Laqista: Serverless Cloud-Fog-Dew Computing Platform for Deep Learning Applications

Seiki Makino
Graduate School of Media and Governance
Keio University
Fujisawa, Japan
kino-ma@sfc.wide.ad.jp

Tadashi Okoshi
Faculty of Environment and Information Studies
Keio University
Fujisawa, Japan
slash@sfc.keio.ac.jp

Jin Nakazawa
Faculty of Environment and Information Studies
Keio University
Fujisawa, Japan
jin@sfc.keio.ac.jp

2025 IEEE International Conference on Smart Computing (SMARTCOMP) | 979-8-3315-8646-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/SMARTCOMP65954.2025.00055

***Abstract*—In Smart Things and smart city applications, IoT devices generate large amounts of data and deep learning technologies are used to acquire useful information from it. Based on the kind of application and data, there are various non-functional requirements, such as low latency for information presentation by MR and privacy for video processing. To serve these requirements, a computing platform needs to make appropriate use of computing resources, namely Cloud, Fog, and Dew. However, there are some technical challenges in designing such a platform: i) transparently satisfying application QoS; ii) running the application across various hardware and OSes without modification; iii) sharing the application context taking into account the validity of values (temporal locality) and the data privacy (spatial locality). In this paper, we introduce `Laqista`, a novel Cloud-Fog-Dew computing platform. `Laqista` serves applications in a serverless manner via the Edgeless API, which schedules requests and abstracts the details of the platform. Applications are separated into Logics and Models, which are converted to lightweight, platform-agnostic formats such as WebAssembly and ONNX, respectively. Additionally, the Context Store synchronizes application context among the nodes, handling the privacy and validity of data. We developed a prototype implementation of `Laqista` in Rust and evaluated its performance. Experimental results show that the `Laqista` design has practical performance and is applicable to real-time applications such as video processing and MR.**

***Index Terms*—Smart Things, Cloud-Fog-Dew Computing, Serverless Computing, Deep Learning**

I. Introduction

Deep learning technology has enabled advanced information processing, leading to smart applications such as Smart Things, Smart Cities, and the Internet of Things. In deep learning applications, useful information is extracted from the data generated by end devices (sensors, smartphones, etc.). An example is a person-searching system. The simplified configuration of this application is shown in Figure 1. The application comprises four steps: i) a client process sends video frames obtained from surveillance footage to the server, ii) a server-side program pre-processes the input frames into vector data and invokes the model, iii) an object detection model processes the vector data inputs and returns a vector output, and iv) the server-side program receives the output and maps it to meaningful information (using labels such as ”dog” or ”human”). Many smart applications are constructed similarly, that is, data generation, data conversion, and model inference parts.

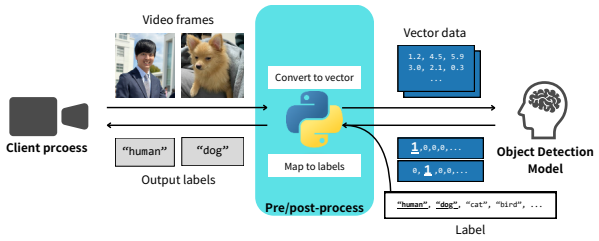

Fig. 1. Configuration of the person-searching system. This system consists of three parts, namely, the client process that generates video frames, the server-side pre-/post-process that performs conversion between meaningful data (images or labels) and vector data, and an object detection model.

According to the use cases and the type of data being processed, applications and their clients raise non-functional requirements for executing these processes. For instance, data collected from Automotive Sensing systems [1], [2] often includes citizens' private information. This occurs, for example, when citizens activities are recorded by dashcam footage. The recorded data must be processed within a trusted domain, which can be provided using Fog computing (*high-privacy scenario*) [2]–[4]. Another advantage of Fog computing is lower network latency. Thus, it is suitable for Mixed Reality (MR) as it ensures that users do not feel discomfort with the interface (*low-latency scenario*) [5], [6]. Meanwhile, high inference accuracy is required in trajectory prediction in autonomous driving systems (*high-accuracy scenario*) [7], [8]. Cloud Computing, which can provide abundant resources, is suitable for this scenario, instead of edge devices with limited computational resources (e.g., the Jetson Orin series).

2693-8340/25/$31.00 ©2025 IEEE
DOI 10.1109/SMARTCOMP65954.2025.00055

Anderson et al. found that these high-accuracy models fail to achieve real-time processing. Traffic sign recognition [9] needs to operate continuously during driving, and even amid network disconnections caused by vehicle movement (*network disconnection scenario*). Therefore, the inference must be conducted within the car independent from outside computers, which is called Dew Computing [10].

To serve such diverse scenarios on a single computing platform, three technical challenges should be overcome. First, the platform must consider the QoS requirements of the clients, in particular, accuracy, latency, and privacy. Simultaneously, it has to hide concrete QoS features, that is, *how* it manages resources to meet the requirements (*transparency* principle of QoS architecture [11]). Owing to the diversity and fluctuating characteristics of Fog and Dew devices, this transparency is non-trivial. Second, the application must be able to run across a heterogeneous environment. In the case of Dew (end) devices, most smartphones have ARM CPUs, whereas many cloud servers have Intel CPUs. In addition, iPhones provide Apple Silicon GPU through Metal API, whereas the major toolkit for NVIDIA GPU is CUDA. It is unfeasible for a developer to prepare programs for each of such diverse environments; thus, the platform should support this portability issue. Finally, applications should maintain their execution context over multiple invocations. For example, a video stream processing application may refer to the processing result of the last frame, so the result must be shared between nodes. Nevertheless, there are two *locality* issues for sharing context: i) The context data have *temporal locality*; for example, processing results from the last frame become invalid after the current processing; ii) The high-privacy scenario generates *spatial locality*, which means the context data should only be stored within a trusted domain. Thus, the platform must maintain context data considering these locality constraints.

In this paper, we introduce `Laqista`: Locality and QoS-aware Intelligence for Smart Things Application, a novel serverless computing platform for deep learning applications. `Laqista` adopts the Cloud-Fog-Dew architecture to support diverse scenarios. Neither the application (and its developer) nor the client need to care about physical resources, their state, or where the application runs. Thus, `Laqista` allows programming in a serverless manner, increasing the efficiency and scalability of the platform.

To address the first challenge, `Laqista` provides the scheduling interface called *Edgeless API* to clients. Beyond serverless, the applications are transparently distributed across Cloud-Fog-Dew, making the platform "edgeless." For this distribution, the Scheduler selects an appropriate node and model to meet QoS requirements, such as inference accuracy, execution time, and privacy.

For the second challenge, applications run on a portable and lightweight runtime, WebAssembly. Additionally, deep learning inferences are conducted on ONNX. Both of these runtimes are isolated from the outside environment, maintaining the security and privacy of the executing host, especially Dew devices.

The Context Store in `Laqista` provides an API for context sharing across instances/nodes maintaining privacy. It also functions under network disconnection on Dew devices; context data is cached locally on the device and remains valid during the specified lifetime.

The contributions of this paper are summarized as follows:

- We identified three technical challenges and requirements to realize a Cloud-Fog-Dew computing platform for smart applications.
- A novel Cloud-Fog-Dew computing platform named `Laqista` is proposed. `Laqista` addresses real-world issues by introducing Edgeless API, Context API, and program execution methods via WebAssembly and ONNX.
- We designed interfaces for implementing a distributed computing system with WebAssembly, whose specifications do not support networking features. In addition, we showed that designs are feasible and effective by developing a prototype implementation of `Laqista`.

II. Related Works

A. Partial Offloading of Deep Learning Models

MAUI [12] is a mobile-cloud computing architecture that automatically offloads functions in smartphone applications written in .NET to cloud servers. As .NET bytecode is platform-independent, program portability is ensured. Implementing deep learning models as combinations of functions allows this offloading to benefit smart applications.

DNN offloading systems including Neurosurgeon [13] and DIAMOND [14] offload model layers to the cloud (or edge) to reduce latency, energy consumption, and throughput. In Neurosurgeon's paper, the authors compared the performance with the MAUI, showing up to 39 times speedup.

Although these proposals offer various features, they are not suitable for our goal owing to a lack of execution environment isolation. The isolation mechanism is essential because end-users private data must be protected on personal Dew devices (smartphones, tablets, PCs, etc.). IoT devices also need security; for instance, the Mirai botnet comprises IoT and embedded devices. Furthermore, there is a risk of unintentionally incorporating malicious code from dependency libraries. In fact, PyTorch was compromised at the end of 2022 [15]. Malicious code was injected via the package registry, making it difficult to discover for application developers and end-users.

B. Inference Serving Systems

Inference Serving Systems such as Clipper [16] and INFaaS [17] provide deep learning inference services on the Cloud. The systems optimize inference accuracy, latency, or cost, by introducing technologies including the intermediate layer of request translation or the model-less system. Additionally, AlpaServe [18] demonstrates efficient model parallelism in inference serving.

While these systems operate on the cloud, Edge Intelligence, which combines deep learning with Edge Computing, is gaining attention. According to Wang et al. [3], our work falls

under *Edge Computing for deep learning* as we aim to design a platform to run deep learning applications.

However, existing research on Edge Computing for deep learning does not address all technological challenges. DeepDecision [19], MAUI [12] and MASM [20] optimize energy consumption, latency, and other metrics. However, the systems focus on inference scheduling algorithms and do not address distribution methods or application call interfaces. Therefore, systems for context sharing and host environment diversity need separate designs. Meanwhile, Zhang et al. [21] proposed an open framework called OpenEI to address the challenges of EI, namely, limited computing resources, data sharing and collaboration, and mismatch between AI algorithms and edge platforms. While `libei` of OpenEI defines how the clients and applications interact, context sharing via `libei` is tied to a specific node's IP address, making it non-functional when the application's execution location is unknown beforehand. In addition to context sharing and application call interface, these works do not consider the isolation of applications.

C. Migration Techniques for Entire Applications

Gackstatter et al. designed a serverless system named WOW that operates at the edge [22]. In WOW, the use of WebAssembly addresses program portability and execution isolation, enabling flexible migration between nodes. The WebAssembly runtime is extremely lightweight compared to Docker containers, improving cold start times by up to 99.5%. However, WebAssembly is unsuitable for deep learning due to the lack of GPU support and a memory limit of 4GB.

Gurgel et al. [23] designed a platform for deep learning applications by extending an Osmotic Computing system, Sapparchi [24]. Osmotic Computing allows context-sharing through its Micro Data (MD) element, whereas OC uses containers or lightweight VMs for application isolation which leads to dependencies on specific OS or hardware. Additionally, the system cannot address trade-offs between inference accuracy and processing time. Thus, further extensions are needed to meet the diverse QoS requirements of Smart Things and Smart City clients.

III. Design of Laqista

In this section, we detail the design of the proposed system, `Laqista`. The primary goal of `Laqista` is to build a Cloud-Fog-Dew computing platform for deep learning applications such as Smart Things, while satisfying the non-functional requirements from the clients (QoS). To address technical challenges, we designed the Edgeless API and Context API, and the application programs are distributed in a portable and lightweight format. The Edgeless API migrates applications *transparently* within the Cloud-Fog-Dew layers, whereas the Context API maintains context through migrations over spatial and temporal locality. These designs allow clients, applications, and operators to utilize resources without knowledge of the details of the platform, making the platform serverless and "edgeless."

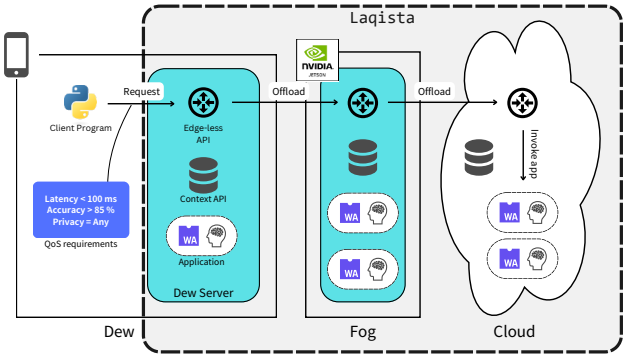

Fig. 2. Overview of the proposed method `Laqista`. `Laqista` consists of three layers: Cloud, Fog, and Dew. The Scheduler, Context Store, and Executor serve the Edgeless API, Context API, and application execution, respectively. All of these exist on all three layers.

A. Cloud-Fog-Dew Layering

As shown in Figure 2, `Laqista` consists of three layers following the Cloud-Fog-Dew architecture [10]. All three layers execute the Scheduler for request scheduling, the Executor for serving application, and the Context Store for managing application contexts. Thus, all layers have equivalent capabilities of application execution. The differences lie in the amounts of computing resources and locality, so requests are distributed considering these two points.

The Cloud layer consists of multiple computers with abundant computation, storage, and network capabilities. Although termed "Cloud" for convenience, public cloud services are not necessary; on-premise servers or lab machines would also function without operational differences. This layer has one Authoritative Node (AN) and others are Non-Authoritative Nodes (NANs). The AN runs the Scheduler and Context Store, whereas the NANs run the Executor. The AN is also responsible for managing applications deployed on the platform.

The Fog layer addresses the drawbacks of using Cloud servers, namely, network latency and privacy concerns. Fog nodes can be installed by any of the `Laqista` platform operators, application developers, or end-users. Examples of Fog nodes include Cloudlets [25], edge computers installed by application providers, or home desktop computers. In any way, Fog nodes exist along the network path (edge) from the client to the cloud (Internet), offering lower latency and higher privacy than the Cloud, while providing more powerful resources than the Dew (client). Fog nodes do not communicate with each other and independently run a Scheduler, Context Store, and Executor. Unlike the Cloud AN, which manages the entire platform, Fog nodes provide more localized functionalities to their connected clients, such as local cooperation for MR or autonomous driving systems. Additionally, the clients can ensure privacy by authenticating the Fog node with a public key. For example, in Smart City applications, end-users or the municipality can ensure that the node is in the trusted domain, in other words, that the municipality manages it.

The Dew layer includes devices that generate raw data,

such as end-users smartphones, desktop computers, VR head-mounted displays, and Smart City sensors. The processing capability strongly depends on the device's specifications. In particular, sensor devices may have minimal computing capabilities, whereas desktop computers often have powerful resources.

Dew devices run the client application program and the *Dew Server* simultaneously. The client program generates data to be processed, for example, by obtaining video frames from a USB camera. To conduct deep learning inferences on this data, the client process first asks for scheduling to the Dew Server's Edgeless API. Using the obtained target node from scheduling, it invokes the application by sending the image data. When invoking the application, the Dew Server routes the request to the appropriate node or conducts disconnected operations. In particular, it operates as a proxy between the client and upper layer. It registers to one parent node on the Fog or Cloud layer as a gateway for the offloading (Section III-B).

B. Edgeless API

In brief, the Edgeless API allocates resources to process upcoming requests. This allocation is transparently conducted through Cloud, Fog, and Dew, so the client is unaware of where the server is (serverless) or whether it is Edge Computing ("edgeless").

To call an application, the client must allocate resources by calling the Dew Server's Edgeless API with non-functional requirements, that is, acceptable processing time, required inference accuracy, and privacy. The Scheduler first checks if the node/device can meet the requirements on its own, based on the past execution time. If it cannot, it forwards the request to the upper layers, and the Fog node does the same (Figure 2 depicts this offloading flow). The Cloud AN eventually receives the request and it selects a NAN.

In addition to an execution location, the API is also responsible for selecting the appropriate model among multiple variants. The application can have multiple models for a single task (as discussed in Section III-C2), and the Scheduler determines the appropriate one based on the required accuracy.

For privacy requirements, clients can specify a particular trusted node using the node's identifier or a public key. This is useful for the application developer and end-user to process the request on the Fog node that they manage. To target multiple nodes while scheduling, the developer or end-user might install the same key pair on the nodes.

To conclude, the Edgeless API returns a node that meets privacy requirements and an RPC that meets accuracy requirements, while estimating the execution time of that combination to respect latency constraints. The platform operator can choose the concrete algorithm for resource estimation and node selection based on needs.

Through the Edgeless API, `Laqista` abstracts resources on the platform from the clients. The client specifies a **requirement**, and the rest is handled by the Scheduler, that is, resource allocation and availability management. For availability management, if a Dew device faces network disconnection, the Dew Scheduler will select the device itself as the target. Therefore, the clients can utilize the resources *transparently* through the non-functional requirements as an *interface*, addressing the first technical challenge. From this perspective, the API makes the platform serverless and "edgeless."

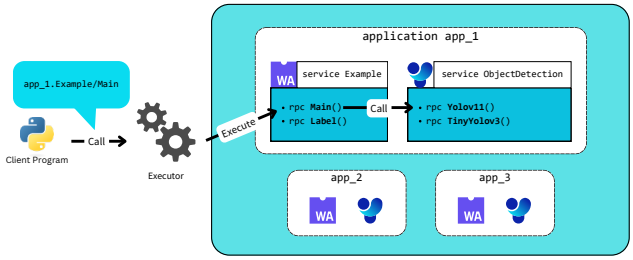

Fig. 3. Containment relationship of application, service, and RPC. An application contains Services of both (or one of) Logic and Model, and a Service contains RPCs. The client invokes the Edgeless API in advance and it calls the obtained RPC on the Executor of the target node. Because Logics and Models are separated, the Logic function must perform an RPC to conduct inference.

C. Applications

1) Program Format: WebAssembly [26] is an executable binary format independent of hardware, OS, or browser implementation. The isolation mechanism was designed for running on the web, making it valuable in various fields such as software plugin development. Moreover, the lightweight instruction set enables its near-native speed [27].

As these features meet our requirements, applications on `Laqista` are presented in WebAssembly. In addition, we adopted ONNX [28] for distributing and executing deep learning models, because WebAssembly does not support deep learning features. Therefore, `Laqista` applications function by invoking ONNX inference via RPC from WebAssembly programs.

Hereafter, we refer to the deep learning inference part represented by ONNX as the "**Model**" and the program part represented by WebAssembly as the "**Logic**."

2) Service Units in Applications: An application encloses the functions it provides in units called *Services*. Models and Logics each have at least one Service, and each Service contains at least one *RPC*. This containment relationship is shown in Figure 3.

For Models, each task is considered as a Service, with individual models treated as RPCs. For instance, an application using an object detection model might have RPCs like YOLOv11 and Tiny YOLO within the `ObjectDetection` Service. By preparing multiple models for a single task, applications can benefit from QoS scheduling via the Edgeless API.

For Logic, a WebAssembly program corresponds to a Service, with individual functions as RPCs. As described later, programs are divided into functions at the points of RPC calls.

Thus, a typical Logic might have an entry point RPC (e.g., `Main`) and a callback RPC after the model RPC call (e.g., `Labeling`).

3) Instantiation and Execution: Three events on the platform trigger instantiation of an application. The first event is when the application is deployed by the developer. The Cloud AN (Scheduler) provides an endpoint for deployment, which then initiates installation on a Cloud NAN. Another event is when a client requests the application on the Fog node or Dew device for the first time. Alternatively, a client can ask a node for instantiation in advance to avoid cold starts.

The Executor on each node manages the instantiated applications and serves incoming requests to them. It performs Ahead-of-Time compilation into executable native binary on the deployment, so that the application can start up instantly.

4) Communication Interface for the Outside World: Logics need to ask the host Executor for interaction with the outside world, that is, returning values to the client and invoking other RPCs. This is because the WebAssembly standard does not specify communication interfaces including networking. In addition, a function cannot directly return complex data types (structs) to the host. These restrictions are critical for `Laqista`, where applications are divided into Logics and Models and they intercommunicate via RPCs.

To address this restriction, data are transferred from the host to the function through the linear memory of WebAssembly. When a Logic is being invoked, the host (Executor) serializes the request payload into Protocol Buffers structs and writes it to the linear memory (which is zero-filled initially). After that, it calls the target function (Logic) passing the pointer to written data as an argument. As pointers can be expressed as a tuple of integer values (the address and data length), we can transfer it safely.

Similarly, the application must return `InvokeResult` using the same mechanism after it completes processing. `InvokeResult` is an enumeration type with three variants, `Finished`, `Error`, and `HostCall`. `Finished` and `Error` indicate successful completion and failure of the execution, respectively. `HostCall` means the Logic wants to call another Service on the platform. Additionally, the Context API is invoked through this interface as well.

As shown in Figure 4, the host invokes the specified Service when `HostCall` is returned. Before calling the Service, scheduling is conducted by the Edgeless API just like normal calls from clients. Thus, the target RPC might run on another node, enabling granular resource utilization.

The calling application must specify a *continuation* as a field. That is a callback function (i.e., an RPC) in the Logic that should be executed after a response has been obtained from the target Service. For example, applications may want to map a label to the inference result as the continuation post-process.

D. Context API

Applications can maintain their context over RPC invocations using the Context API provided by the Context Store. In

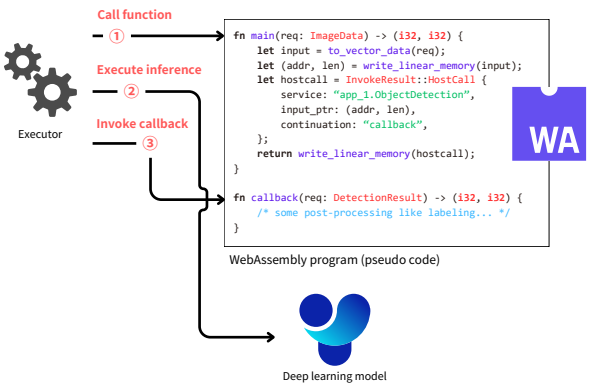

Fig. 4. Execution flow when `HostCall` is returned. The caller (a Logic) writes two data to the linear memory, the enum and the request body for the called Service (a Model). The host (an Executor) invokes the specified Service, and then calls back the caller's function.

the Cloud layer, the AN manages it, whereas in the Fog and Dew layers, each node reads and writes values on the local file system or memory. Alternatively, in the Cloud layer, external implementations like Redis may be used for scalability or other requirements.

The context can be accessed like a key-value store, with arbitrary keys chosen by the application, such as frame numbers or timestamps for video processing. Context data is divided by two scopes: *Node* and *App*. Node scope identifies where the data was written, and App scope indicates the application that initiated the write, allowing unique namespaces for different applications.

When new values are written, the Context Stores in each layer synchronize that data immediately. If the initial write destination was not the Cloud AN (the application was running on Fog or Dew), the node first transfers the data to the Cloud. Then, the AN propagates it from top to down, going through Fog and Dew. This propagation only applies to the nodes running the application that initiated the write operation.

Two non-functional requirements, *Privacy* and *Invalidate*, can be specified for the values being written. *Privacy* limits the node to propagate the value, in a similar manner to Edgeless API. In addition to the latter function, the data can be encrypted by the key for storing on untrusted nodes (typically, the Cloud AN). The other one *Invalidate* specifies the expiration time of the value. Data with this specification becomes invalid after the specified period. Based on this, a disconnected Dew device can determine whether the data is explicitly invalid to avoid entering an erroneous state.

IV. Evaluation

A. Experimental setup

We developed a prototype implementation of `Laqista`[1]. As a PoC, we implemented the Scheduler and Executor for Cloud, Fog, and Dew in `Laqista`. We selected Rust for development,

[1]https://github.com/kino-ma/Laqista

TABLE I
MACHINES USED DURING THE EXPERIMENT AND THEIR SPECIFICATIONS

Device Name	CPU	GPU	Memory	OS
Server	Intel i9-10940X	NVIDIA RTX A5000	128 GB	Ubuntu 20.04
Desktop	Intel Xeon E5-1620	Radeon R9 270X × 2	32 GB	NixOS 24.05
Laptop	Apple M3 Max	Apple M3 Max	128 GB	macOS 14.1
Client VM	AMD EPYC 7702P (16 vCPUs)	N/A	16 GB	NixOS 24.05

TABLE II
ELAPSED TIME TO START WEBASSEMBLY PROGRAMS

Measurement Target	Mean	Med.	Std. Dev
Compile	66.357 ms	66.610 ms	4.1531 ms
Instantiate	558.69 μs	556.63 μs	8.7580 μs
Compile and Instantiate	74.735 ms	75.509 ms	3.9061 ms

gRPC for communication protocol, Wasmer for WebAssembly, and WONNX for ONNX runtime[2].

For the experiment, we implemented a sample image processing application. The Logic first resized the input images to 224×224 and then invoked the image classification Model, SqueezeNet [29]. The Logic also extracted the most probable label from the output. The sample application was also implemented with Rust and compiled into WebAssembly.

We used three physical machines and a client VM for the experiment, as presented in Table I.

B. Latency

We measured the time required by `Laqista` to process a single request. All measurements were conducted on the *Laptop* machine. As a measurement tool, we used Criterion.rs[3].

1) Application Instantiation: In `Laqista`, applications are transformed into WebAssembly, possibly causing performance degradation. First, we determined the instantiation time of applications by measuring the AoT compilation, the instantiation of that pre-compiled binary, and the end-to-end startup time conducting those two. The measurement results are listed in Table II.

Based on the result, the application could start up in approximately 500 μs. The compilation to intermediate code is required only once, at deployment time, so it does not affect runtime performance. Additionally, the compilation time is shorter than the instantiation of Linux containers up to 100 times on V8 [30], making WebAssembly a good choice for isolation technology in distributed computing systems.

[2]Wasmer: https://wasmer.io/, wonnx: https://github.com/webonnx/wonnx
[3]https://bheisler.github.io/criterion.rs/book/index.html

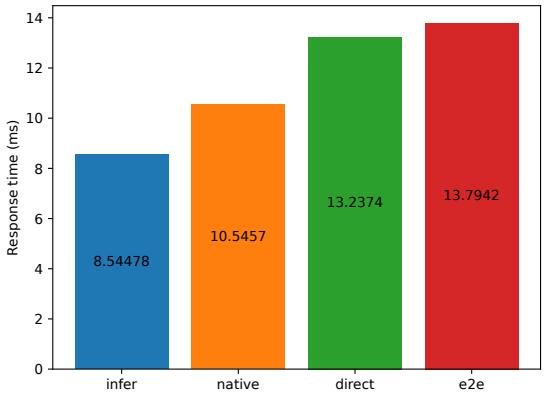

Fig. 5. Comparison of processing time among only inference using ONNX model (*infer*), invocation with native binary (*native*), with WebAssembly (*direct*), and via Edgeless API and with WebAssembly (*e2e*, this is the normal invocation on `Laqista`).

2) Runtime Latency Overhead: The main two possible causes of runtime latency overhead are execution speed degradation brought by WebAssembly runtime and invocation of Edgeless API. To identify those effects, we ran tests on four scenarios: i) application without Logic (only inference by the Model, for comparison), ii) application compiled directly to the native code, without invocation of Edgeless API, iii) application compiled to WebAssembly, without invocation of Edgeless API, iv) application compiled to WebAssembly, via invocation of Edgeless API. The results are shown in Figure 5.

As expected, a performance degradation between native code and WebAssembly code of approximately 25%, or 2–3 ms, can be observed. Even though WebAssembly is said to achieve "near-native speed," it still has a small runtime overhead. In addition, the Edgeless API has an extra overhead of approximately 500 μs, caused by reconnection to the Scheduler, scheduling, and reconnection to the target node. Furthermore, the native application bypasses the HostCall interface and the continuation system, reducing the reconnection and de-/serialization of payload and response.

However, these overheads caused by `Laqista` are acceptable in real-world applications. First, runtime performance should continue to improve over time as Jangda et al. [31] showed, comparing the benchmark results in 2017 and 2019. Second, the overheads will be added with each increase in processing within WebAssembly; thus, they will have less effect on smart applications, which are likely to have the dominant processing in deep learning models. Additionally, in practical scenarios in image processing, this overhead falls within a practical range. For example, the client requires a response time of approximately $1000 \div 24 \simeq 40$ ms to process 24 fps video streams. We obtained the result of 13.8 ms and the ratio was $13.8 \div 40 = 0.345 \simeq 35\%$; hence, real-time

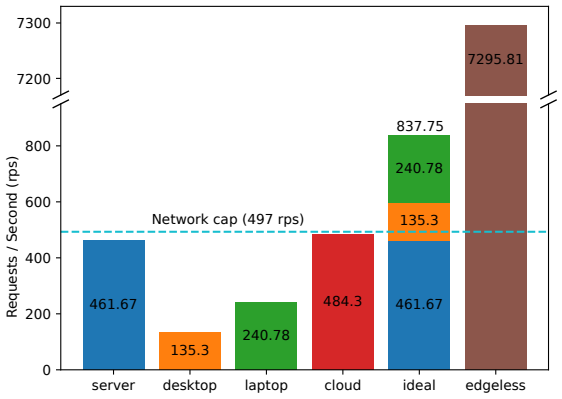

Fig. 6. Throughput measured on each machine. Three bars on the left side (*server*, *desktop*, *laptop*) show the throughput on a single node setup. The red bar with label *cloud* represents the result on the cloud cluster composed of the three machines. The stacked bar with label *ideal* shows the sum of three single-node results and means the theoretical throughput. The broken brown bar at the right shows the throughput for the Edgeless API called standalone without actual calls to the application. The dashed cyan line shows the virtual cap calculated from the network bandwidth.

performance was achieved. Furthermore, the overhead value of 3.2 ms compared to native binary services corresponds to only $3.2 \div 40 = 0.08 = 8\%$. From these observations, the overhead introduced by `Laqista` is not problematic in practical scenarios.

C. Throughput

Next, we measured the application delivery throughput on `Laqista` using Grafana k6[4]. Unless otherwise noted, the k6 clients were run at Client VM, and the targets were the Cloud cluster composed of all of or one of three machines: Server, Desktop, and Laptop. We ran four experiments: i) single node for each machine, ii) a Cloud cluster from the three machines, and iii) standalone Edgeless API on the Laptop machine. The results are shown in Figure 6.

First, we obtained a single-node performance of 240.78 requests per second (req/s) on the *Laptop* machine. Because we saw a latency of up to 14 ms on the same machine, it should serve at least $1000 \div 14 \simeq 71.4$ req/s in series. The actual value was 240.78 req/s, meaning it serves approximately $240.78 \div 71.4 \simeq 3.4$ requests in parallel. Therefore, the Executor is efficiently utilizing the node's resources.

When constructing a Cloud cluster, `Laqista` provides sufficient throughput for smart applications. Summing up the throughput of the three machines, they should theoretically serve a maximum of $461.67 + 135.3 + 240.78 = 837.75$ req/s. We observed 484.30 req/s, equivalent to serving 20 clients sending 24 fps video streams in parallel. The difference with the theoretical value was caused by a bottleneck in the network. According to the k6 report, the experiment consumed 888 Mbps bandwidth. The Client VM and each machine were interconnected via a 1 Gbps link with an actual measured bandwidth of 906 Mbps reported by the `iperf2` utility. Therefore, `Laqista` demonstrated a throughput exceeding the network bandwidth in an image processing application. Practical client devices (e.g., smartphones, sensors) are very likely to connect to 1 Gbps or lower links at present, so `Laqista` is unlikely to become a bottleneck.

Furthermore, the Edgeless API showed a throughput of 7295.81 req/s, which is far higher than the application delivery performance ($7295.81 \div 484.30 \simeq 14.9$ times). Thus, invoking the Edgeless API does not create a bottleneck for application execution.

[4]https://k6.io/

V. Discussions

A. Client Development

The only requirement for programs to act as `Laqista` clients is the capability to call the Edgeless API and send requests to target nodes based on scheduling results. Specifically, both are calls to gRPC services. Therefore, a program is ready to utilize the platform simply by importing a gRPC library. As gRPC is a widely-known open framework with libraries for various programming languages, `Laqista` is available for many existing programs with minimal modifications.

Another necessary preparation is to launch the Dew Server. This is a simple procedure, requiring only downloading and executing the executable binary (or source code) of the `Laqista` runtime. The client program may do this automatically, and thus, the end user will be unaware of using `Laqista`.

B. Performance Improvements

To further enhance `Laqista`'s processing performance, several additional mechanisms can be considered.

First, the Adaptive DNN model offloading techniques introduced in Section II-A can be implemented as **an application on** `Laqista` rather than a counterpart of `Laqista`. The main functionality of the platform is resource abstraction, but not performance improvement of applications. Applications can exceed its original performance by implementing offloading algorithms such as Neurosurgeon within it.

Additionally, a pre-emption mechanism would be effective. In our prototype, GPU task scheduling depends on the pre-installed schedulers. By switching a lower-priority processing with a higher-priority one, more requests could be satisfied with the QoS. The platform's performance can be further improved by replacing the schedulers with optimized ones for running multiple models with varying priorities.

VI. Conclusion

In this research, we proposed a novel Cloud-Fog-Dew computing platform `Laqista` for serving deep learning applications including Smart Things, Smart Cities, autonomous driving, and MR. The requests are served on heterogeneous devices/servers, scheduled by the Edgeless API for transparently satisfying the non-functional requirements. The clients and applications are unaware of where the request is served

(serverless) and whether this is performed at the Edge ("edgeless"). The Context Store maintains the applications' context data over such elastic execution, considering the temporal and spatial localities of the data. Furthermore, we designed an interface for data exchange and communication between WebAssembly programs and the host or RPC, which enables the execution of WebAssembly programs on distributed computing systems.

We developed a prototype implementation of `Laqista` and conducted quantitative performance evaluations. On an image processing application, `Laqista` brought a latency overhead of only 3 ms, and the end-to-end latency was approximately 13 ms, sufficiently efficient for real-time video stream processing. Application developers can further improve the performance by implementing DNN offloading techniques on the application layer. At the same time, `Laqista` served 484 req/s with three nodes, which is sufficient to serve 20 clients sending 24 fps video streams. This throughput hit the network capacity of the 1 Gbps link; thus, `Laqista` would not be a bottleneck in practice. Therefore, `Laqista` serves smart applications efficiently, enabling various applications with complex requirements such as real-time performance, privacy, and high-accuracy inference.

ACKNOWLEDGEMENT

This work was supported by JSPS KAKENHI Grant Number JP23H00476. These research results were partly obtained from the commissioned research project (Grant No. 23602) by the National Institute of Information and Communications Technology (NICT), Japan.

REFERENCES

[1] Y. Chen, J. Nakazawa, T. Yonezawa, T. Kawasaki, and H. Tokuda, "An empirical study on coverage-ensured automotive sensing using door-to-door garbage collecting trucks," in *Proceedings of the 2nd International Workshop on Smart*, 2016, p. 1–6.

[2] K. Mikami, Y. Chen, J. Nakazawa, Y. Iida, Y. Kishimoto, and Y. Oya, "Deepcounter: Using deep learning to count garbage bags," in *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2018, p. 1–10.

[3] X. Wang, Y. Han, V. C. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, p. 869–904, 2020.

[4] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, p. 637–646, 2016.

[5] Y. Itoh, T. Langlotz, J. Sutton, and A. Plopski, "Towards indistinguishable augmented reality: A survey on optical see-through head-mounted displays," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, p. 1–36, 2021.

[6] S. Shannigrahi, S. Mastorakis, and F. R. Ortega, "Next-generation networking and edge computing for mixed reality real-time interactive systems," in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 2020, p. 1–6.

[7] A. Rasouli, M. Rohani, and J. Luo, "Bifold and semantic reasoning for pedestrian behavior prediction," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, p. 15600–15610.

[8] N. Sharma, C. Dhiman, and S. Indu, "Pedestrian intention prediction for autonomous vehicles: A comprehensive survey," *Neurocomputing*, vol. 508, p. 120–152, 2022.

[9] D. Tabernik and D. Skočaj, "Deep learning for large-scale traffic-sign detection and recognition," *IEEE transactions on intelligent transportation systems*, vol. 21, no. 4, p. 1427–1440, 2019.

[10] K. Skala, D. Davidovic, E. Afgan, I. Sovic, and Z. Sojat, "Scalable distributed computing hierarchy: Cloud, fog and dew computing," *Open Journal of Cloud Computing (OJCC)*, vol. 2, no. 1, p. 16–24, 2015.

[11] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of qos architectures," *Multimedia systems*, vol. 6, p. 138–151, 1998.

[12] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, 2010, p. 49–62.

[13] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, p. 615–629, 2017.

[14] J. Ahn, Y. Lee, J. Ahn, and J. Ko, "Server load and network-aware adaptive deep learning inference offloading for edge platforms," *Internet of Things*, vol. 21, p. 100644, 2023.

[15] "Compromised pytorch-nightly dependency chain between december 25th and december 30th, 2022." https://pytorch.org/blog/compromised-nightly-dependency/, 2022.

[16] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A Low-Latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, p. 613–627.

[17] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "Infaas: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, p. 397–411.

[18] Z. Li, L. Zheng, Y. Zhong, V. Liu, Y. Sheng, X. Jin, Y. Huang, Z. Chen, H. Zhang, and J. E. Gonzalez, "Alpaserve: Statistical multiplexing with model parallelism for deep learning serving," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, p. 663–679.

[19] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM 2018-IEEE conference on computer communications*. IEEE, 2018, p. 1421–1429.

[20] W. Zhang, Z. Zhang, S. Zeadally, H.-C. Chao, and V. C. Leung, "Masm: A multiple-algorithm service model for energy-delay optimization in edge artificial intelligence," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, p. 4216–4224, 2019.

[21] X. Zhang, Y. Wang, S. Lu, L. Liu, and W. Shi, "Openei: An open framework for edge intelligence," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, p. 1840–1851.

[22] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, p. 140–149.

[23] L. Gurgel, A. Souza, N. Cacho, and F. Lopes, "Deep learning distribution model using osmotic computing," in *2023 IEEE International Smart Cities Conference (ISC2)*. IEEE, 2023, p. 1–7.

[24] A. Souza, N. Cacho, T. Batista, and R. Ranjan, "Sapparchi: an osmotic platform to execute scalable applications on smart city environments," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, 2022, p. 289–298.

[25] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, p. 14–23, 2009.

[26] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, p. 185–200.

[27] "Wasmer: The universal webassembly runtime," https://wasmer.io/.

[28] "Onnx — home," https://onnx.ai/, 2024.

[29] F. N. Iandola, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[30] "How workers wokrs - cloudflare workers docs," https://developers.cloudflare.com/workers/reference/how-workers-works/, Oct 10 2024.

[31] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of WebAssembly vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, p. 107–120.