tyche*-NP, *Tyche*-NS, and *Tyche*-NPS.

L. Sensitivity Analysis

We conduct an experiment to evaluate the impact of two *Tyche* 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. 14 shows their impact on the average E2E latency of a workload from Azure Trace.



(a) Average E2E latency *vs.* increasing workloads.

(b) Average E2E latency *vs.* increasing function numbers.

Fig. 15. Scalability comparison of *Tyche* and other serverless solutions.

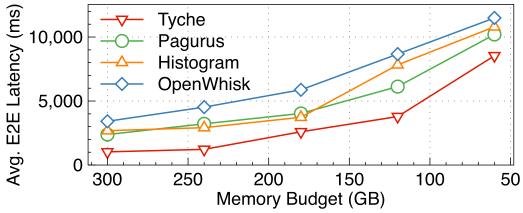


Fig. 16. Robustness to limited memory budgets.

As observed, the performance of *Tyche* 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. Conversely, pre-loading a function too late (high P_{load}) makes requests miss the opportunity of being accelerated. We set *Tyche*'s P_{load} to be 0.06 to achieve the optimal acceleration.

M. Scalability and Overhead

To evaluate the scalability of *Tyche*, TYCHE + Pagurus is given increasingly heavier workloads, varying from 10 to 180 requests per minute. The performance is shown in Fig. 15(a). *Tyche* consistently outperforms Pagurus across different scales. Furthermore, we evaluate *Tyche*'s scalability with increased function numbers. We keep the memory budget and overall request rate unchanged while increasing the number of functions from 10 up to 1000 (The functions are created using the same manner as Section VII-I). As Fig. 15(b) shows, *Tyche* consistently outperforms Pagurus across different functions numbers. Besides, the performance gap widens significantly as the number of functions grows, showing the Pre-Loading Scheduler's effectiveness in selecting the most valuable functions to pre-load.

Then we evaluate the performance of *Tyche* against other baselines under constrained resource budgets by varying the container pool's size. As Fig. 16 shows, *Tyche* consistently

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

outperforms other baselines under different memory budgets, showing stronger robustness. InstaInfer is not compared here because its performance is identical to Tyche's in this test. This experiment evaluates the pre-loading logic under memory pressure, which is shared by both systems.

Next, we report the latency and resource overhead of each component.

Under peak load, *Tyche*'s Proactive Pre-Loader adds 3 ms latency, while the Intra-Container Manager adds 2 ms to 11 ms when clearing memory for an incoming invocation, which is negligible compared to the 1500 ms to 5000 ms saved. Resource overhead is minimal: the Proactive Pre-Loader uses less than 0.3 CPU cores and 72 MB memory; the scheduler uses 0.3 CPU cores and 135 MB; the Intra-Container Manager uses 0.1 CPU cores and 9 MB; and the multiple non-root user security mechanism uses 1 MB memory. Energy overhead is negligible since GPU loading is required for inference regardless of pre-loading. Our in-memory approach avoids the disk I/O overhead of snapshot-based alternatives. A misprediction of unused pre-loaded functions only results in a lost opportunity for further acceleration but does not cause additional resource cost, as *Tyche* only utilized the idle instances' unused memory. Overall, the combined overhead of all *Tyche*'s components is negligible compared to the demands of the workloads.

VIII. Related Work

Serverless inference: Serverless computing has been applied to ML inference [17], [18], [67], [68], [69], [70], [71], but they overlook the substantial loading latency. Some methods increase throughput by batching requests [72], [73], complemented by *Tyche*'s batch handling. AsyFunc [19] pre-loads layers to mitigate bursts but doesn't address cold starts, covering only 52% of ML artifact loading time, as indicated in Fig. 1. Tetris [18] and Optimus [17] share layers between models but miss library and GPU overheads.

Cold-start mitigation: Many studies attempt to address cold-start issues, which can be classified into four major categories: 1) Pre-warming that predictively pre-warms container in advance [12], [14], [27], [32], [33], [34] and keeps them warmed [11], [13], [14], [15], [26], [28], [29], [30], [31]. 2) Virtualization Refactoring [8], [22], [40], [42], [43], [44] that use new virtualization technique to accelerate warming. 3) Container Sharing [9], [13], [37], [38], [39] that shares container among functions. 4) Snapshot based methods [8], [21], [22], [30] that stores snapshots of functions. Among them, pre-warming, virtualization refactoring, and container sharing overlook the unique loading stage for ML inference functions. Snapshot methods store inference states and dependencies, but snapshot loading causes 100-1000 ms startup delays and do not support GPUs.

Pre-loading in serverless: Several works [20], [23], [24] enable user-defined pre-loading during instance startup. Azure warmup trigger [20] pre-loads primitives during scaling but fails for cold starts and pre-warmed containers. AWS Lambda static initialization [23] executes components once per container but cannot pre-load before first invocation. Work [24] executes primitives on pre-warmed containers but underutilizes idle resources and does not support GPUs. InstaInfer [1], closest to *Tyche*, reduces inference latency via opportunistic pre-loading but sacrifices efficiency for accuracy, causing scheduling overhead and suboptimal decisions in large-scale clusters.

Function data caching: Some studies [74], [75] cache ephemeral data of functions in local storage or cloud server, while others [76], [77] keep data in containers. *Tyche* focuses on pre-loading libraries and models into memory, which is orthogonal to these data caching techniques.

IX. Conclusion

This paper proposed *Tyche*, 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 relying on popular cold-start mitigation approaches. *Tyche* comprises a Proactive Pre-Loader to estimate when to load each function and schedules load balancing, a Pre-Loading Scheduler to assign to-be-loaded functions to suitable idle containers and GPUs, a Pre-Loading Agent to operate the pre-loading decisions, and an Intra-Container Manager for controlling the loading and off-loading of each function. Our extensive experimental evaluation demonstrates four key contributions. *Tyche* achieves substantial latency reduction, decreasing end-to-end latency by up to 90% and outperforming snapshotting methods by up to 2.5×. These gains come with remarkable cost-efficiency, requiring negligible extra resource cost while proving to be over 20× more cost-effective than comparable commercial solutions. Our centralized scheduler demonstrates exceptional scalability, achieving up to 1.9× speedup over distributed alternatives in multi-node clusters. Finally, *Tyche* maintains robust large-scale performance, achieving a 73% pre-load hit rate for the most active functions in our 1000-function evaluation.

Acknowledgment

Preliminary results have been presented in the ACM SoCC'24 [1]. 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. 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] 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. 2024 ACM Symp. Cloud Comput.*, 2024, pp. 178–195.

[2] K. Lee, V. Rao, and W. Arnold, "Accelerating Facebook's infrastructure with application-specific hardware," 2019. Accessed: Jul. 07, 2024. [Online]. Available: https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/

[3] "Alexa skills - serverless applications lens," 2023. Accessed: Jul. 07, 2024. [Online]. Available: https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/alexa-skills.html

[4] Azure Samples, "Serverless AI chat with RAG using LangChain.js," 2024. Accessed: Jul. 07, 2024. [Online]. Available: https://learn.microsoft.com/en-us/samples/azure-samples/serverless-chat-langchainjs/serverless-chat-langchainjs/

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

[5] Nuclio, “Nuclio: Serverless platform for automated data science,” 2024. Accessed: Jul. 12, 2024. [Online]. Available: https://nuclio.io/

[6] Y. Fu et al., “ServerlessLLM: Locality-enhanced serverless inference for large language models,” 2024, *arXiv:2401.14351*.

[7] E. Jonas et al., “Cloud programming simplified: A berkeley view on serverless computing,” 2019, *arXiv: 1902.03383*.

[8] D. Du et al., “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 467–481.

[9] I. E. Akkuset al., “{SAND }: Towards { High-Performance} serverless computing,” in *Proc. 2018 Usenix Annu. Tech. Conf.*, 2018, pp. 923–935.

[10] Z. Shen et al., “X-Containers: Breaking down barriers to improve performance and isolation of cloud-native containers,” in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 121–135.

[11] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, “On-demand container loading in AWS Lambda,” in *Proc. 2023 USENIX Annu. Tech. Conf.*, 2023, pp. 315–328.

[12] J. Stojkovic, T. Xu, H. Franke, and J. Torrellas, “SpecFaaS: Accelerating serverless applications with speculative function execution,” in *Proc. 2023 IEEE Int. Symp. High- Perform. Comput. Archit.*, 2023, pp. 814–827.

[13] Z. Li et al., “Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing,” in *Proc. 2022 USENIX Annu. Tech. Conf.*, 2022, pp. 69–84.

[14] M. Shahrad et al., “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *Proc. 2020 USENIX Annu. Tech. Conf.*, 2020, pp. 205–218.

[15] A. Fuerst and P. Sharma, “FaasCache: Keeping serverless computing alive with greedy-dual caching,” in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 386–400.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” 2018, *arXiv: 1810.04805*.

[17] Z. Hong et al., “Optimus: Warming serverless ML inference via inter-function model transformation,” in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 1039–1053.

[18] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, “Tetris: Memory-efficient serverless inference through tensor sharing,” in *Proc. 2022 USENIX Annu. Tech. Conf.*, 2022, pp. 473–488.

[19] 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. ACM Symp. Cloud Comput.*, 2023, pp. 324–340.

[20] Microsoft, “Azure functions warmup trigger,” 2023. Accessed: Jun. 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-warmup

[21] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 559–572.

[22] L. Ao, G. Porter, and G. M. Voelker, “FaaSnap: FaaS made fast using snapshot-based VMs,” in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 730–746.

[23] Web Amazon Services, “Optimizing static initialization - AWS Lambda,” 2023. Accessed: Jun. 12, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/operatorguide/static-initialization.html

[24] E. Hunhoff, S. Irshad, V. Thurimella, A. Tariq, and E. Rozner, “Proactive serverless function resource management,” in *Proc. 2020 6th Int. Workshop Serverless Comput.*, 2021, pp. 61–66.

[25] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “SeBS: A serverless benchmark suite for function-as-a-service computing,” in *Proc. 22nd Int. Middleware Conf.*, 2021, pp. 64–78.

[26] R. B. Roy, T. Patel, and D. Tiwari, “IceBreaker: Warming serverless functions better with heterogeneity,” in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 753–767.

[27] J. R. Gunasekaran, P. Thinakaran, N. Chidambaram, M. T. Kandemir, and C. R. Das, “Fifer: Tackling underutilization in the serverless ERA,” in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 280–295.

[28] L. Pan, L. Wang, S. Chen, and F. Liu, “Retention-aware container caching for serverless edge computing,” in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1069–1078.

[29] R. B. Roy, T. Patel, and D. Tiwari, “DayDream: Executing dynamic scientific workflows on serverless platforms with hot starts,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2022, pp. 1–18.

[30] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip redundant paths to make serverless fast,” in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[31] Z. Lin, K.-F. Hsieh, Y. Sun, S. Shin, and H. Lu, “FlashCube: Fast provisioning of serverless functions with streamlined container runtimes,” in *Proc. 11th Workshop Program. Lang. Operating Syst.*, 2021, pp. 38–45.

[32] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, “Agile cold starts for scalable serverless,” in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput.*, 2019, Art. no. 21.

[33] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, “Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 153–167.

[34] X. Cai et al., “Incendio: Priority-based scheduling for alleviating cold start in serverless computing,” *IEEE Trans. Comput.*, vol. 73, no. 7, pp. 1780–1794, Jul. 2024.

[35] H. Yu et al., “RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing,” in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 335–350.

[36] T. Elgamal, “Costless: Optimizing cost of serverless computing through function fusion and placement,” in *Proc. 2018 IEEE/ACM Symp. Edge Comput.*, 2018, pp. 300–312.

[37] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, “{ORION } and the three rights: Sizing, bundling, and prewarming for serverless { DAGs},” in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 303–320.

[38] E. Oakes et al., “SOCK: Rapid task provisioning with serverless-optimized containers,” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 57–70.

[39] T. Schirmer, J. Scheuner, T. Pfandzelter, and D. Bermbach, “FUSIONIZE: Improving serverless application performance through feedback-driven function fusion,” in *2022 IEEE Int. Conf. Cloud Eng.*, 2022, pp. 85–95.

[40] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, “Memory deduplication for serverless computing with medes,” in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 714–729.

[41] K.-T. A. Wang, R. Ho, and P. Wu, “Replayable execution optimized for page sharing for a managed runtime environment,” in *Proc. 17th Eur. Conf. Comput. Syst.*, 2019, pp. 1–16.

[42] A. Agache et al., “Firecracker: Lightweight virtualization for serverless applications,” in *Proc. USENIX Symp. Networked Syst. Des. Implementation*, 2020, pp. 419–434.

[43] Google, “gVisor,” 2018. [Online]. Available: https://gvisor.dev/

[44] P. Silva, D. Fireman, and T. E. Pereira, “Prebaking functions to warm the serverless cold start,” in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 1–13.

[45] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, “Memory deduplication for serverless computing with medes,” in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 714–729.

[46] X. Wei et al., “No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing,” in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2023, pp. 497–517.

[47] Y. Zhang et al., “Faster and cheaper serverless computing on harvested resources,” in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 724–739.

[48] H. Yu, C. Fontenot, H. Wang, J. Li, X. Yuan, and S.-J. Park, “Libra: Harvesting idle resources safely and timely in serverless clusters,” in *Proc. 32nd Int. Symp. High- Perform. Parallel Distrib. Comput.*, 2023, pp. 181–194.

[49] H. Yu, H. Wang, J. Li, X. Yuan, and S.-J. Park, “Accelerating serverless computing by harvesting idle resources,” in *Proc. ACM Web Conf.*, 2022, pp. 1741–1751.

[50] Z. Zhou, Y. Zhang, and C. Delimitrou, “Aquatope: Qos-and-uncertainty-aware resource management for multi-stage serverless workflows,” in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 1–14.

[51] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, “{INFaaS}: Automated model-less inference serving,” in *Proc. 2021 USENIX Annu. Tech. Conf.*, 2021, pp. 397–411.

[52] J. Enes, R. R. Expósito, and J. Touriño, “Real-time resource scaling platform for Big Data workloads on serverless environments,” *Future Gener. Comput. Syst.*, vol. 105, pp. 361–379, 2020.

[53] Apache OpenWhisk, 2024. Accessed: Jul. 07, 2024. [Online]. Available: https://openwhisk.apache.org

[54] AWS Lambda, “Configure lambda function memory,” 2024. Accessed: Jul. 07, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html/

[55] T. Yu et al., “Characterizing serverless platforms with serverlessbench,” in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 30–44.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.

[56] R. M. Karp, *Reducibility Among Combinatorial Problems*. Berlin, Germany: Springer, 2010, pp. 219–241.

[57] B. Cheswick, “An evening with berferd in which a cracker is lured, endured, and studied,” in *Proc. Winter USENIX Conf.*, San Francisco, 1992, pp. 20–24.

[58] N. Corporation, “NVIDIA multi-process service,” *Softw. Available From NVIDIA*, 2024. Accessed: May 30, 2024. [Online]. Available: https://docs.nvidia.com/deploy/mps/index.html

[59] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 84–90.

[60] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[61] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

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

[63] C. Szegedy et al., “Going deeper with convolutions,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.

[64] Z. Li, “GitHub—Pagurus,” 2022. Accessed: Oct. 26, 2023. [Online]. Available: https://github.com/lzjzx1122/Pagurus

[65] Microsoft, “Azure functions premium plan,” 2024, Accessed: Jul. 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-functions/functions-premium-plan?tabs=portal

[66] “Pricing - microsoft azure function,” 2024. Accessed: Jul. 12, 2023. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/functions/

[67] J. Jarachanthan, L. Chen, F. Xu, and B. Li, “AMPS-Inf: Automatic model partitioning for serverless inference with cost efficiency,” in *Proc. 50th Int. Conf. Parallel Process.*, 2021, pp. 1–12.

[68] J. Cho, D. Zad Tootaghaj, L. Cao, and P. Sharma, “SLA-driven ML inference framework for clouds with heterogeneous accelerators,” in *Proc. Mach. Learn. Syst.*, vol. 4, pp. 20–32, 2022.

[69] J. Jiang et al., “Towards demystifying serverless machine learning training,” in *Proc. 2021 Int. Conf. Manage. Data*, 2021, pp. 857–871.

[70] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 45–59.

[71] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, “Optimizing inference serving on serverless platforms,” *Proc. VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.

[72] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, “Batch: Machine learning inference serving on serverless platforms with adaptive batching,” in *Proc. IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.

[73] Y. Yang et al., “Infless: A native serverless system for low-latency, high-throughput inference,” in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 768–781.

[74] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic ephemeral storage for serverless analytics,” in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 427–444.

[75] D. Mvondo et al., “OFC: An opportunistic caching system for faas platforms,” in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 228–244.

[76] A. Wanget al., “{InfiniCache }: Exploiting ephemeral serverless functions to build a { Cost-Effective} memory cache,” in *Proc. 18th USENIX Conf. File Storage Technol.*, 2020, pp. 267–281.

[77] F. Romero et al., “Faa$: A transparent auto-scaling cache for serverless applications,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 122–137.

Yifan Sui received the BEng degree in communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2021. He is currently working toward the PhD degree in the School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China. His research interests include cloud computing and serverless computing.

Hanfei Yu received the BE degree in electronic engineering from Shanghai Jiao Tong University, Shanghai, China, in 2019, and the MS degree in computer science from the University of Washington, Tacoma, WA, USA, in 2021. He is currently working toward the PhD degree in computer engineering with the Stevens Institute of Technology, Hoboken, NJ, USA. His research interests include cloud computing, serverless computing, reinforcement learning, and AI systems.

Yitao Hu (Member, IEEE) received the BS degree from the Department of Electrical Engineering, Shanghai Jiao Tong University, China, in 2014, and the PhD degree from Networked Systems Lab (NSL), University of Southern California, US, in 2021. He is currently an assistant professor with the College of Intelligence and Computing, Tianjin University, China. His research focuses on developing inference systems capable of deploying LLM and DNN models in large-scale cloud clusters, aiming for peak performance, efficiency and scalability. He has published papers at the leading conferences/journals, including SoCC, Ubicomp, INFOCOM, IWQoS, ASPLOS, SIGCOMM and TPDS.

Jianxun Li (Senior Member, IEEE) received the Dr. Eng. degree in control theory and engineering with highest honors from Northwestern Polytechnical University, Xian, China, in 1996. He is a professor with the Department of Automation, Shanghai Jiao Tong University, Shanghai, China. From 1997 to 1999, he joined the Key Laboratory of Radar Signal Processing of Xidian University, Xian, China, as a postdoctoral fellow. He was a visiting professor with the Imperial College London, London, U.K., from 2006 to 2007. His main research interests include information fusion, infrared image processing and parameter estimation.

Hao Wang (Member, IEEE) received the BE degree in information security, from Shanghai Jiao Tong University, in 2012, the ME degree in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2015, and the PhD degree in the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2020. He is an assistant professor in the Department Electrical and Computer Engineering with Stevens Institute of Technology, Hoboken, NJ, USA. His research interests include distributed ML systems, AI security and forensics, privacy-preserving data analytics, serverless computing, and high-performance computing. He is a recipient of the NSF CRII Award.

Authorized licensed use limited to: Stevens Institute of Technology. Downloaded on January 30,2026 at 13:19:29 UTC from IEEE Xplore. Restrictions apply.