

InfSquad: SLO-Aware Serverless Machine Learning Inference With Wasm-Assisted Hybrid Functions

Weilong Wang^{ID}, Borui Li^{ID}, *Member, IEEE*, Hai Wang^{ID}, *Member, IEEE*, Wei Xi^{ID}, *Member, IEEE*, Shuai Wang^{ID}, *Senior Member, IEEE*, and Tian He^{ID}, *Fellow, IEEE*

Abstract—Though serverless computing offers transformative benefits for deploying machine learning (ML) services, it faces challenges in meeting strict real-time service-level objectives (SLOs) of ML inference while maintaining resource efficiency. Fortunately, a new binary instruction format, WebAssembly (or Wasm), offers a promising solution for serverless ML inferences thanks to its short startup time and efficient execution compared with traditional container-based solutions. Therefore, we introduce InfSquad, a serverless ML inference framework designed to balance SLO-aware execution with resource efficiency. The key design of InfSquad is a Wasm-assisted hybrid serverless function runtime to harness the complementary strengths of Wasm and traditional containerized function runtime. Based on the hybrid runtime, InfSquad advocates an SLO-aware runtime scheduling approach that delivers efficient serverless inference. InfSquad leverages a proactive runtime recycling mechanism to increase resource efficiency further. Experiments on real-world applications show that, compared with state-of-the-art serverless systems, InfSquad achieves 19.8%–85.1% SLO violation reduction with 32.5%–74.8% fewer resources.

Index Terms—Serverless computing, machine learning inference, service-level objective (SLO), WebAssembly.

I. INTRODUCTION

SERVERLESS computing [1], [2], [3], [4], known for its effortless deployment and pay-as-you-go pricing model, is widely adopted to deploy various online machine learning (ML) services [5], [6], including data analysis [7], [8], video processing [9], [10], etc. ML inference with the serverless paradigm involves deploying inference tasks within isolated serverless functions and invoking them when needed. As a part of online services, these serverless inference tasks are typically posed with stringent latency service-level objectives (SLOs). For instance, applications such as online question-answering robots and fraud detection typically require 90% of requests to be processed within 200 ms [11].

Received 30 May 2025; revised 27 February 2026; accepted 11 March 2026. Date of publication 16 March 2026; date of current version 12 June 2026. This work was supported by the National Natural Science Foundation of China under Grant 62302096, Grant U24B20152, and Grant 62272098. (Corresponding author: Shuai Wang.)

Weilong Wang, Borui Li, Shuai Wang, and Tian He are with the Key Laboratory of Computer Network and Information Integration, School of Computer Science and Engineering, Southeast University, Jiangsu Nanjing 210096, China (e-mail: shuaiwang@seu.edu.cn).

Hai Wang is with the Sixty-third Research Institute, National University of Defense Technology, Jiangsu, Nanjing 210007, China.

Wei Xi is with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, Shanxi 710049, China.

Digital Object Identifier 10.1109/TSC.2026.3674218

Existing works of serverless ML inference [11], [12], [13] utilize containers as the serverless runtime. It requires hundreds of milliseconds to load model weights and heavyweight ML environments, resulting in a significant cold start delay. To tackle this issue and meet the latency SLO, these works either pre-warm the function in advance or keep functions active for subsequent requests, which leads to resource wastage. Other serverless ML platforms [14], [15] leverage multiple serverless runtimes, such as virtual machines (VMs) with containers, to achieve SLO-aware ML inference. Specifically, the heavyweight VMs run continuously and handle periodic requests, while containers are invoked on demand to manage unpredictable bursty inference. In summary, these existing works satisfy SLO by over-provisioning resources.

Fortunately, the rise and increasing utilization of WebAssembly (or Wasm) offer promising opportunities to facilitate SLO-aware yet resource-efficient serverless ML inference. Proposed in 2017, Wasm [16], [17], [18], [19] is a lightweight and efficient binary instruction format that is executable in a variety of environments. Wasm achieves approximately 100 times faster cold starts than traditional containers [20], making it particularly suitable for serverless inferences that require strict adherence to SLOs [21], [22]. Knative [23], an open-source serverless framework based on Kubernetes, now supports Wasm as a new function runtime alongside containers. These progresses encourage us to rethink the design of a serverless framework for SLO-aware ML inference with *hybrid function runtimes* that harnesses the advantages of both Wasm and traditional containers.

Designing such a framework faces non-trivial challenges: (i) How to design the hybrid function runtime to leverage the benefits of both Wasm and containers? According to our preliminary experiments, ML models exhibit varying performance across different function runtimes (i.e., Wasm and container), concerning different batch sizes and execution backends (i.e., CPU or GPU). (ii) How to schedule the serverless ML inferences with the hybrid function runtime in an SLO-aware manner while minimizing the resource overhead?

To address the aforementioned challenges, we introduce **InfSquad**, a serverless framework tailored for SLO-aware ML inference. The key idea of InfSquad lies in extending traditional single serverless function runtime to the design of *Wasm-assisted hybrid function runtime* to form a cooperative “squad” for ML inference. ML inference is served by either Wasm or a container, depending on the affinity of different models to various

runtimes. Additionally, considering the fast cold start of Wasm, InfSquad leverages another lightweight Wasm instance to do preparatory works such as data pre-processing when initiating the container.

Based on the hybrid function runtime, InfSquad proposes an *SLO-aware runtime scheduling* mechanism to acquire the optimal serverless inference configuration, including the selection of serverless runtime, batch size, and execution backend. To minimize resource footprint, InfSquad introduces a *proactive runtime recycling* mechanism to determine whether a function should be reused for further requests. The contributions of this paper are summarized as follows:

- We introduce InfSquad, an SLO-aware serverless ML framework that leverages Wasm-assisted hybrid functions. The key insight of InfSquad stems from the observation that different ML models exhibit varying levels of affinity for serverless runtimes and execution backends.
- We formulate the serverless ML inference scheduling task as a sequential decision-making problem and introduce an SLO-aware runtime scheduling algorithm built on clustered deep reinforcement learning (c-DRL). It simultaneously accounts for dynamic SLO constraints, the batch size, the serverless runtime, and the selection of execution backends, leveraging their complementary strengths to enhance inference performance.
- We build InfSquad on top of Knative [23], an open-source serverless framework, and evaluate it using real-world workloads. Evaluation results show that compared with state-of-the-art serverless frameworks, InfSquad reduces SLO violation rates by 19.8%–85.1% while using 32.5%–74.8% fewer resources.

The remainder of this paper is structured as follows: Section II presents the background and motivation behind InfSquad. Section III provides an overview of the framework, outlining its key components and design principles. Section IV delves into the methodology and technical design of InfSquad. Section V describes the experimental setup and provides a comprehensive analysis of InfSquad compared to existing methods. Section VII discusses the related works and Section VIII concludes the paper.

II. BACKGROUND AND MOTIVATION

In this section, we first discuss the specific characteristics of Wasm that are significant in our design, and provide illustrative examples of how Wasm performs in serverless ML inference, then present the target use cases of InfSquad.

A. Preliminaries of Wasm

Wasm is a low-level bytecode format that supports compilation from multiple high-level languages such as C, C++, C#, and Rust. It is now widely used as an isolation technology in serverless computing akin to containers.

Serving machine learning tasks with Wasm: In 2020, the Wasm working group introduces `wasi-nn` API to facilitate Wasm to execute ML tasks. `wasi-nn` provides a system interface definition for how Wasm interacts with existing ML

frameworks such as PyTorch and ONNX. This proposal includes APIs such as `wasi_nn::load()` for model loading, `wasi_nn::set_input()` to specify tensor input, and `wasi_nn::compute()` to invoke the inference process. With these `wasi-nn` APIs as a bridge, Wasm instances are able to execute ML models through existing ML inference libraries such as `libtorch` for PyTorch.

The import and export mechanism of Wasm: By design, Wasm offers strong guarantees of memory safety by limiting memory access to a single linear byte array, which is referenced using offsets from zero. This design enables efficient bound check during both compilation and execution. To facilitate controlled sharing of data and functions between Wasm modules and their external environments, the Wasm standard introduces import and export mechanisms. The import mechanism allows a Wasm module to specify its dependencies on external functionalities or memory regions from the host environment, while the export mechanism allows the host environment to access certain linear memory addresses inside the Wasm module. This bidirectional communication plays a vital role in facilitating dynamic linking, runtime adaptability, and effective collaboration between Wasm modules and the environments in which they are hosted.

B. Motivating Experiment

As previously stated in Section I, Wasm offers exceptionally low cold-start latency compared with containers, which motivates us to use Wasm to serve ML inference tasks under strict SLO constraints. We now evaluate how Wasm performs during ML inference compared with containers. Specifically, does Wasm consistently outperform other function runtimes in ML inference scenarios? To explore this, we utilize two prevalent ML models, ResNet50 and MobileNet, and evaluate their performance across various batch sizes, execution backends (i.e., CPU and GPU), and different types of serverless runtimes. Our findings are shown in Fig. 1.

Observation #1: Wasm demonstrates reduced cold start latency compared to containers, particularly in ML applications.

Fig. 1 presents the stacked execution time and memory consumption of Wasm and containers when running ML inference workloads. From the figure, we observe that the container's cold start latency is composed of two main components: (1) initialization of the function runtime and (2) configuration of the ML environment along with loading the model weights. In contrast, the cold start overhead of Wasm on both CPU and GPU is negligible. This is because Wasm is a compact, low-level bytecode representation, making its loading from storage significantly faster than that of a container. The heavyweight ML libraries with Wasm are also compiled into low-level binaries, which further improves the start latency.

Observation #2: There is no silver bullet when selecting runtimes and execution backends for different models.

As shown in Fig. 1, when the batch size is small, both models tend to use Wasm on CPU due to its fast cold start. As the batch size grows, leveraging the parallelism of the GPU becomes a more efficient choice. The reason behind the high overhead

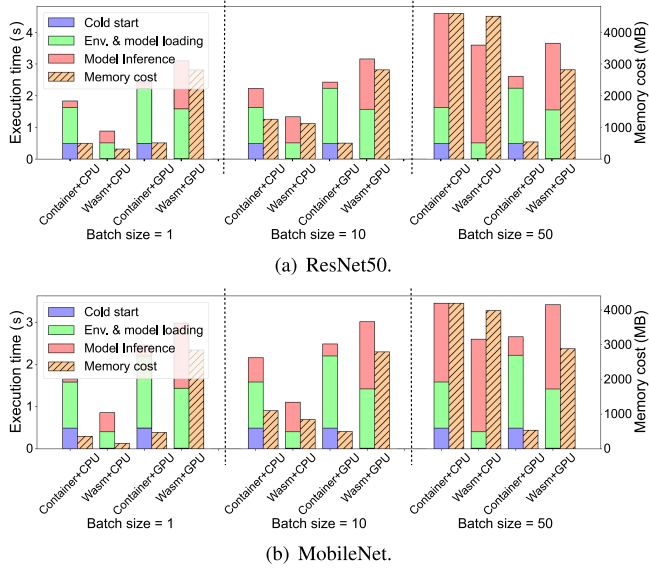


Fig. 1. Stacked execution time and memory consumption of Wasm and container for different ML models.

for Wasm on GPU under small batches is that Wasm supports GPU through the `wasi-nn` [24] interface, which leads to a substantial amount of redundant operations during computational execution. Regarding the selection of runtimes, relying exclusively on either Wasm or containers as the only serverless function runtime is not ideal. For example, with a batch size of 50, the optimal selection for ResNet50 inference is to use a container on a GPU. In contrast, for the light-weight MobileNet model, the most suitable configuration is Wasm on a CPU. These observations underscore the need for a *hybrid serverless runtime* that is adapted to the specific characteristics of each ML model to achieve optimal inference performance.

C. Target Real-World Use Cases

Nowadays, an increasing number of ML applications are deployed using the serverless paradigm, typically with container-based backends. However, the cold start of containerized runtimes often makes it challenging to maintain latency service-level objectives (SLOs). InfSquad is designed to address this issue by integrating the fast startup characteristics of Wasm into a hybrid serverless runtime, thereby minimizing SLO violations during ML inference.

This design aligns closely with real-world industrial use cases. A representative example is fraud detection in financial platforms, where serverless ML pipelines are commonly adopted to handle dynamic, event-driven transaction streams [25], [26]. In these systems, each incoming transaction is analyzed within a strict latency bound, typically on the order of milliseconds, since delayed decisions allow fraudulent transactions to be approved, making subsequent recovery nearly impossible [27]. With InfSquad, reducing the SLO violation means earlier fraud blocking. In practice, this improvement translates directly into higher fraud detection rates, fewer false approvals, and enhanced user trust in real-time financial services.

III. SYSTEM OVERVIEW

In this section, we first discuss the key design principles of and introduce the system overview of InfSquad.

Design principles: The design of InfSquad is guided by two key principles.

- *SLO-aware:* As an online ML inference serving framework, the violation of SLOs leads to customer dissatisfaction or financial loss [28]. Hence, the foremost principle of InfSquad is to meet the user-defined latency SLOs posed for ML services.
- *Resource-efficient:* Satisfying SLOs cannot come at the expense of over-provisioning resources. Therefore, InfSquad should provide resource-efficient serverless inference solutions by design.

The system overview of InfSquad is shown in Fig. 2. During the offline profiling stage, a *serverless function analyzer* examines the serverless program and generates the hybrid inference runtime for online serving, detailed in Section IV-A. The *model-wise affinity profiler* is responsible for measuring the runtime affinity of each deployed inference task, which provides important information for the scheduling algorithm. Specifically, the profiler models the execution time and resource usage of different ML models under different batch sizes, serverless runtimes, and execution backends.

During online inference serving, InfSquad focuses on achieving SLO-aware serverless ML inference with minimum resource footprint. As we mentioned in the motivating experiment, different selections of different function runtimes, batch sizes, and execution backends (i.e., CPU or GPU) are required under different system states, leading to varying impacts on SLO satisfaction. Towards this, we propose an *SLO-aware hybrid runtime scheduler* to determine the appropriate configuration for the ingress requests with full consideration of their SLOs (Section IV-B). After execution, a *proactive runtime recycling* module determines whether the executed function instance is recycled for consecutive inference to further reduce the resource footprint (Section IV-C).

IV. DESIGN OF INFSQLAD

In this section, we describe the three key pillars of InfSquad, i.e., the Wasm-assisted hybrid function runtime, SLO-aware hybrid runtime scheduling, and the management of cold starts with a proactive runtime recycling mechanism.

A. Wasm-Assisted Hybrid Function Runtime

As mentioned in motivation, the primary idea of InfSquad is the introduction of a hybrid serverless runtime based on Wasm and traditional containers for SLO-aware ML inference. Developing such a hybrid serverless runtime presents a crucial challenge, namely, how to fully utilize the strengths of each serverless runtime for further SLO-aware scheduling of each inference request. To address this problem, InfSquad proposes a hybrid serverless function runtime design. Specifically, the hybrid runtime not only harnesses Wasm and containers to handle ML inference requests but also leverages the rapid cold-start

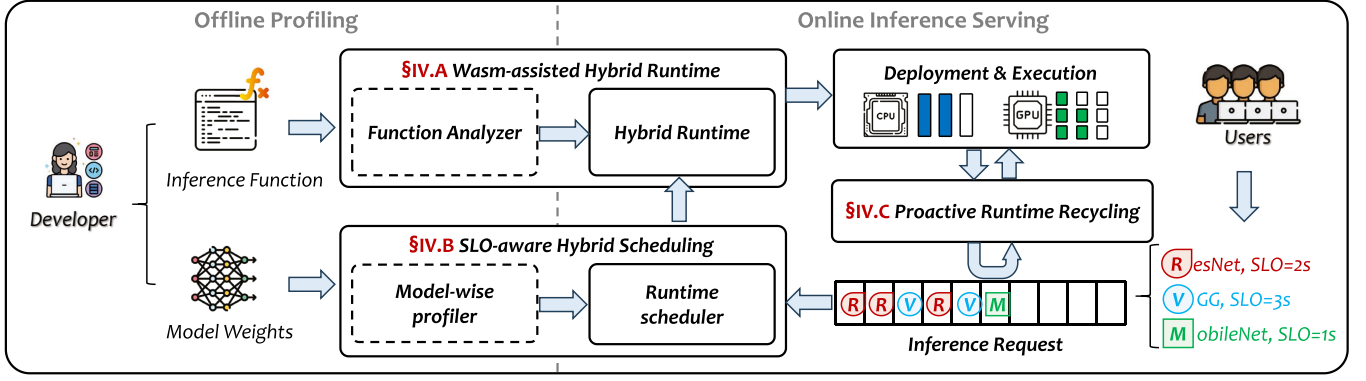


Fig. 2. System overview of InfSquad.

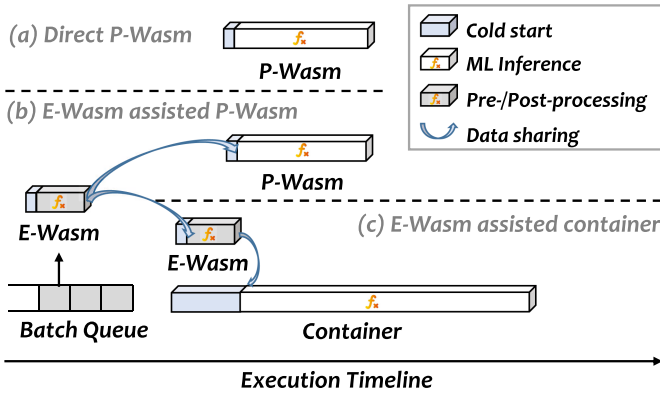


Fig. 3. Illustration of Wasm-assisted hybrid function runtime.

capability of Wasm to perform preparatory tasks during the initialization of containers. Moreover, InfSquad also introduces an efficient data sharing mechanism for zero-copy data transmission between Wasm and containers.

1) *Decoupled Design of Wasm-Assisted Serverless Runtime:* The hybrid serverless runtime of InfSquad includes three kinds of instances: E-WASM, P-WASM, and traditional containers. P-WASM and containers are designed to serve the inference independently. Moreover, besides the model inference, ML applications usually include pre-processing and post-processing tasks [29], [30]. E-WASM is designed as a lightweight Wasm-based executor to handle these tasks.

Fig. 3 illustrates the typical use cases of each instance. Existing works [11], [13] demonstrate the effectiveness of request batching to achieve efficient ML inference. InfSquad also considers batching as one of the vital scheduling criteria. Packing multiple requests into a batch means some requests should wait in the set, which impacts the satisfaction of SLOs. Therefore, for urgent requests, InfSquad directly spawns a P-WASM instance for timely processing, as shown in Fig. 3(a). InfSquad further separates the Wasm-based instance into P-WASM and E-WASM. Fig. 3(b) and (c) illustrate how E-WASM assists P-WASM and containers. Taking pre-processing tasks, such as tensor transformation and data augmentation for computer vision models, as an example, E-WASM leverages the waiting time in the batch set to

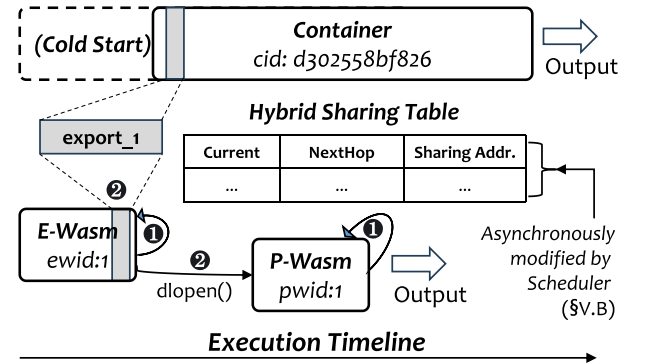


Fig. 4. Cross-runtime data sharing mechanism.

perform these tasks efficiently. Moreover, E-WASM serves as an assistant during the prolonged cold-start of container instances, performing preparatory tasks to minimize inference time.

The above decoupled design of the Wasm-assisted serverless runtime forms a “squad” for SLO-aware inference. This decoupled design takes advantage of the characteristics of each serverless runtime to enhance the tunable knobs of serverless ML inference. Moreover, this hybrid runtime design is easy to manage by commercial-off-the-shelf serverless frameworks (e.g., Knative) rather than requiring intrusive modifications to existing management frameworks.

2) *Efficient Data Sharing for Hybrid Runtime:* The hybrid function runtime and the decoupled design pose a new problem: how do we handle cross-instance data sharing efficiently? For example, the pre-processed data of E-WASM should be shared with its subsequent P-WASM or container instance for further inference. Traditional data-sharing methods are not suitable for the hybrid runtime. Leveraging the shared file is inefficient, and the inter-process memory mapping [18] breaks the memory isolation of Wasm instances. Thanks to the import/export and dynamic linking mechanism of Wasm [31], we design a zero-copy data sharing for the hybrid runtime.

The cross-runtime data sharing is shown in Fig. 4. We design two data-sharing approaches for different runtimes, respectively. Between E-WASM and P-WASM instances, we leverage the dynamic linking technique to load the Wasm instances together, in

```

1 def handler(event, context):
2     @InfSquad_EWasm
3     def convert_img(event):
4         r = requests.get(event['url'])
5         img = tf.image.decode_jpeg(r.content,
6                                 channels=3)
7         return img
8     img = convert_img(event)
9     converted_img = tf.image.convert_image_dtype(
10         img, tf.float32)[tf.newaxis, ...]
11     result = detector(converted_img)
12     return (result['detection_class_entities'])

```

Fig. 5. Example annotation to label E-WASM content.

which data-sharing is turned into data passing between variables. For the Wasm-container sharing surface, we exploit the memory export mechanism of Wasm to selectively share the linear memory that contains instance outputs to the next hop container. The above approaches minimize the data sharing overhead without breaking the isolation between instances.

Another core component of our approach is a hybrid sharing table, which is an importable object for Wasm instances. The rationale behind this design is that it decouples the workflow information (i.e., next hop) and scheduling decisions from the instance. This decoupling avoids instance rebuilding when the scheduling changes, and further improves the reusability of the Wasm instances, especially P-WASM, which minimizes the storage overhead of the hybrid runtime. With this sharing table, Wasm instances first look up the table for next hop information (① in Fig. 4) and prepare the sharing mechanism (via memory export or dynamic linking, ② in Fig. 4).

3) *Annotation-Based Wasm Instance Generation*: The next question is how to generate the appropriate E-WASM and P-WASM instances to act as the new tunable knobs for serverless inference scheduling. InfSquad adopts an annotation-based generation method to complete the initial decoupling of instances. In Fig. 5, we utilize a real-world serverless inference program for image object detection from AWS Lambda as an example. InfSquad provides code annotation decorators (@InfSquad_EWasm) for developers to label the pre-/post-processing stage of their serverless functions.

For the annotated serverless functions, InfSquad undergoes a policy-based validation to check whether the labels are legal for E-WASM and P-WASM abstraction. Specifically, InfSquad checks the following two policies during program analysis: 1) The execution time of each code snippet annotated as E-WASM is no longer than cold start latency. If it exceeds, the E-WASM instance is too heavy for agile scheduling. 2) The annotated part should be stateless to avoid unexpected behavior of the hybrid runtimes. Once the checks are passed, we profile the annotated code with different execution targets for the scheduler's usage.

B. SLO-Aware Hybrid Runtime Scheduling

Based on the aforementioned Wasm-assisted hybrid serverless runtime, InfSquad further schedules each request for appropriate configurations, i.e., batch size, execution runtime, and hardware backend.

1) *the dynamic-SLO Scheduling Problem*: Unlike existing works [11], [28], InfSquad considers that *each request has*

different SLOs rather than applying a common SLO for requests to the same ML service. The rationale behind this is that while InfSquad manages the latency for online serverless inference, it has a limited capacity to manage networking and redirection time among online services. InfSquad subtracts these additional times for each request from the predefined SLO, offering a more realistic and general assumption about latency management.

Hence, we consider a set of inference requests served by InfSquad, $\mathcal{Q} = \{q_1, q_2, \dots, q_n\}$, at a specific time. Each request q_i includes its request content req_i and its latency SLO constraint SLO_i , i.e., $q_i = \langle req_i, SLO_i \rangle$. To simplify the notation, we assume $\mathcal{Q}$ is sorted in ascending order of the SLO. As described in the motivation, the scheduling decisions for inference requests $\mathcal{Q}$ involve three aspects: batch size, selected function runtime, and execution backend. We define the feasible set for batch size as $\mathcal{B} = \{1, 2, \dots, |\mathcal{Q}|\}$. The feasible set for function runtime $\mathcal{R}$ includes using direct P-WASM, E-WASM assisted P-WASM, and E-WASM assisted container to serve inference requests, as shown in Fig. 3. The feasible set for execution backend is defined as $\mathcal{D} = \{\text{CPU}, \text{GPU}\}$. It is worth noting that both $\mathcal{R}$ and $\mathcal{D}$ are extensible when there are more new function runtimes or backend accelerators. Therefore, the set of scheduling decisions is defined as $\mathcal{A} = \mathcal{B} \times \mathcal{R} \times \mathcal{D}$.

When InfSquad takes action $a \in \mathcal{A}$, it forms a batch of requests $\mathcal{E} \subseteq \mathcal{Q}$ to serve with selected runtime and backend. We call $\mathcal{E}$ as the ready queue for requests. We use t_a and c_a to denote the inference time and resource cost of action a . During the scheduling, InfSquad first ensures that action a satisfies the SLO of each request in the batch. Furthermore, InfSquad also seeks to minimize the inference latency t_a and computational resource consumption c_a when taking action a . Consequently, we formulate the request scheduling problem for each time slice in InfSquad as:

$$\arg \min_{a \in \mathcal{A}} \quad \alpha t_a + (1 - \alpha) c_a, \quad (1a)$$

$$s.t. \quad t_a \leq \min_{q_i \in \mathcal{E}} SLO_i, \quad (1b)$$

$$c_a \leq C, \quad (1c)$$

where α is the configurable weight to trade off between the latency and resource consumption. Eq. (1b) constraints that action a satisfies the SLOs in the batch. C represents the current computational resources of the system.

2) *Problem Analysis*: The optimization problem in (1) describes the scheduling problem of each time step. As the ML inference requests arrive continuously, the scheduling problem in (1) turns out to be a sequential decision-making problem.

Although deep reinforcement learning (DRL) is widely studied for this sequential decision problem, the dynamic nature of the action space $\mathcal{A}$ poses challenges for deploying DRL in InfSquad. Specifically, the action space of batch size selection is $|\mathcal{Q}|$ at each time step, which is (i) typically large in online serverless serving systems and (ii) dynamic across different time steps because of the randomness of request ingress, preventing the usage of DRL. Therefore, we transform the problem of selecting batch sizes from $\mathcal{B}$ into determining whether each request in $\mathcal{Q}$

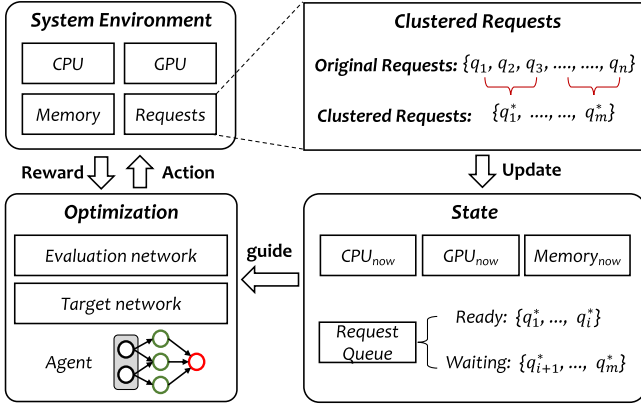


Fig. 6. The c-DRL framework of InfSquad.

should be included in the emitting request batch $\mathcal{E}$. Correspondingly, we use a new binary action space for batch size selection, defined as $\mathcal{B}^* = \{0, 1\}$, where 1 represents adding request q_i to the current batch and 0 means not. This transformation of action spaces facilitates the usage of DRL in InfSquad but leads to a new efficiency problem. The computational efficiency of DRL is linearly related to the number of requests in $\mathcal{Q}$. A larger $\mathcal{Q}$ leads to more DRL decision iterations and higher decision costs. Assuming that each request q has the same decision-making cost, denoted as c_q , the total decision-making cost is $C_{|\mathcal{Q}|} = c_q \times |\mathcal{Q}|$.

To facilitate a lightweight yet optimized scheduling in InfSquad, we incorporate the idea of clustering to construct clustered-DRL (c-DRL). The key idea of c-DRL is to cluster similar SLO requests into a group and apply the same action to them. Because they usually have the same execution priority and decision outcome. Such an idea reduces the training and decision time by the factor of cluster numbers, facilitating efficient scheduling of serverless functions online. Specifically, we have $\mathcal{Q}^* = \text{Cluster}(\mathcal{Q}, k, \mu)$, where k is the number of clusters. μ is a scaling factor that controls the maximum cluster size. For each request group $q^* \in \mathcal{Q}^*$, we have $|q^*| \leq \mu \cdot \lfloor |\mathcal{Q}|/k \rfloor$. This scaling factor trade-off between the efficiency and the effectiveness of the scheduling algorithm. Each q^* contains multiple q (where $q \in \mathcal{Q}$), and the decision-making results of these multiple q within q^* are identical. Therefore, q^* is treated as an independent q , with its decision-making cost also being c_q . The decision-making cost of $\mathcal{Q}^*$ is $C_{|\mathcal{Q}^*|} = c_q \times |\mathcal{Q}^*|$, and $C_{|\mathcal{Q}^*|} \leq C_{|\mathcal{Q}|}$. To summarize, the decision-making cost $C_{|\mathcal{Q}^*|}$ of c-DRL is always no greater than the decision-making cost $C_{|\mathcal{Q}|}$ of the original DRL.

3) *SLO-Aware Runtime Scheduling Via clustered-DRL*: With the clustering idea, we then introduce the key components of the c-DRL approach for serverless runtime scheduling, i.e., the *state space*, *action space*, *reward function* and *optimization strategy*. The c-DRL framework of InfSquad as shown in Fig. 6. The trained c-DRL model generates the optimal combination of batch size, execution backend, and selected serverless runtime.

State space S: The environment state of the scheduling problem encompasses the relevant performance attributes of the serverless inference, i.e., (i) the current available system resource C_t , (ii) the clustered requests and their corresponding

Algorithm 1: Hybrid Decision Strategy of InfSquad.

Input: Current environment state of scheduling $\mathcal{S}_t = \{C_t, \mathcal{Q}_t^*, \mathcal{Q}_t^S\}$, SLO, L_{Wasm} , L_{Con} , analyzer for calculating costs $Ay(\cdot)$.
Output: The decision strategy $\mathcal{S}'_t = \{C'_t, \mathcal{Q}_t^{*'}, \mathcal{Q}_t^{S'}\}$.

```

1 if the length of  $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S \leq L_{Wasm}$  then
2    $c^*, t^* = Ay(\mathcal{Q}_t^* \cup \mathcal{Q}_t^S, \text{P-WASM}, \text{CPU})$ ;
3   if  $c^*, t^*$  does not exceed  $C_t$  and SLO then
4     Use P-WASM with CPU for  $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S$ ;
5   else
6     Use the decision from c-DRL ;
7   end
8 else if the length of  $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S \leq L_{Con}$  then
9   if Keep-alive pool is not null then
10     $c^*, t^* = Ay(\mathcal{Q}_t^* \cup \mathcal{Q}_t^S, \text{Con}_w, \text{GPU})$ ;
11    if  $c^*, t^*$  does not exceed  $C_t$  and SLO then
12      Use Warm container with GPU for
13       $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S$ ;
14    else
15      Use the decision from c-DRL ;
16    end
17  else
18     $c^*, t^* = Ay(\mathcal{Q}_t^* \cup \mathcal{Q}_t^S, \text{P-WASM}, \text{CPU})$ ;
19    if  $c^*, t^*$  does not exceed  $C_t$  and SLO then
20      Use P-WASM with CPU for  $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S$ ;
21    else
22      Use the decision from c-DRL ;
23    end
24  end
25   $c^*, t^* = Ay(\mathcal{Q}_t^* \cup \mathcal{Q}_t^S, \text{Con}_c, \text{GPU})$ ;
26  if  $c^*, t^*$  does not exceed  $C_t$  and SLO then
27    Use Cold container with GPU for  $\mathcal{Q}_t^* \cup \mathcal{Q}_t^S$ ;
28  else
29    Use the decision from c-DRL ;
30  end
31 end

```

SLO requirements in the waiting queue $\mathcal{Q}_t^*$, and (iii) the selected requests in the ready queue to wait for batched execution $\mathcal{Q}_t^S$. Hence, the state set at time slot t is $\mathcal{S}_t = \{C_t, \mathcal{Q}_t^*, \mathcal{Q}_t^S\}$.

Action space A: As we discussed before, the action space of the c-DRL consists of the selection of batch, serverless runtime, and execution backend, i.e., $\mathcal{A} = \mathcal{B}^* \times \mathcal{R} \times \mathcal{D}$.

Reward R: The goal of the scheduling algorithm is to minimize the latency and resource usage of inference requests while satisfying the SLO. Therefore, we design the reward function of time slot t based on (1).

$$\mathbb{R}_t = \begin{cases} \frac{\alpha}{t_a} + \frac{1-\alpha}{c_a}, & \text{the SLOs of requests are satisfied.} \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Optimization strategy O: With the state space $\mathcal{S}$, action space $\mathcal{A}$, and reward function $\mathbb{R}$, we leverage the deep Q-learning (DQN) strategy to find the optimized value function of each

action for scheduling. During training, the DQN approach utilizes an evaluation network to choose actions and updates them through gradient descent. To enhance training stability, a target network is introduced as a fixed copy of the evaluation network, serving as a steady target for a specific number of steps. Thus, the Q-value in DQN is updated as:

$$Q_{t+1} = Q_t + \eta(\mathbb{R}_t + \gamma \max_{a'} Q_t(s', a'; \theta) - Q_t(s, a; \theta)), \quad (3)$$

where η represents the learning rate, determining the extent to which the error is learned, γ represents the discount factor, and θ denotes the adjustable parameter vector of the network.

Typically, Wasm performs better when the number of requests is small because it is closer to the native system. However, as the number of requests increases, Containers gradually outperform Wasm due to their stronger GPU support capabilities. Moreover, when Container is the instance, it has two startup methods: cold startup Con_c and warm startup Con_w , and warm startup has better performance than cold startup. Therefore, when computing resources are sufficient, people obtain relatively accurate configuration schemes through specific rule constraints. But system resources are not always sufficient.

For this reason, we incorporate DRL with the rule-driven heuristic scheme and comprehensively design a hybrid decision strategy of InfSquad, which is outlined in Algorithm 1. In scenarios with sufficient resources and clear workloads, we use fast and low-overhead heuristic algorithms to complete resource configuration (lines 3-27). Otherwise, we use c-DRL to complete resource configuration. Such scenarios are directly judged using L_{Wasm} , L_{Con} , and the results of the performance analyzer $Ay(\cdot)$. L_{Wasm} denotes a request number threshold when the ingress requests do not exceed L_{Wasm} , then the rule-driven heuristic directly serves the request with the P-WASM. L_{Con} denotes a request number threshold when the ingress requests exceed L_{Con} , then the rule-driven heuristic directly serves the request with the container in GPU Con_c . When the ingress requests fall between L_{Wasm} and L_{Con} , if there are keep-alive containers Con_w that meet the requirements, the container is used; otherwise, P-WASM is adopted. Here, $L_{Wasm} < L_{Con}$ satisfy the following conditions:

$$\begin{cases} Ay(L_{Wasm}, \text{P-WASM}, \text{CPU}) = Ay(L_{Wasm}, Con_w, \text{GPU}), \\ Ay(L_{Con}, \text{P-WASM}, \text{CPU}) = Ay(L_{Con}, Con_c, \text{GPU}). \end{cases} \quad (4)$$

Compared with rule-driven heuristic methods that rely solely on static rules, the hybrid decision strategy has stronger adaptability and decision accuracy. Compared with only c-DRL, the hybrid decision strategy improves decision efficiency while maintaining adaptability.

C. Managing Cold Starts With Proactive Runtime Recycling

Cold starts act as a vital optimization factor in serverless frameworks [32], [33], [34], [35]. Particularly for serverless ML inference, the time consumed for a cold start even exceeds the actual computation time for inference [36]. The state-of-the-art

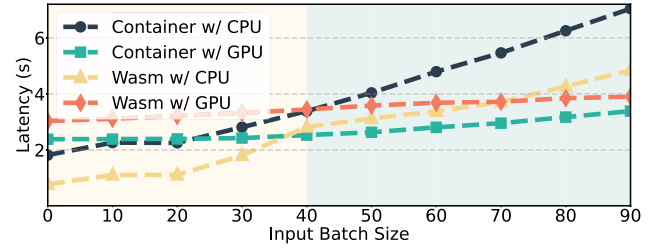


Fig. 7. Affinitive batch range of different runtimes and backends of an example image classification model.

Algorithm 2: Proactive Runtime Recycling.

Input: Current recycled function $f \in \mathcal{R} \times \mathcal{D}$, its affinitive batch range O_f , and current environment state of scheduling $\mathbb{S}_t = \{C_t, Q_t^*, Q_t^S\}$.

Output: Environment state after function recycling $\mathbb{S}'_t = \{C'_t, Q_t^{*'}, Q_t^{S'}\}$.

```

1 if  $Q_t^S = \emptyset$  then
2   | Keep-alive function pool  $\cup f \leftarrow f$  ;
3 else
4   | if  $(0, |Q_t^S|) \cap O_f \neq \emptyset$  then
5     |  $N \leftarrow \max\{(0, |Q_t^S|) \cap O_f\}$ ;
6     |  $f$  proactively steals top  $N$  requests from  $Q_t^S$  ;
7     | Update  $\mathbb{S}'_t = \{C'_t, Q_t^{*'}, Q_t^{S'}\}$  ;
8   | else
9     | Keep-alive function pool  $\cup f \leftarrow f$  ;
10  | end
11 end

```

approach [11] to reduce cold starts is to use the function invocation history to initiate the function predictively, i.e., *pre-warm*, and keep the function alive for subsequent requests after its execution, i.e., *keep-alive*.

The hybrid function runtime design of InfSquad requires us to rethink the cold-start mitigation problem in the serverless framework. Specifically, the predictive warm-up period is not necessary in InfSquad because the cold-start time of the Wasm-assisted hybrid function runtime is short. This design also reduces the resource wastage during the pre-warm period. Moreover, the hybrid runtime leads to a new problem that the function to be keep-alived is tailored with a specific runtime for a backend. Fig. 7 shows the illustration of the runtime affinity of an image classification ML model. We see that the optimal selection of hybrid runtime and backend varies with the input batch sizes. How to recycle the invoked function for inference efficiency is a non-trivial problem for InfSquad.

To address this problem, we propose a *proactive runtime recycling* mechanism to reuse the invoked function instance for improved SLO-aware and resource-efficient inference. The key idea of our approach is to maximize the reuse of invoked functions by allowing them to *proactively “steal” requests from the ready-to-execute request set*, thereby improving the SLO satisfaction rate with minimal additional resources. We specifically identify the affinitive batch range of a function instance, which,

when the batch size falls within this range, outperforms other runtime and backend choices. This range is determined during the offline profiling of each model.

Based on this definition, we have developed an algorithm for proactive function instance recycling, outlined in Algorithm 2. If the ready queue is not empty and the requests in it are processed optimally using the recycled function within the appropriate batch range, we directly utilize the recycled function to retrieve the requests from the queue and process them, as depicted in lines 3-8. Otherwise, the recycled function is sent to the keep-alive function pool to await invocation for a specified period, known as the keep-alive window. We implement the long-short term histogram strategy, a widely-used approach in existing serverless frameworks [11], to manage the keep-alive window.

V. PERFORMANCE EVALUATION

Our evaluation of InfSquad focuses on the following research questions: (1) How does InfSquad perform for SLO satisfaction and resource efficiency in comparison to state-of-the-art serverless inference frameworks [11], [12], [13]? (2) How does each building block of InfSquad contribute to the overall performance? (3) What are the overheads of InfSquad?

A. Methodology

Implementation details: We implement InfSquad on top of Knative [23], a widely-used open-source serverless framework. We leverage WasmEdge [37] as the Wasm runtime, Docker [38] as the container runtime for InfSquad. The model-wise affinity profiler is implemented using a random forest algorithm with linear correction. We further reduce the profiling overhead using Bayesian optimization during the training process of the profiler. The evaluation is conducted on server clusters, each equipped with i9-13900H and NVIDIA RTX 4060 GPUs.

The scheduling algorithm based on c-DRL is trained offline, where the learning rate η for training is $1e-3$, the discount factor γ is 0.99, and the buffer size is $1e6$. We set the training batch of internal learning as 32, the epoch of training is set to 500, and the number of timesteps is set to 2,000. We use the ϵ -greedy strategy to select the actions during training to balance exploration and exploitation. The configurable weight α that trades off between latency and resource consumption in Eq. 1 is set to 0.5. We use the K-means as the clustering algorithm with cluster number $k = 50$, and scaling factor $\mu = 2$. To prevent randomness and ensure rigor, we run the experiments 10 times.

Service workload: To further test the environmental adaptability of InfSquad, we simulate the dynamic invocation of inference services. We use the production trace from real world dataset with both long-term periodicity and short-term burst features [39]. Fig. 8 shows parts of the three typical types of production traces in [39]: *sporadic*, *bursty*, and *periodic*.

Baseline: We compare InfSquad with the following state-of-the-art serverless ML serving baselines: (1) *BATCH* [13]: Setting a fixed batch size cache space, executing ML inference as a single instance when the latency constraint is met or cache demand reaches the batch size. (2) *INFless* [11]: Adaptively selecting batches based on system computing resource information and

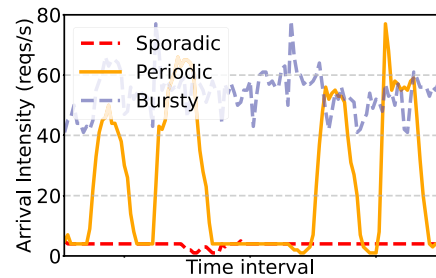


Fig. 8. Illustration of three real-world service workload traces used in evaluation.

the serverless environment system, and setting container warm start and keep-alive window parameters based on the historical demand data distribution histogram. (3) *Pagurus* [12]: Mainly using warm start methods to reduce the likelihood of cold starts, adaptively setting the containers that keep everything alive or retain important package loads, and the duration of keep-alive. To ensure experimental fairness, the baseline methods were provided with GPU computing support. Since the original Pagurus [12] paper does not explicitly address batch processing capabilities, we augmented its functionality by integrating techniques from BATCH [13] to enable batch processing support.

Benchmark: We utilize common ML tasks in daily transportation work as the benchmark, such as image classification, semantic segmentation, and object detection in daily traffic work. We utilize real transportation datasets [39] to implement road traffic request frequency and evaluate our work comprehensively with 8 models [40], [41]: (1) SqueezeNet, (2) ShuffleNet, (3) MobileNet, (4) ResNeXt, (5) ResNet50, (6) DeeplabV3, (7) VGG11, (8) TabNet.

Evaluation criteria: To evaluate the effectiveness of InfSquad, we utilize three evaluation criteria widely used in serverless frameworks [11]: (1) *SLO violation rate*: The percentage of requests whose SLO is not satisfied. (2) *Average latency*: The average latency of ML inferences, including queuing delay and inference delay. (3) *Average memory footprint/usage*: The average amount of computational memory required for each request.

Service SLO: To give a broader perspective of how InfSquad performs under different SLOs, we define three service SLO scenarios, specifically *strict*, *medium*, and *loose*. These scenarios correspond to setting the SLOs to $2\times$, $3\times$, and $4\times$ of the average execution time of the benchmark model.

B. Main Results

In this section, we outline the benefits of InfSquad relative to other baseline mechanisms. Unless otherwise stated, the experiments assume *periodic* data and a *strict* service SLO.

Comparison of SLO violation rate and average latency: As shown in Fig. 9, InfSquad consistently outperforms the baselines in terms of both SLO violation rate and average latency, regardless of the underlying model used for request processing. Specifically, compared to individual model deployments, InfSquad achieves a reduction of more than 80% in SLO violations rate and a more than 20% improvement in average

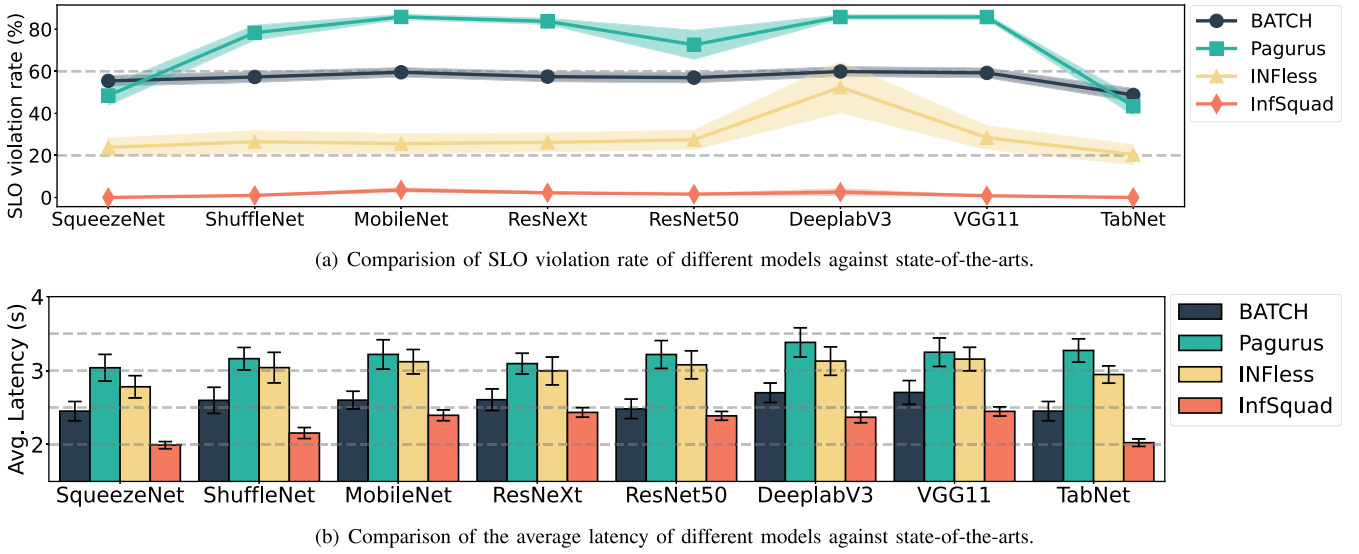


Fig. 9. The SLO violation rate and average latency of different models.

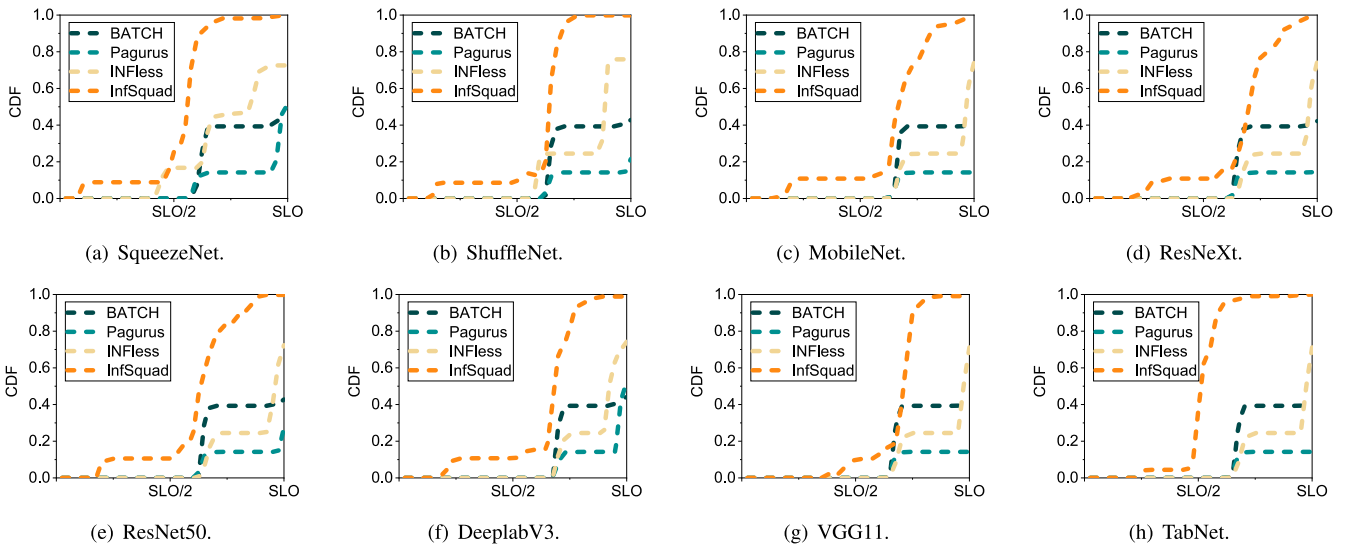


Fig. 10. CDF of function requests satisfying SLO constraints and the distribution of their execution latency for different models.

latency. Because InfSquad combines the advantages of both Wasm and container two runtimes, and utilizes Wasm to assist the container in doing part of the work, this effectively reduces the impact of cold start on the overall computation request. In addition, the hybrid decision strategy of InfSquad is well suited to the highly dynamic characteristics of incoming requests. It can determine the optimal batch size, execution backend, and serverless runtime for each request it processes. Although other methods utilize batch for batch processing, they consider relatively simple constraints, and it is difficult to obtain the optimal batch size. Furthermore, although existing approaches take into account how different execution backends affect the handling of function requests, they do not explicitly incorporate the role of the serverless runtime into their decision-making process. InfSquad proposes a direct container reuse strategy that

simplifies container management, further alleviates cold-start overhead, and enhances the overall execution efficiency of the system.

Comparison of CDF of SLO satisfaction rate: As illustrated in Fig. 10, InfSquad achieves a higher Cumulative Distribution Function (CDF) than other serverless ML inference approaches, and this superiority holds across most models. For most approaches, the latency of completed function requests is primarily distributed between SLO/2 and SLO, since their main objective is to maximize the number of served requests while satisfying the latency constraints. Our approach still attains the largest number of completed function requests. Unlike other approaches, InfSquad still completes some function invocations with latency below SLO/2, thereby effectively coping with sudden traffic spikes. This advantage comes from InfSquad leveraging Wasm to help

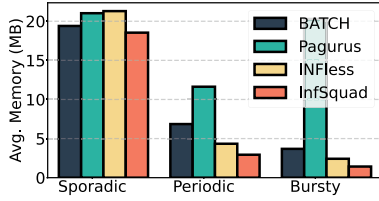


Fig. 11. Inference resource footprint under different service workloads.

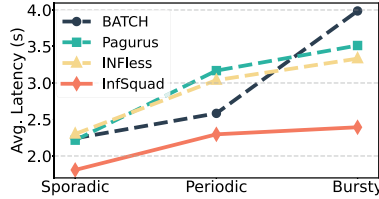


Fig. 12. Latency performance under different service workloads.

the container finish part of its workload. During the container's startup phase, Wasm exploits its extremely low startup latency to step in first and handle a portion of the container's tasks. This design reduces the container's overall execution latency and mitigates the performance penalty caused by cold starts. For each request set determined by the hybrid decision strategy, InfSquad selects the optimal batch size, execution backend, and serverless runtime for the current environment, thereby enhancing the system's parallel computing capability.

Comparison under different service workload: To further validate InfSquad's high adaptability to request environments, we conducted experiments in *sporadic*, *bursty*, and *periodic*. Fig. 11 and 12 show that InfSquad is superior to other solutions under all different kinds of workloads, especially when the request volume is high. This is because the hybrid runtime of InfSquad brings a variety of configuration alternatives to combine the advantages of both Wasm and container. It is able to select the best batch size and execution backend. In addition, the direct utilization of function instances and the self-applied selection of the optimal number of requests for function instances, efficiently reduces the runtime switching overhead. It improves the overall execution efficiency.

In *sporadic*, since there are relatively fewer requests, each method fulfills the requests within the SLO regulations, InfSquad has the lowest average latency. This is because, when the number of requests is small, InfSquad prioritizes the utilization of Wasm to complete the work. Wasm has lower resource consumption and start-up, and execution latency when processing small batches of data because it has a greater affinity to the system. When the number of requests is relatively low, the average memory of InfSquad is not optimal, but it is close to being optimal. This is because when using Wasm to assist other runtimes, there is an overhead associated with data sharing. But this overhead is not linear. As the number of requests increases (e.g., *sporadic*, *bursty*), the proportion of data sharing overhead in the computation of the requests gradually decreases until it becomes negligible.

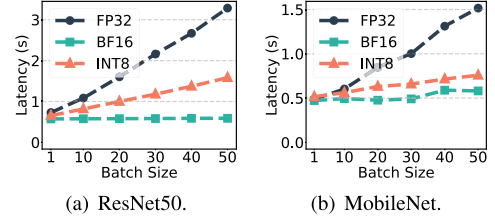


Fig. 13. Performance of models quantized at different levels.

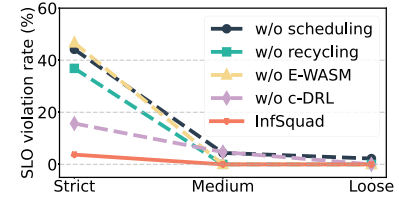


Fig. 14. The impact of different SLO constraints on the SLO violation rate.

Comparison of different quantization levels of ML models: Model quantification is one of the important means to improve the inference performance of models. Therefore, we analyze the performance of different quantized models in InfSquad. As shown in Fig. 13, we evaluate the performance of models with different quantization levels (FP32, BF16, INT8) when deployed on P-WASM. Model quantization effectively improves task execution efficiency and task completion rate, while reducing resource consumption. A particularly interesting observation is that a lower quantization level does not necessarily yield better performance. This is attributed to two main reasons. First, when Wasm executes ML inference, it relies on wasi-nn, whose internal execution logic is incompatible with the INT8 format. As a result, the INT8 format is converted back to BF16 or FP32, which further increases the conversion overhead. Second, for small-scale models, the additional quantization and dequantization operations required for INT8 computation may offset the performance gains brought by INT8 itself.

C. Ablation Study

To further validate the necessity of each building block in InfSquad, we conduct ablation experiments in this section. We evaluate with the following three variants of InfSquad: (1) *w/o scheduling*: InfSquad without scheduling method in Section IV-B, and utilize a fixed batch size; (2) *w/o recycling*: InfSquad without the proactive runtime recycling in Section IV-C; (3) *w/o E-WASM*: no E-WASM to assist other instances to complete the InfSquad. (4) *w/o c-DRL*: InfSquad only use our designed rule-driven heuristic scheme in algorithm 1, the c-DRL decisions involved are assumed to be empty. Fig. 14–17 shows that each part of InfSquad is essential, and InfSquad is superior to other ablation methods. We also observe that tighter SLO requirements lead to improved performance of InfSquad.

Comparison of SLO violation rate: As shown in Fig. 14, the SLO violation rate of the baseline is higher than that of InfSquad. This is because the adaptive scheduler method dynamically

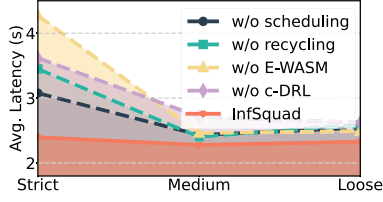


Fig. 15. The impact of different service SLO constraints on the average latency.

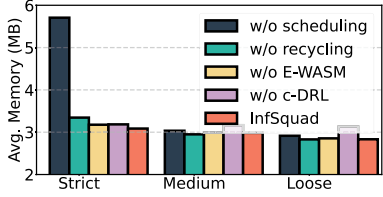


Fig. 16. The impact of different SLO constraints on the average memory.

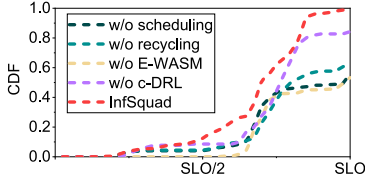


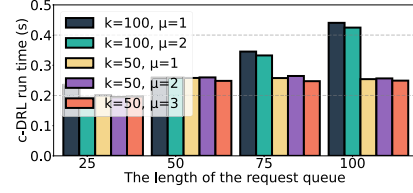
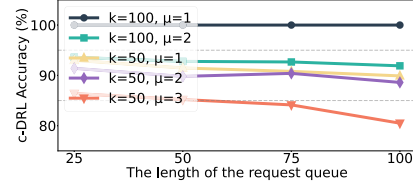
Fig. 17. Latency CDF distribution of different ablation variants of InfSquad.

selects the optimal batch size, execution backend, and serverless runtime based on real-time function requests. Moreover, both the E-WASM and recycling methods effectively reduce the impact of cold-start latency and improve task completion rates. The hybrid decision strategy effectively achieves optimal resource management. Consequently, InfSquad significantly enhances the system's computing capacity and reliably meets the SLO requirements of function requests.

Comparison of average latency: As shown in Fig. 15, we observe that the average latency of *w/o* E-WASM and *w/o* recycling is slightly worse than other methods. The rationale behind this is the *w/o* E-WASM method ineffectively utilizes the quick start-up and efficient computation features of Wasm. The *w/o* recycling method does not properly leverage the recycling mechanism. This results in numerous cold starts during runtime, which affects the overall execution time. The *w/o* c-DRL method cannot achieve fine-grained accurate decision-making when encountering dynamic fuzzy decision scenarios. This demonstrates the necessity of E-WASM recycling mechanism, and hybrid decision strategy.

Comparison of average memory: As shown in Fig. 16, the *w/o* scheduling method has the highest average memory. This is because it ineffectively implements adaptive batch size, execution backend, and function runtime. It fails to maximize the system's parallel computing capability. This highlights the necessity of adaptive batching.

Comparison of CDF of SLO satisfaction rate: As shown in Fig. 17, the overall CDF growth trends of the ablation variants

Fig. 18. The impact of k and μ on c-DRL run time.Fig. 19. The impact of k and μ on c-DRL decision accuracy.

are similar, but their task completion rates are lower than InfSquad. For the requests below SLO/2, the *w/o* E-WASM method barely completes any tasks, failing to effectively handle sudden bursts. This is because E-WASM effectively reduces the impact of cold-start latency and, in some cases, improves model inference efficiency.

D. Overhead of InfSquad

The overhead of InfSquad mainly comes from the decision of c-DRL. The overhead of the scheduling algorithm is linearly related to the size of the request set.

The c-DRL optimizes the overhead of reinforcement learning through clustering, as shown in Fig. 18 and 19. The overhead of c-DRL is related to the number of cluster points k and the service tolerance μ . As shown in Fig. 18, when the set space for decision-making is fixed, the larger the k value, the higher the overhead of c-DRL. This is because a larger k value requires a larger decision-making set space for DQN. As shown in Fig. 19, the larger the μ value, the lower the overhead of c-DRL, and it has an upper limit. The decision accuracy is decreasing. This is because a larger μ value allows a single decision to be utilized by multiple request decision points, affecting the granularity of request decisions and reducing decision accuracy. When k and μ are set to fixed values, the overhead of c-DRL continues to grow until it reaches a fixed value and no longer increases, due to the optimization and limitation of the set space by k and μ . In this study, we define $k = 50$, $\mu = 2$, the maximum c-DRL overhead is 0.25 s, with decision accuracy maintained above 90%. The size of the Wasm template file we designed in Section IV-A is 4.2 MB, which accounts for a small percentage of the resources required for the whole request.

VI. DISCUSSION

In this section, we discuss important open issues of InfSquad such as the applicability toward large language models. We concentrate on traditional ML models and workloads, and do not include Large Language Model (LLM) use cases in evaluation. This decision is based on two main considerations. First,

serverless LLMs constitute a substantial and distinct research area of their own. Second, our system is fundamentally based on Wasm. While Wasm has recently gained support for running LLMs, its capabilities for LLM deployment still trail behind state-of-the-art container-based solutions [37].

InfSquad offers a Serverless ML framework enhanced by Wasm. As Wasm technology continues to evolve, InfSquad is, in principle, capable of supporting the deployment of a wide range of ML models in the future. Moreover, for LLM-based services, the theoretical performance gains with InfSquad are even more pronounced. The reason is that LLMs typically have enormous parameter sizes and complex toolkit dependencies, which lead to substantial cold start overhead—precisely the aspect that InfSquad is designed to optimize. In other words, the larger the cold start overhead, the more significant the performance advantages of InfSquad are.

VII. RELATED WORK

Serverless machine learning frameworks: The judicious utilization of batching in server ML inference work not only reduces the computational costs of serverless, but also effectively improves resource utilization and lowers the overall service latency. BATCH [13] proposes a specialized buffering layer on serverless platforms to perform fixed-size batching of ML inference requests. By aggregating multiple requests into a single batch, it enhances resource utilization and processing efficiency, thereby lowering costs while preserving high performance. MBS [42], which determines batch sizes in a workload-adaptive manner, boosts the execution efficiency of heterogeneous ML inference tasks, but its focus is mainly on batching decisions in NLP workloads. INFless [11] leverages batching to improve performance. It integrates batching policies as first-class citizens within the serverless framework to shorten the response time of individual inference requests and to increase overall throughput. These works do not consider the impact of the cold start of a single function runtime instance on batching decisions. InfSquad preprocesses incoming requests in real-time with Wasm, effectively utilizing idle time and cold start time during low request rates, expanding the advantages of batch processing within limited resources.

Hybrid deployment of serverless and VMs: The mixed deployment of different isolation approaches is an effective solution for serverless workloads. For example, Mark [15] explores the cost-effectiveness of serverless inference in-depth and proposes an innovative hybrid deployment approach. It utilizes AWS EC2 for stable computational support while leveraging serverless systems for efficiently handling sudden traffic surges, achieving a cost-effective and responsive ML inference service. Sponge [14] combines serverless architecture with virtual machines for stream processing. It utilizes serverless for dynamic scalability and VMs for maintaining stability and high performance, optimizing efficiency and reliability in scenarios with short-term demand. Such hybrid deployments are suitable for short-term surge demands. RR-WCMD [43] integrates Wasm and container to reduce service construction cost for traditional microservice workloads. Since it does not account for service

compute cost, it is less suited for dynamic ML inference workloads with time-varying request arrivals. By jointly considering the computation characteristics and end-to-end lifecycle cost of ML inference services, InfSquad effectively handles time-varying surge demands across different periods.

Resource scheduling for serverless computing: Optimized scheduling of heterogeneous computing resources in the system effectively improves task execution efficiency, reducing overall service latency and costs. Tetris [44] reduces system memory resource consumption by sharing memory resources among multiple inference tasks, thus improving inference efficiency while ensuring inference quality. Faasm [18] and WasmFlow [45] leverage the features of Wasm to enhance task computational performance and security, optimizing start-up times and state management. They ensure efficient execution and scalability of demands. Pagurus [12] improves start-up speed and resource utilization by sharing container instances among different serverless functions, reducing overall system service latency. These works effectively address various challenges in resource management and task execution. The issues of start-up delay and system resource scheduling in highly dynamic demand scenarios still pose challenges. InfSquad utilizes a resource monitor to track system resources in real-time and intelligently selects the optimal batch size, execution backend, and function runtime using the c-DRL model.

VIII. CONCLUSION

In this paper, we propose InfSquad, an SLO-aware serverless ML framework that leverages Wasm-assisted hybrid functions. The key design of InfSquad is a Wasm-assisted hybrid serverless function runtime to harness the complementary strengths of Wasm and traditional containerized function runtime. We formulate the serverless ML inference scheduling task as a sequential decision-making problem and introduce an SLO-aware runtime scheduling algorithm. InfSquad simultaneously accounts for dynamic SLO constraints, the batch size, the serverless runtime, and the selection of execution backends, leveraging their complementary strengths to enhance inference performance. Experimental results with real-world benchmarks demonstrate that InfSquad achieves 19.8%–85.1% SLO violation rate reduction with 32.5%–74.8% fewer resources compared to state-of-the-art serverless systems.

REFERENCES

- [1] Z. Zhang, C. Jin, and X. Jin, "Jolteon: Unleashing the promise of serverless for serverless workflows," in *Proc. 21st USENIX Symp. Networked Syst. Des. Implementation*, 2024, pp. 167–183.
- [2] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," *ACM Comput. Surv.*, vol. 57, no. 3, pp. 1–36, 2024.
- [3] Y. Yang et al., "Multi-objective deep reinforcement learning for function offloading in serverless edge computing," *IEEE Trans. Serv. Comput.*, vol. 18, no. 1, pp. 288–301, Jan./Feb. 2025.
- [4] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A deep recurrent-reinforcement learning method for intelligent autoscaling of serverless functions," *IEEE Trans. Serv. Comput.*, vol. 17, no. 5, pp. 1899–1910, Sep./Oct. 2024.

- [5] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and M. B. Chhetri, "FaasHouse: Sustainable serverless edge computing through energy-aware resource scheduling," *IEEE Trans. Serv. Comput.*, vol. 17, no. 4, pp. 1533–1547, Jul.–Aug. 2024.
- [6] A. Mampage, S. Karunasekera, and R. Buyya, "Deep reinforcement learning for scheduling applications in serverless and serverful hybrid computing environments," *IEEE Trans. Serv. Comput.*, vol. 18, no. 2, pp. 718–728, Mar./Apr. 2025.
- [7] S. Werner and S. Tai, "A reference architecture for serverless Big Data processing," *Future Gener. Comput. Syst.*, vol. 155, pp. 179–192, Jun. 2024.
- [8] C. Jin et al., "DITTO: Efficient serverless analytics with elastic parallelism," in *Proc. ACM SIGCOMM 2023 Conf.*, 2023, pp. 406–419.
- [9] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 263–274.
- [10] S. Fouladi et al., "Encoding, fast and slow: {Low-Latency} video processing using thousands of tiny threads," in *Proc. 14th USENIX Symp. Networked Syst. Des. Implementation*, 2017, pp. 363–376.
- [11] 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.
- [12] Z. Li et al., "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 69–84.
- [13] A. Ali, R. Pincirol, F. Yan, and E. Smirni, "BATCH: Machine learning inference serving on serverless platforms with adaptive batching," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.
- [14] W. W. Song, T. Um, S. Elnikety, M. Jeon, and B.-G. Chun, "Sponge: Fast reactive scaling for stream processing with serverless frameworks," in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 301–314.
- [15] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for Cost-Effective, SLO-aware machine learning inference serving," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 1049–1062.
- [16] B. Li, H. Fan, Y. Gao, and W. Dong, "WaWoT: Towards flexible and efficient web of things services via webassembly on resource-constrained iot devices," *IEEE Trans. Comput.*, vol. 74, no. 3, pp. 1094–1108, Mar. 2024.
- [17] B. Li et al., "Bringing webassembly to resource-constrained IoT devices for seamless device-cloud integration," in *Proc. 20th Annu. Int. Conf. Mobile Syst., Appl. Serv.*, 2022, pp. 261–272.
- [18] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 419–433.
- [19] B. Li, W. Dong, and Y. Gao, "WiProg: A webassembly-based approach to integrated IoT programming," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–10.
- [20] X. Lyu, L. Cherkasova, R. Aitken, G. Parmer, and T. Wood, "Towards efficient processing of latency-sensitive serverless dags at the edge," in *Proc. 5th Int. Workshop Edge Syst., Anal. Netw.*, 2022, pp. 49–54.
- [21] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput.*, 2022, pp. 140–149.
- [22] wasmCloud Community, "wasmCloud community," 2023. [Online]. Available: <https://wasmcloud.com/community/>
- [23] K. Authors, "Knative: Running serverless workloads on kubernetes," 2024. [Online]. Available: <https://knative.dev/>
- [24] WASI, "Webassembly system interface (WASI)," 2025. [Online]. Available: <https://github.com/WebAssembly/WASI>
- [25] W. Amazon Services, "Real-time fraud detection using AWS serverless and machine learning services," 2023. [Online]. Available: <https://aws.amazon.com/blogs/machine-learning/real-time-fraud-detection-using-aws-serverless-and-machine-learning-services/>
- [26] S. Gupta, "Serverless fraud detection using amazon lambda, node.js, and hazelcast cloud," 2023. [Online]. Available: <https://hazelcast.com/blog/serverless-fraud-detection-using-amazon-lambda-node-js-and-hazelcast-cloud/>
- [27] Visa, "Reduce fraud with visa protect," 2025. [Online]. Available: <https://corporate.visa.com/en/products/visa-protect-a2a-payments.html>
- [28] Q. Pei, Y. Yuan, H. Yu, Q. Chen, and F. Liu, "ASYFUNC: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proc. 2023 ACM Symp. Cloud Comput.*, 2023, pp. 324–340.
- [29] M. Du, R. Tang, W. Fu, and X. Hu, "Towards debiasing DNN models from spurious feature influence," in *Proc. AAAI Conf. Artif. Intell.*, vol. 36, no. 9, 2022, pp. 9521–9528.
- [30] Y. Guo, J. Liu, G. Li, L. Mai, and H. Dong, "Fast and flexible human pose estimation with hyperpose," in *Proc. 29th ACM Int. Conf. Multimedia*, 2021, pp. 3763–3766.
- [31] Wasm, "WebAssembly component model," 2025. [Online]. Available: <https://github.com/WebAssembly/component-model>
- [32] K. Xiao, S. Yang, F. Li, L. Zhu, X. Chen, and X. Fu, "Making serverless not so cold in edge clouds: A cost-effective online approach," *IEEE Trans. Mobile Comput.*, vol. 23, no. 9, pp. 8789–8802, Sep. 2024.
- [33] 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.
- [34] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: State-of-the-art, challenges and opportunities," *IEEE Trans. Serv. Comput.*, vol. 16, no. 2, pp. 1522–1539, Mar./Apr. 2023.
- [35] Y. Huang et al., "Mobility-aware seamless virtual function migration in deviceless edge computing environments," *IEEE Trans. Mobile Comput.*, vol. 23, no. 7, pp. 7999–8014, Jul. 2024.
- [36] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 280–295.
- [37] WasmEdge Community, "WasmEdge runtime," 2025. [Online]. Available: <https://github.com/WasmEdge/WasmEdge>
- [38] D. Merkel et al., "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 239, no. 2, pp. 1–5, 2014.
- [39] T. Zhang, "Dataset for time series analysis - Transportation," 2023. [Online]. Available: <http://gitlab.fei8s.com/tianchengZhang/dastaset-for-timeseries/-/tree/main/Transportation>
- [40] PyTorch, "Pytorch hub," 2025. [Online]. Available: <https://pytorch.org/hub/>
- [41] S. Ö. Arik and T. Pfister, "TabNet: Attentive interpretable tabular learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 8, 2021, pp. 6679–6687.
- [42] A. Ali, R. Pincirol, F. Yan, and E. Smirni, "Optimizing inference serving on serverless platforms," *Proc. VLDB Endowment*, vol. 15, no. 10, 2022, pp. 2071–2084.
- [43] Z. Li, D. Zeng, and R. Chen, "Webassembly or container? Joint optimization of microservice consolidation and deployment towards cost efficient edge-end consortium," in *Proc. IEEE/ACM 32nd Int. Symp. Qual. Serv.*, 2024, pp. 1–10.
- [44] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 1–16.
- [45] X. Pang et al., "Featherlight stateful webassembly for serverless inference workflows," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 8, pp. 1651–1665, Aug. 2025.



Weilong Wang received the MS degree from the School of Computer and Control Engineering, Yantai University. He is currently working toward the PhD degree with the School of Computer Science and Engineering, Southeast University. His research interests include serverless computing, WebAssembly, and edge computing.



Borui Li (Member, IEEE) received the PhD degree in computer science from Zhejiang University, China, in 2022. He is currently an assistant professor with the School of Computer Science and Engineering, Southeast University. His research interests include AI-native IoT, WebAssembly, edge computing, and deviceless computing. He has authored or coauthored research papers in prestigious conferences and journals, including NSDI, MobiCom, ATC, INFOCOM, TC, and TOSN. He is also a member of the ACM.



Hai Wang (Member, IEEE) received the PhD degree from the School of Computer Science and Engineering, Southeast University, Nanjing, China, in 2024. He is currently a research assistant with the Sixty-third Research Institute, National University of Defense Technology. He is also with JD Logistics, Beijing, China, and Warwick University, Coventry, U.K. His research interests include spatio-temporal mining and decision-making in smart logistics, human-centered computing, and behavior-aware mobile computing.



Shuai Wang (Senior Member, IEEE) received the B.S. and M.S. degrees from the Huazhong University of Science and Technology, China, and the PhD degree from the Department of Computer Science and Engineering, University of Minnesota, in 2017. Since Nov. 2022, he has been the head with Computer Engineering Department. He is currently a professor with the School of Computer Science and Engineering, Southeast University. His research interests include the wireless networks and sensors, cyber-physical systems, data science.



Wei Xi (Member, IEEE) received the PhD degree in computer science from Xi'an Jiaotong University, in 2014. He is currently an associate professor with Xi'an Jiaotong University. His research interests mainly include wireless networks, mobile computing, and artificial intelligence. He is also a member ACM and CCF.



Tian He (Fellow, IEEE) is currently a professor with the School of Computer Science and Engineering, Southeast University. He was the recipient of the NSF CAREER Award in 2009, McKnight Land-Grant Chaired Professorship in 2011, George W. Taylor Distinguished Research Award in 2015, China NSF Outstanding Overseas Young Researcher I and II in 2012 and 2016, and Eight Best Paper awards in international conferences, including MobiCom, SenSys, and ICDCS. His research interests include wireless networks, networked sensing systems, cyber-physical systems, real-time embedded systems. He was the few general/program chair positions for multiple conferences and on many program committees. He also has been an editorial board member for six international journals, including *ACM Transactions on Sensor Networks*, *IEEE Transactions on Computers*, and *IEEE/ACM Transactions on Networking*. He is a fellow of ACM.