A WEBASSEMBLY-ENABLED FEDERATED LEARNING DEPLOYMENT FRAMEWORK FOR FOG COMPUTING ENVIRONMENTS

DOI: 10.36724/2072-8735-2026-20-6-52-62

Manuscript received 21 March 2026;
Accepted 26 May 2026

Dang Van Thang,
The Bonch-Bruevich Saint-Petersburg State University of Telecommunications (SPbSUT), St. Petersburg, Russia,
dangvanthang_sut@mail.ru

Andrey E. Koucheryavy,
The Bonch-Bruevich Saint-Petersburg State University of Telecommunications (SPbSUT), St. Petersburg, Russia,
akouch@mail.ru

Keywords: *WebAssembly, federated learning, fog computing, 6G networks, heterogeneous devices*

Federated learning (FL) in fog computing (FC) environments for 6G networks offers significant benefits across a wide range of domains by leveraging data locality to preserve the privacy of sensitive information, while also reducing communication costs and relieving server load. However, deploying FL tasks on fog nodes with heterogeneous hardware configurations, distinct operating systems, and diverse instruction set architectures remains a major challenge, especially given 6G's stringent latency requirements and tight integration with real-time applications. WebAssembly (Wasm) is a portable bytecode format that is secure through sandboxing and capability-based security, delivers near-native performance, has a compact footprint, and runs across platforms without recompilation, providing a potential means to overcome these limitations in 6G-enabled fog computing. In this study, we present a Wasm-integrated federated learning deployment framework for fog computing in 6G networks. To evaluate practical feasibility and performance, we conducted experimental evaluations on a heterogeneous set of devices. The analysis indicates that the Wasm-based framework reduces memory and CPU utilization while maintaining high accuracy, aligning with the performance requirements of 6G.

Information about authors:

Dang Van Thang, *The Bonch-Bruevich Saint-Petersburg State University of Telecommunications (SPbSUT), St. Petersburg, Russia. ORCID: https://orcid.org/0009-0009-2219-3767*

Andrey Koucheryavy, *The Bonch-Bruevich Saint-Petersburg State University of Telecommunications (SPbSUT), St. Petersburg, Russia. ORCID: https://orcid.org/0000-0003-4479-2479*

Для цитирования:
Данг В.Т., Кучерявый А.Е. Структура для развёртывания федеративного обучения на основе WebAssembly в гетерогенных туманных вычислениях // T-Comm: Телекоммуникации и транспорт. 2026. Том 20. №6. С. 52-62.

For citation:
V.T. Dang, A.Y. Koucheryavy, "A WebAssembly-Enabled Federated Learning Deployment Framework for Fog Computing Environments," *T-Comm*, 2026, vol. 20, no. 6, pp. 52-62.

1. Introduction

Over the past decade, the traditional cloud computing model has been a principal driver of the artificial intelligence (AI) revolution. However, as AI applications increasingly penetrate domains requiring real-time interaction and the handling of sensitive data, particularly in 6G networks characterized by very high throughput and ultra-low latency, the inherent limitations of cloud-centric processing, including network latency and privacy risks, have become more apparent than ever. This has spurred a shift from centralized processing toward distributed paradigms such as edge computing and fog computing [1]. Functioning as an intermediate layer between edge devices and the cloud in 6G networks, fog computing extends computation and storage closer to data sources. By processing data locally, this architecture substantially reduces latency, conserves bandwidth, and supports applications such as telepresence, holography-type communications, autonomous vehicles, industrial IoT, and remote medical monitoring, thereby enabling immediate responses to real-world events in 6G environments [2]. Consequently, a tightly coupled Cloud–Fog–Client (IoT) infrastructure architecture has emerged [3], as illustrated in Fig. 1. Recent surveys confirm that fog computing is becoming a key enabler for latency-sensitive services such as telepresence and immersive communications in next-generation networks [4].

In parallel, advanced machine learning approaches have been intensively studied, developed, and deployed in practice. Among these, FL is particularly well suited to distributed computing models such as fog computing in 6G networks. FL enables multiple devices to collaboratively train a shared AI model without transmitting raw data to a central server. Instead, each device (fog node or client) performs local training and communicates only model parameter updates for aggregation. This approach addresses data-privacy concerns, reduces server-side computational load and communication costs, and aligns naturally with fog environments in which data are generated and stored across millions of distributed devices [5].

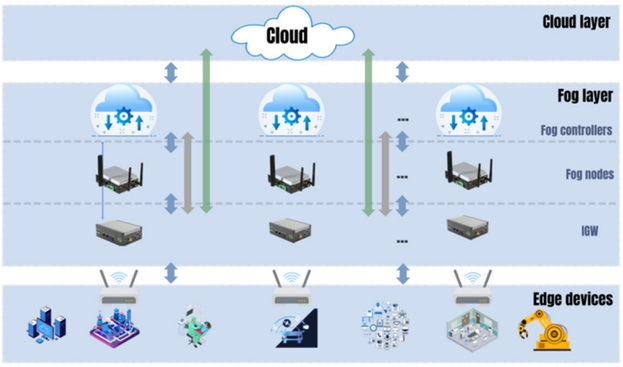

Fig. 1. Cloud–Fog–Edge devices infrastructure architecture

Despite the promise of combining FL with fog computing for 6G, practical deployment faces substantial challenges. Chief among these is heterogeneity across fog nodes: differing instruction set architectures (ISAs), diverse operating systems, and disparities in computational resources, CPU speeds, and RAM capacities—complications that are amplified by 6G's demands for high integration and multi-platform interoperability. Moreover, in fog computing environments, edge devices often operate under severe compute and network constraints. Fog nodes such as IoT devices, mobile devices, or edge servers may have weak CPUs, limited RAM (from a few megabytes to a few gigabytes), and unstable connectivity with low bandwidth or high latency. These constraints complicate FL deployment, as local training is resource-intensive for data processing and model updates, while communication of updates can congest the network [6].

Security is also a critical factor in user-facing service deployments. Although FL in fog computing keeps raw data on the device rather than sending it to a central location, inference attacks, poisoning attacks, and denial-of-service attacks remain significant risks.

To help address this problem, we propose applying Wasm to the deployment of federated learning in fog computing environments. Originally designed as a high-performance compilation target for web applications, Wasm has rapidly evolved into a versatile technology for out-of-browser settings, including fog computing, edge computing, and the Internet of Things (IoT). A principal advantage of deploying federated learning with Wasm is its high performance coupled with a compact bytecode footprint, which reduces the load on constrained computing resources by enabling fast execution without per-device recompilation. This is particularly beneficial for resource-limited fog nodes, where AI models can run efficiently without exhausting CPU or RAM. Moreover, the flexibility of Wasm, particularly its integration with lightweight runtimes such as Wasmtime or WasmEdge, makes it suitable for energy-constrained environments. This allows edge devices to operate for extended periods while ensuring safe and effective on-device processing of sensitive data. From a security perspective, adopting Wasm for federated learning in fog computing provides a substantial additional layer of protection. Wasm is built around a sandboxed security model, meaning that bytecode executes within a strictly isolated environment that prevents malicious code from accessing the host system, external memory, or a device's sensitive data. This design helps mitigate risks arising from vulnerability exploits or injection attacks, particularly on heterogeneous and attack-prone fog nodes.

This paper evaluates the feasibility and effectiveness of using Wasm as a platform for deploying FL in FC for 6G networks. Building on an analysis of key challenges - heterogeneous fog nodes, resource constraints, and security risk - we propose an integrated Wasm-based approach. The main contributions are as follows:

- Wasm-based FL deployment framework. We present a detailed framework that integrates FL with WebAssembly in fog computing, leveraging advantages such as high performance with compact bytecode, cross-platform deployment without recompilation, and integration with lightweight runtimes including Wasmtime and WasmEdge.
- Implementation and experiments on real devices. To validate the approach, we implement an experimental prototype emulating a fog environment on three devices: two Raspberry Pi 4 units and a Xiaomi 13 (Android), representing diverse architectures, operating systems, and resource profiles. FL training is performed on-device; parameter updates are transmitted over the network and aggregated centrally. The results show that Wasm achieves shorter processing times than native code while delivering near-equivalent performance, with reduced resource and memory

consumption. This makes it suitable for weaker fog nodes and supports stable operation under unstable network conditions.

• Performance analysis and future directions. We collect and analyze metrics from fog nodes, evaluating accuracy, loss, training time, and CPU/RAM utilization. Based on the results and related work, we outline extensions such as integrating advanced encryption and optimization for IoT in 6G networks.

The paper is organized into five parts: Section 1 introduces the background, the challenges of fog computing and FL in 6G, the proposed integration of WebAssembly, the objectives, and the contributions. Section 2 summarizes related work. Section 3 describes the architecture for deploying FL with WebAssembly in fog environments. Section 4 presents the experiments and evaluation on Raspberry Pi 4 and Xiaomi 13 devices, including performance analysis and trade-offs. Section 5 discusses security considerations and future research directions.

2 Overview of the Research Landscape

This section reviews the existing literature to delineate the context and the research gap that this work targets within the development of 6G networks. A central challenge is the need for distributed processing and for the deployment capabilities and scalability of AI models on heterogeneous, resource-constrained devices. The analysis is structured around four themes: (i) system-level challenges for federated learning (FL) at the edge; (ii) current approaches to managing heterogeneity; (iii) the role of Wasm as a universal runtime; and (iv) the positioning of the proposed framework's novelty.

2.1 System Challenges in Federated Learning within Fog Environments

Deploying FL in distributed settings such as fog computing faces a range of inherent system-level challenges that have been widely documented in survey literature.

• System and resource heterogeneity. This is among the most significant barriers. Fog and edge nodes differ substantially in hardware (CPU, memory), software (operating systems, ISAs), and network capabilities [7]. Such heterogeneity complicates deployment and leads to performance issues, for example, "stragglers" (slow nodes) that degrade the convergence speed of the overall FL system [8]. Moreover, these devices often operate under severe resource constraints, requiring algorithms and models to be highly efficient in computation and memory.

• Security and privacy: While FL is designed to protect privacy by keeping data local, other threats persist. Model updates can be attacked to infer sensitive training information (inference attacks) or maliciously altered to corrupt the global model (poisoning attacks). Consequently, a secure execution environment at each node is crucial.

2.2 Current Approaches to Managing Heterogeneity

Several approaches have been adopted to cope with heterogeneity, each with trade-offs in the context of fog computing.

• Cross-compilation and native deployment: This traditional approach compiles source code separately for each target platform. Although it can deliver optimal performance, it lacks portability and scalability. Maintaining distinct toolchains and build pipelines for every {ISA, OS} pair is costly and error-prone when managing a diverse device fleet [9].

• Virtualization and containerization: Container technologies such as Docker provide a consistent execution environment by packaging applications with their dependencies, effectively mitigating "dependency hell." However, this comes with notable resource costs. Containers often exhibit slower startup times and larger memory and disk footprints, making them ill-suited to resource-constrained IoT and edge devices [10]. In addition, containers remain architecture-dependent, typically requiring separate images for different architectures (e.g., one for ARM and another for x86).

• Existing FL frameworks: Popular frameworks such as TensorFlow Federated (TFF), PySyft, and Flower offer powerful tools for building the algorithmic logic of FL. Nevertheless, they primarily target the algorithmic layer and provide limited support for system-level deployment challenges. These frameworks often assume that execution environments are pre-provisioned on client devices, and support for less common architectures such as ARM may be limited or not prioritized.

2.3 WebAssembly: A Universal Runtime for Fog Computing

Wasm is emerging as a foundational technology capable of addressing the foregoing issues. Originally designed for the browser, Wasm has rapidly been recognized as a versatile, portable, and secure compilation target for out-of-browser environments [11].

• Standalone runtimes: The maturation of standalone runtimes such as Wasmtime (from the Bytecode Alliance) and WasmEdge (a CNCF project) is pivotal. These runtimes are optimized for server-side and edge applications, deliver high performance via JIT/AOT compilation, and support multiple platforms across both x86 and ARM architectures [12].

• WebAssembly System Interface (WASI): WASI is a core component that makes Wasm useful beyond the browser. It specifies a standardized, platform-independent system interface that enables Wasm modules to interact safely and in a controlled manner with host operating system resources (e.g., file systems, networking, clocks). Notably, WASI adopts capability-based security [13]. By default, a Wasm module has no permissions; the host must explicitly grant "capabilities" (e.g., access to a specific directory). This enforces the Principle of Least Privilege in a natural and robust way.

2.4 Positioning of This Work and the Research Gap

Building on the above analysis, this work is positioned to fill a critical research gap. Its distinctiveness and novelty are best understood in comparison with recent studies and the evolution of the Wasm ecosystem.

A clear bifurcation is emerging in the application of Wasm to machine learning: a web-centric approach and an infrastructure-centric approach. The web-centric direction is exemplified by the recent FLAT framework [14], which aims to allow any device with a browser to join an FL cluster without installation or configuration. FLAT addresses user accessibility and ease of participation; the browser acts as both runtime and sandbox, abstracting away OS/ISA complexity.

By contrast, this work pursues an infrastructure-centric

direction. The objective is to address the challenge of deploying a single codebase across headless fog nodes (e.g., Raspberry Pi or edge servers) that lack user interfaces. The problem tackled here concerns operational and infrastructural complexity for system administrators. In the proposed framework, the Wasm runtime (Wasmtime) is an explicit abstraction layer that must be deployed and managed on the host OS. Consequently, the focus of this study is on system-level performance relative to native code. Table 1 summarizes and contrasts the key characteristics of the four approaches discussed above, highlighting the distinct positioning of the proposed infrastructure-centric Wasm-FL framework.

In addition, the current Wasm ecosystem focuses heavily on accelerating inference through the WASI-NN specification [15], while offering limited support for training. This creates an important and urgent research gap. In a related line of work, the authors previously proposed a framework combining FL and fog computing using client sampling and dynamic thresholding techniques [16], which further motivates the present investigation into Wasm as a portable execution substrate for such systems. The present work contributes a potential solution by providing a performance baseline and a cross-platform operational model, addressing whether FL training with Wasm support in fog environments is feasible from a performance standpoint.

Table 1

Comparative Analysis of Techniques for Managing Heterogeneity in Federated Learning

Criterion	Native (Cross-Compilation)	Web-centric Wasm (FLAT)	Infrastructure-centric Wasm-FL (Proposed)
Portability	Very low. Requires recompilation for each {ISA, OS} pair	Very high. Runs in any modern browser	Very high. Single .wasm, any Wasm runtime
Safety & Isolation	Low. Executes directly on the host OS	Very high. Browser sandbox	Very high. Wasm sandbox, WASI capability-based security
Startup Time	Very fast	Fast	Very fast
Memory Footprint	Very small	Medium	Very small
Performance	Optimal (baseline)	Lower than native due to browser environment	Near-native
Target Environment	Servers, embedded devices	Web browsers, end-user devices	Headless fog/edge infrastructure nodes

3 Proposed Framework

This section presents a Wasm-based federated learning framework deployed in fog computing environments. It is designed to address device heterogeneity and execution safety, and to alleviate bottlenecks in IoT and telecommunications networks. The framework adopts the standard FedAvg paradigm, in which fog nodes act as collection points for data from edge devices and use Wasm as a portable, secure execution layer for local training.

3.1 System Model

We propose a three-tier physical FL architecture comprising a Sensor layer, a Fog layer, and a Cloud layer, as depicted in Figure 2, where:

- *Sensor Layer:* This layer consists of data-acquisition devices (e.g., cameras and sensors). We assume these sensors are locally connected to fog nodes (via LAN, USB, or LoRa) within communication range and continuously stream raw data to their associated fog node.
- *Fog Layer:* This layer corresponds to the clients in FedAvg and includes a set of fog nodes Fog $\mathcal{F}=\{1,\ldots,\mathrm{F}\}$, where $F=|\mathcal{F}|$. Each fog node $\mathrm{f}\in\mathcal{F}$ (Raspberry Pi 4, NVIDIA Jetson,…) is responsible for collecting and locally storing the entire dataset D_f from the sensors within its domain. The dataset D_f is considered private and never leaves node *f*. Each fog node runs a Wasm runtime to execute the training module dispatched from the Cloud.
- *Cloud Layer:* A central server (orchestrator) coordinates the FedAvg process: selecting clients, distributing the model, aggregating updates, and redistributing the global model.

The central problem in federated learning is to find the global parameter vector *w** that optimizes the global loss *F(w)* over data distributed across fog nodes, without sharing raw data among nodes or with the Cloud. Formally,

$$\min_{w} F(w)=\sum_{f=1}^{F}\frac{N_f}{N}F_f(w), N=\sum_{f=1}^{F}N_f \qquad (1)$$

Where, $F_f(w)=\frac{1}{N_f}\sum_{(x,y)\in D_f}\ell(w;x,y)$ is the local loss on the private dataset D_f (of size $N_f=|D_f|$) at fog node *f*, $\ell(w;x,y)$ denotes the loss for a single sample, *F* is the number of fog nodes.

This constitutes a distributed optimization problem under data-privacy and heterogeneity constraints. Data at each fog node need not follow the global distribution and may vary in size; furthermore, fog nodes differ in instruction set architectures (ISAs), run different operating systems, and exhibit substantial disparities in computational resources, CPU speeds, and RAM capacities.

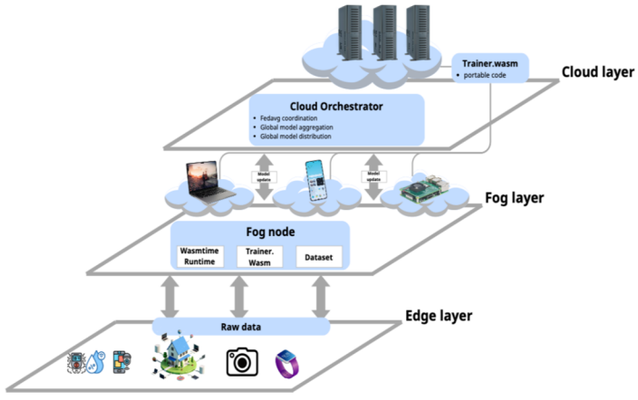

Fig. 2. WebAssembly-integrated system architecture

3.2 Wasm: a new, core component of the system

Wasm is, by design, a low-level machine-code format with standardized semantics, engineered to be independent of any specific embedded platform. This property enables the same module to execute across diverse architectures with deterministic and consistent behavior. In this study, the system is built to run on Wasmtime – a high-performance, standalone runtime developed by the Bytecode Alliance, a nonprofit whose participants include Mozilla, Fastly, Intel, and Red Hat. Wasmtime is designed around four core principles that make it well suited for production deployments [17]:

• *Fast:* Wasmtime integrates an optimizing compiler to produce high-quality machine code and supports both just-in-time (JIT) and ahead-of-time (AOT) compilation. The runtime is specifically optimized for rapid instance initialization and for minimizing overhead in host-guest function calls.

• *Secure:* Implementation in Rust provides a strong foundation for memory safety. The project undergoes continuous fuzz testing funded by Google OSS-Fuzz and follows a rigorous security-response process to patch vulnerabilities promptly.

• *Configurable:* Wasmtime exposes fine-grained configuration options for resource control, allowing limits on CPU and memory consumption per Wasm instance. This makes the runtime suitable for a wide range of deployment environments – from resource-constrained embedded devices to large servers handling many concurrent instances [18].

• *Standards Compliant:* Wasmtime passes the official WebAssembly test suite, and its developers actively participate in standardization efforts. A key commitment is to avoid proprietary extensions, thereby ensuring portability and compatibility of Wasm modules.

To support production readiness, Wasmtime classifies platform support into three tiers, enabling developers to assess deployment risk (Table 2).

Table 2

Wasmtime platform-support tiers

Tier	Description	Example platforms
Tier 1	Production-ready, fully tested, and comprehensively supported	x86_64-unknown-linux-gnu, x86_64-apple-darwin, x86_64-pc-windows-msvc
Tier 2	Near production-ready, well maintained but may lack some Tier-1 criteria	aarch64-unknown-linux-gnu, aarch64-apple-darwin
Tier 3	Supported but not widely tested; not recommended for production	riscv64gc-unknown-linux-gnu, x86_64-unknown-freebsd, aarch64-linux-android

By design, Wasm is a fully isolated computing environment. A Wasm module, by default, cannot perform I/O operations such as reading files, opening network connections, or accessing the system clock [19]. This isolation underpins Wasm's security model but also poses obstacles for practical applications. WASI was developed to address this issue by defining a standardized, platform-independent system interface that provides APIs for safe, controlled interaction between a Wasm module and the outside world. The goal of WASI is to extend Wasm's "write once, run anywhere" promise from the CPU level to the system level, enabling a single module to interact with different operating systems through the same API set.

The core design principle of WASI is capability-based security. Unlike traditional access models (e.g., POSIX), in which a program runs with the user's privileges and can attempt to access any resource, WASI adopts an inverted model: a Wasm module starts with zero authority and no ambient privileges. To access a resource, the host runtime must explicitly grant the module a capability – an unforgeable handle to that resource. The module can act only on resources for which it holds such a handle. This mechanism forces developers to specify required permissions up front and naturally enforces the Principle of Least Privilege [20], while eliminating the "confused deputy" problem common in traditional security systems.

Although Wasm supports multiple source languages (e.g., C/C++ and AssemblyScript), Rust remains the most popular choice due to its balance of performance, memory safety, and usability. The Rust-Wasm pairing is often described as ideal, not only for technical reasons but also for alignment in design philosophy.

• *Memory safely without a Garbage Collector:* Rust guarantees memory safety at compile time via ownership and the borrow checker, eliminating the need for a runtime garbage collector (GC). This aligns with Wasm, which is likewise designed without a built-in GC. Compiling Rust to Wasm yields compact modules that do not carry a bulky GC runtime, reducing binary size, speeding load times, and avoiding unpredictable pauses – thereby delivering stable performance [21].

• *High performance and Zero-Cost Abstractions:* Rust is engineered for performance on par with C/C++. Its hallmark "zero-cost abstractions" (e.g., iterators, closures, Option, Result) are fully optimized by the compiler into efficient machine code with no runtime overhead – matching Wasm's objective of near-native performance.

• *Mature tooling ecosystem:* The Rust community provides comprehensive Wasm tooling. Utilities such as wasm-pack automate build and packaging workflows (including npm-compatible bundles), while wasm-bindgen generates the glue code for seamless Rust-JavaScript interoperation. The cargo package manager simplifies dependency management.

• *A strategic fit across the stack:* Rust is well suited both for writing Wasm modules (guest code) and for building secure, high-performance Wasm runtimes (host code), with Wasmtime being a canonical example [22]. This enables a unified ecosystem in which developers can use the same language and tooling across the entire application stack.

The pipeline from Rust source to an executable WASI binary is illustrated in Figure 3.

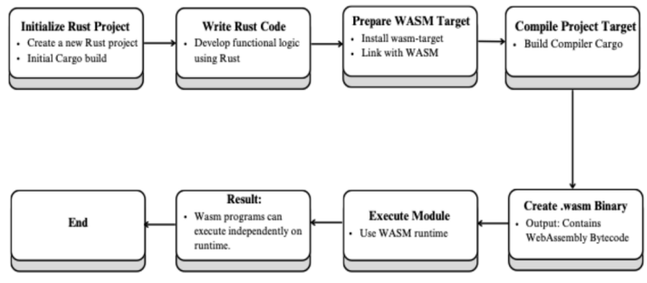

Fig. 3. Compilation pipeline from Rust source to a WASI binary

This pipeline highlights that the Rust-to-WASM toolchain is in a phase of transition and continued maturation. While tools such as wasm-pack are optimized for the web context and IoT edge-computing environments [23], compiling for standalone runtimes like Wasmtime is simpler and typically requires only standard cargo commands. This reflects the growing maturity of WASM as a first-class compilation target within the Rust ecosystem.

Building on the system components described above, the overall execution flow of the proposed framework is organized into the following steps.

• *Step 1 – Initialization.* The Cloud server initializes the global model parameters and compiles the training logic into a portable Wasm module (trainer.wasm) along with a configuration file (config.json) specifying the learning rate, batch size, and number of local epochs.

• *Step 2 – Client selection*. At the beginning of each communication round, the server selects a subset of available fog nodes based on a predefined participation policy.

• *Step 3 – Model and module distribution.* The server broadcasts the current global model parameters and the Wasm training module to all selected fog nodes.

• *Step 4 – Local training within Wasm sandbox.* Each fog node loads the received model and executes local training entirely within a Wasm sandbox. The sandbox enforces capability-based access control, granting the module read access to the local dataset directory and write access to the model output directory only, while blocking all other system resources. The private dataset never leaves the node.

• *Step 5 – Parameter transmission.* Upon completion of local training, each fog node transmits only the updated model parameters and the local dataset size back to the server. No raw data is shared.

• *Step 6 – Aggregation.* The server aggregates the received updates through weighted averaging proportional to each node's dataset size, producing an updated global model.

• *Step 7 – Iteration.* Steps 2 through 6 are repeated for a predefined number of communication rounds until the model converges or the round limit is reached.

4. Implementation and Performance Evaluation

This section details the experimental setup and analyzes results to quantitatively assess the effectiveness of the Wasm–based FL deployment framework relative to a native execution baseline. The objective is to answer the research questions posed: (i) the ability to preserve the convergence trajectory, (ii) system costs (time and resources), and (iii) factors influencing performance.

4.1. Experimental environment design

The experiments were conducted in the MegaNetlab research laboratory at The Bonch-Bruevich Saint Petersburg State University of Telecommunications [24]. The setup emulates federated learning in a fog-computing environment under two conditions: training on the Wasm platform and training with standard native code. The system follows a three-tier Cloud–Fog–Client architecture, in which:

• Cloud (Server): Acts as the central orchestrator running the FedAvg aggregation algorithm, distributing the global model and aggregating updates from fog nodes. The server is deployed on a MacBook Pro M1 with an 8-core CPU, 14-core GPU, 32 GB RAM, and a 1 TB SSD, ensuring fast processing of model aggregation tasks.

• Fog: Comprises three heterogeneous fog nodes in both hardware and operating systems: two Raspberry Pi 4 units (ARM64, Linux) and a Xiaomi 13 smartphone (ARM, Android). This diversity is intentionally chosen to represent the typical heterogeneity of real fog environments. Fog nodes perform local training on their private data and send gradient updates to the server.

• Sensor/Edge: Emulates raw data sources for the fog nodes. In this study, data are pre-partitioned for each fog node to simulate local collection from sensors.

Figure 4 illustrates the actual deployment configuration, where the three fog devices are connected to the server over a LAN, forming a complete FL system.

The experiment is designed with reasonable assumptions to focus on the core performance evaluation of Wasm. Communication between Cloud and Fog uses gRPC/TCP; no raw data are transmitted to the Cloud – only model updates (31.4 KB per direction per round). Fog nodes are assumed to have stable network connections to the server to eliminate noise from uncertain network latency, enabling precise measurement of pure computational overhead.

The data model is MNIST (10 classes), a standard dataset in FL research. Data are partitioned in a non-IID manner using a Dirichlet distribution with α=0.5 to emulate realistic imbalance: each fog node holds a subset with label skew. Specifically, nodes receive different numbers of samples and are concentrated on certain labels, reflecting characteristics of real-world distributed data. The learning model is multiclass logistic regression (Softmax) with 7,850 parameters (784 × 10 weights + 10 biases). This simple model is chosen to isolate and evaluate the overhead of the WebAssembly layer without confounding effects from optimizer complexity. The compact model size (31.4 KB) further minimizes the impact of network bandwidth on measurements.

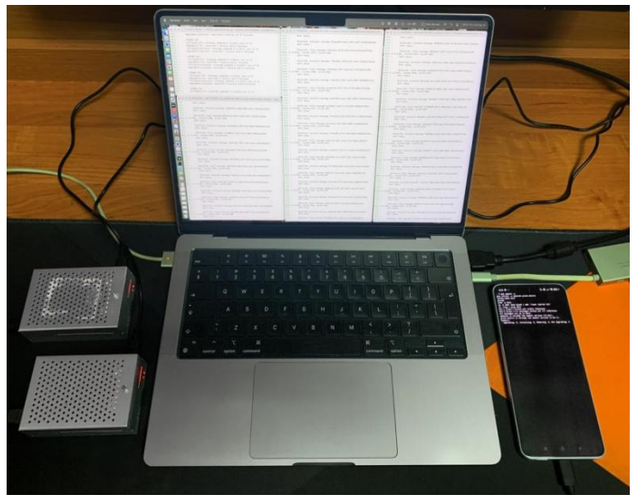

Fig. 4. Experimental deployment on physical devices

The core difference in the experiment lies in the execution layer. Local training code is written in Rust and compiled along two paths for comparison:

● Baseline (Native): Compiled directly to machine code for each specific architecture (aarch64 for Raspberry Pi/Xiaomi; x86_64 for other platforms).

● WebAssembly: Compiled to the wasm32-wasi target to produce the portable module trainer.wasm, which is then executed via the Wasmtime runtime on all fog nodes.

The Wasm environment is configured with strict capability-based security. The module is granted only the permissions needed to read data from a designated directory and write the model to an output directory; it has no access to the network or other file-system areas. This sandboxing ensures safety even if the training code fails or contains malicious components.

The evaluation metrics collected directly include: accuracy and loss for convergence; run time, throughput (samples/sec), maximum resident set size (max RSS), and initialization latency (elapsed sec) for system performance; as well as estimated CPU active time and CPU utilization (%CPU) for resource assessment.

4.2. Evaluation Results and Analysis

Accuracy: Experimental results show that both deployment methods achieve comparable convergence trajectories in terms of accuracy. After 10 training rounds, the native approach attains an average accuracy of 90.15% (±0.12%), while the Wasm approach reaches 89.86% (±0.15%), a difference of only 0.29%. Extending to 50 rounds, this gap narrows to 0.04% (Native: 91.25% ±0.08%, Wasm: 91.29% ±0.10%). At 100 rounds, Wasm achieves 91.49% (±0.09%) versus 91.49% (±0.07%) for Native, indicating high stability over long-horizon convergence (Fig. 5, 6). The small discrepancy may stem from numerical precision in the Wasm environment, but it does not materially affect overall performance.

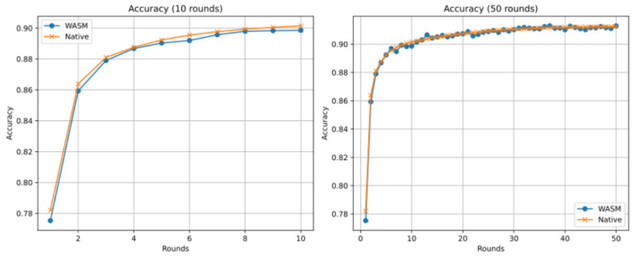

Fig. 5. Accuracy across training rounds (10 and 50 rounds)

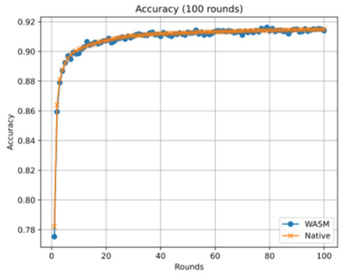

Fig. 6. Accuracy across training 100 rounds

Loss function: The loss-reduction trends of the two methods are nearly identical over training rounds (Fig. 7, 8). At round 10, Native records a loss of 0.339 (±0.005) versus 0.346 (±0.006) for Wasm. By round 50, both converge to approximately 0.296 (Native: 0.296 ±0.004, Wasm: 0.296 ±0.005). After 100 rounds, Native's loss is 0.286 (±0.003) and Wasm's is 0.289 (±0.004), a negligible difference. These results demonstrate that using Wasm does not affect the correctness of the FL algorithm or the model's ability to converge.

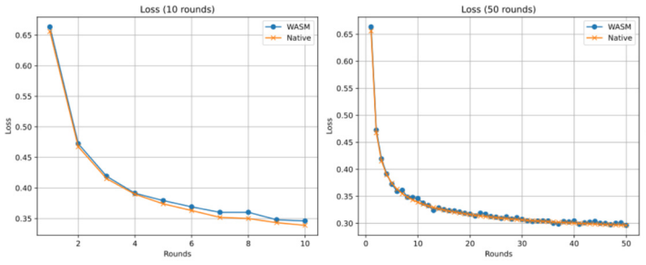

Fig. 7. Loss across training rounds (10 and 50 rounds)

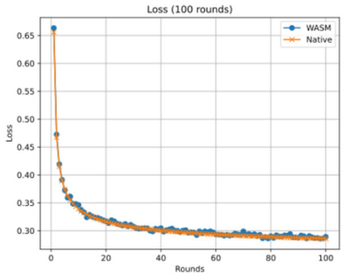

Fig 8. Loss across training 100 rounds

Training time results (Fig. 9) show that Wasm incurs a wall-clock overhead of 64.6% at 10 rounds (132.60 s vs. 80.51 s for Native), which decreases to 23.1% at 50 rounds (310.84 s vs. 252.45 s) and further to 10.2% at 100 rounds (514.00 s vs. 466.63 s). This trend reflects the amortization effect: as the number of rounds increases, the fixed per-round Wasm initialization cost occupies a progressively smaller share of the total training time, making Wasm increasingly time-competitive for longer workloads.

Throughput analysis (Fig. 10) shows that Native processes 120,990 samples/s (±5,120) at 10 rounds versus 51,460 samples/s (±1,230) for Wasm, a ratio of 2.35×. This ratio remains stable at 50 rounds (2.43×) and 100 rounds (2.46×), corresponding to a consistent computational throughput overhead of 135–146%. The stability of this ratio across all scenarios confirms that the throughput gap is a structural characteristic of the Wasm execution layer.

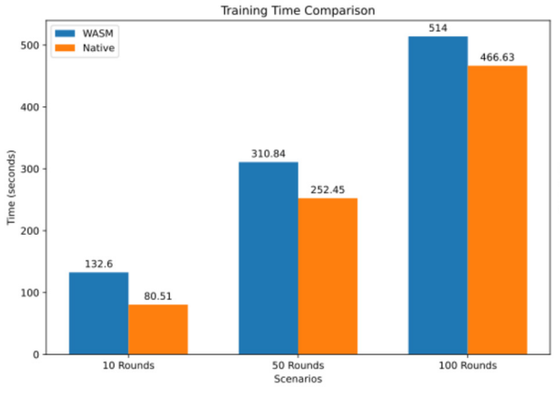

Fig. 9. Traning time comparison

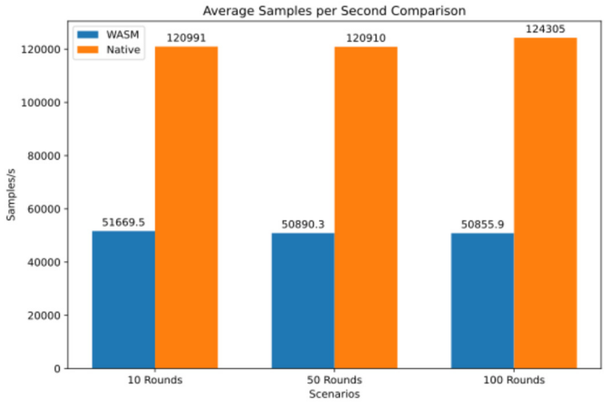

Fig. 10. Averege samples per second comparision

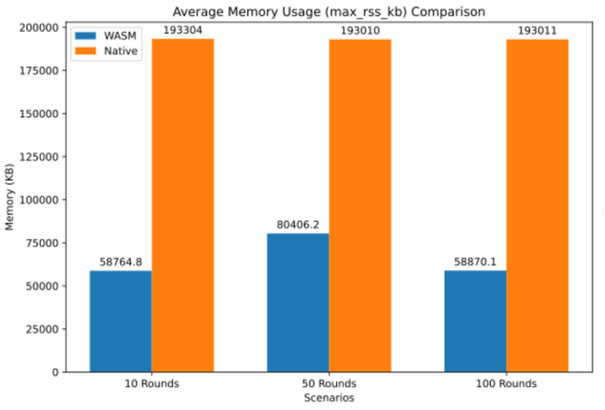

Fig. 12. Average memory usage comparison

The average per-round initialization time for Wasm (2.02 s ±0.04 s at 10 rounds) exceeds that of Native (1.57 s ±0.03 s), but the gap narrows at larger scales (Fig. 11): Wasm 2.03 s (±0.05 s) vs Native 1.58 s (±0.04 s) at 50 rounds; Wasm 2.02 s (±0.05 s) vs Native 1.59 s (±0.04 s) at 100 rounds. This may reflect Wasm's ahead-of-time compilation path, which reduces startup time compared with loading and compiling traditional native code. The average initialization latency across all scenarios is presented in Fig. 12.

RSS memory (Resident Set Size) analysis shows a pronounced advantage for Wasm in memory efficiency. Native consumes 193.0 MB on average (±2.5 MB) and remains stable across scenarios. In contrast, Wasm uses only 58.8 MB on average (±1.2 MB), saving 69.5% of memory relative to Native. As illustrated in Fig. 12, this gap is consistent across all training scenarios (10, 50, and 100 rounds), confirming the stability of Wasm's memory efficiency advantage. Specifically:

- 10 rounds: Wasm 58.8 MB vs Native 193.3 MB (69.6% savings)
- 50 rounds: Wasm 58.8 MB vs Native 193.0 MB (69.5% savings)
- 100 rounds: Wasm 58.9 MB vs Native 193.0 MB (69.5% savings)

This benefit is particularly important in fog-computing settings, where edge devices often have limited memory resources (e.g., Raspberry Pi, Gateway…).

Estimated CPU active time (Fig. 13) shows Wasm exhibiting about 25% higher active time than Native (due to runtime overhead), yet with a lower max total %CPU: Wasm ~22-25% vs Native 82–90% (Fig. 13, 14). This indicates that Wasm is better suited to resource-constrained environments such as Fog and Edge computing.

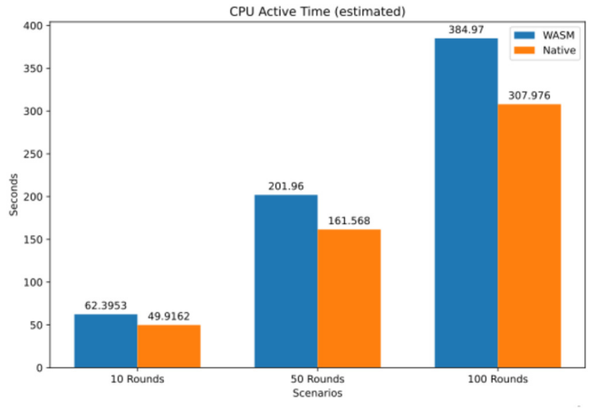

Fig. 13. CPU active time

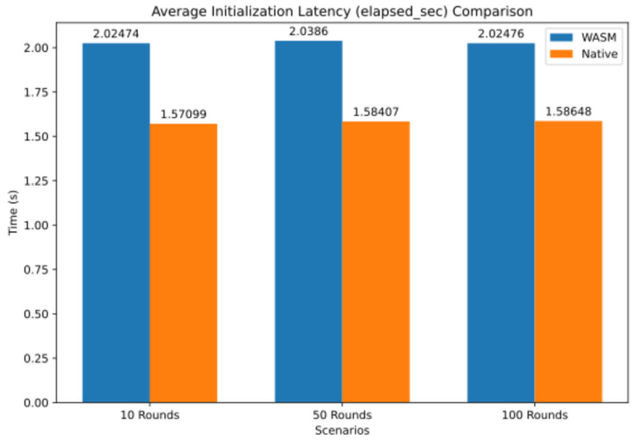

Fig. 11. Average initialization latency comparison

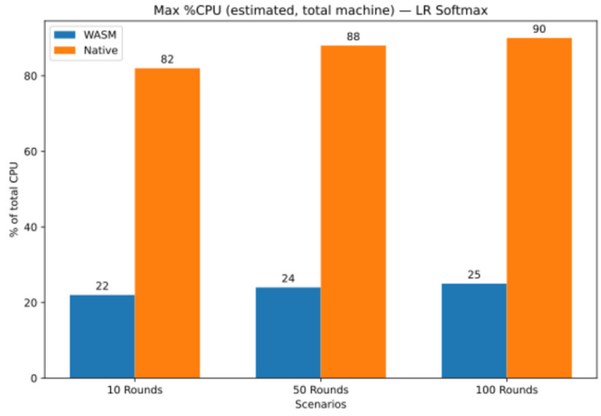

Fig. 14. Max %CPU during training

5. Conclusion

This study presented a Wasm-based framework for deploying federated learning in fog-computing environments, addressing device heterogeneity, resource constraints, and system security. Through theoretical analysis and empirical evaluation on physical devices (Raspberry Pi 4 and Xiaomi 13), we showed that the proposed framework preserves a convergence trajectory equivalent to native execution, with nearly identical accuracy and loss (difference <0.3% after 100 training rounds). At the same time, Wasm offers notable resource-efficiency benefits: a 69.5% reduction in memory usage (58.8 MB versus 193 MB for native) and lower CPU utilization (22-25% versus 82-90%), despite a wall-clock training overhead that decreases from 64.6% at 10 rounds to 10.2% at 100 rounds as initialization costs are amortized, and a stable computational throughput overhead of 135-146% reflecting the intrinsic cost of the Wasm execution layer. These results affirm Wasm as a portable and secure execution layer and clarify the performance–resource trade-offs useful to developers and architects when deploying privacy-preserving distributed AI at the edge/fog in 6G contexts.

Nevertheless, several limitations remain. The evaluation focused on a simple model (multiclass logistic regression on MNIST) and assumed a stable network; it did not measure or analyze energy consumption, an especially important factor for edge devices, which may limit representativeness for complex real-world scenarios. To address these limitations and enhance practicality, several directions are identified for future work.

First, leveraging modern Wasm features represents a natural next step. Wasm support for SIMD (Single Instruction, Multiple Data) enables vectorization of matrix operations and may substantially narrow the performance gap with native code. Additionally, ahead-of-time (AOT) compilation can eliminate JIT-related overhead and improve throughput in production settings.

Second, reducing communication cost remains a priority as the framework scales. Applying gradient compression techniques such as quantization and sparsification, together with communication-efficient FL methods including FedProx and SCAFFOLD, would lower bandwidth requirements when working with larger models. Delta encoding, in which only weight changes rather than full model parameters are transmitted, is another promising direction.

Third, extending support to more complex models will broaden the framework's applicability. Deep architectures such as CNNs (MobileNet, EfficientNet) for computer vision tasks and RNN/LSTM networks for time-series forecasting are natural candidates, requiring optimization of numerical libraries in Rust and improved multi-dimensional tensor handling within Wasm.

Fourth, deploying the framework in real-world domains will validate its practical value. Concrete application areas include healthcare (training diagnostic models on decentralized patient data), smart cities (traffic analytics and energy management), and Industry 4.0 (predictive maintenance), each of which presents distinct requirements in terms of latency, security, and scale.

Fifth, strengthening attack resistance will be essential for production deployment. Integrating Differential Privacy to perturb gradients and mitigate inference attacks, combined with robust aggregation mechanisms such as Krum and coordinate-wise median to detect Byzantine nodes, would complement the existing capability-based security provided by WASI and establish a layered defense architecture.

Overall, this research supports Wasm as a viable foundational technology for federated learning on heterogeneous fog nodes, striking a practical balance among performance, portability, and security. Wasm not only offers a new approach to building distributed AI systems but also advances the realization of fog computing combined with FL in the era of distributed AI. The proposed future directions lay the groundwork for deeper integration with telecommunications and IoT infrastructure and broader 6G platforms.

References

[1] S. Iftikhar, S. S. Gill, C. Song, M. Xu, M. S. Aslanpour, A. N. Toosi, *et al*., “AI-Based Fog and Edge Computing: A Systematic Review, Taxonomy and Future Directions,” *Internet of Things*, vol. 21, p. 100674, 2023.

[2] A. A. Ateya, A. A. A. El-Latif, A. Muthanna, A. Volkov, and A. Koucheryavy, *Enabling Metaverse and Telepresence Services in 6G Networks*. New York: River Publishers, 2025, 254 p., doi: 10.1201/9788770046749.

[3] I. Stojmenovic, S. Wen, X. Huang, and H. Luan, “An Overview of Fog Computing and Its Security Issues,” *Concurrency and Computation: Practice and Experience*, vol. 28, no. 10, pp. 2991–3005, 2016.

[4] D. V. Thang, A. Volkov, A. Muthanna, A. Koucheryavy, A. A. Ateya, and D. N. K. Jayakody, "Future of telepresence services in the evolving fog computing environment: A survey on research and use cases," *Sensors*, vol. 25, no. 11, p. 3488, 2025, doi: 10.3390/s25113488.

[5] L. Li, Y. Fan, M. Tse, and K. Y. Lin, “A Review of Applications in Federated Learning,” *Computers & Industrial Engineering*, vol. 149, p. 106854, 2020.

[6] B. T. Hasan and A. K. Idrees, “Federated Learning for IoT/Edge/Fog Computing Systems,” in *Federated Learning*. Palm Bay, FL, USA: Apple Academic Press, 2024, pp. 47–75.

[7] Q. Xia, W. Ye, Z. Tao, J. Wu, and Q. Li, “A Survey of Federated Learning for Edge Computing: Research Problems and Solutions,” *High-Confidence Computing*, vol. 1, no. 1, p. 100008, 2021.

[8] L. I. Barona López and T. Borja Saltos, “Heterogeneity Challenges of Federated Learning for Future Wireless Communication Networks,” *Journal of Sensor and Actuator Networks*, vol. 14, no. 2, p. 37, 2025.

[9] H. G. Abreha, M. Hayajneh, and M. A. Serhani, “Federated Learning in Edge Computing: A Systematic Survey,” *Sensors*, vol. 22, no. 2, p. 450, 2022.

[10] “Unleashing Edge AI with WebAssembly: Performance, Portability, and a Hands-On Guide,” *DEV Community*. [Online]. Available: https://dev.to/vaib/unleashing-edge-ai-with-webassembly-performance-portability-and-a-hands-on-guide-p7o (accessed Oct. 28, 2025).

[11] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, “Research on WebAssembly Runtimes: A Survey,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–47, 2025.

[12] S. E. Khelifa, M. Bagaa, A. O. Messaoud, and A. Ksentini, “Case Study of WebAssembly Runtimes for AI Applications on the Edge,” in *Proc. 2024 Global Information Infrastructure and Networking Symposium (GIIS)*, 2024, pp. 1–6.

[13] T. Heinrich, N. C. Will, R. R. Obelheiro, and C. A. Maziero, “Uso de Chamadas WASI para a Identificação de Ameaças em Aplicações WebAssembly,” in *Proc. Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, 2023, pp. 237–250.

[14] M. Garofalo, M. Colosi, A. Catalfamo, and M. Villari, “Web-Centric Federated Learning over the Cloud-Edge Continuum Leveraging ONNX and WASM,” in *Proc. 2024 IEEE Symposium on Computers and Communications (ISCC)*, 2024, pp. 1–7.

[15] J. Bachmeier, V. Yussupov, J. Henß, and H. Koziolek, “WebAssembly with WASI-NN for Edge Machine Learning Inference: Experiences and Lessons Learned,” in *European Conference on Software Architecture*, Cham, Switzerland: Springer Nature Switzerland, 2025, pp. 343–359.

[16] D. Van Thang, A. Volkov, A. Muthanna, I. A. Elgendy, R. Alkanhel, D. N. K. Jayakody, and A. Koucheryavy, "A framework

integrating federated learning and fog computing based on client sampling and dynamic thresholding techniques," *IEEE Access*, 2025, doi: 10.1109/ACCESS.2025.

[17] "Introduction – Wasmtime," *Wasmtime Documentation*. [Online]. Available: https://docs.wasmtime.dev/ (accessed Oct. 28, 2025).

[18] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using WebAssembly," in *Proc. 31st USENIX Security Symp. (USENIX Security '22)*, Boston, MA, Aug. 2022, pp. 1975–1992.

[19] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with WebAssembly runtimes," in *Proc. 22nd IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, 2022, pp. 140–149, doi: 10.1109/CCGrid54584.2022.00023.

[20] S. Berard, "Part One: Exploring WebAssembly to Power Secure, Portable Applications Spanning the Cloud to Tiny Edge Devices," *Atym*. [Online]. Available: https://www.atym.io/post/exploring-webassembly-to-power-secure-portable-applications-spanning-the-cloud-to-tiny-edge-devices (accessed Oct. 28, 2025).

[21] Hoffman, Kevin. *"Programming WebAssembly with Rust: unified development for web, mobile, and embedded applications,"* 2019: 1-220.

[22] L. Wagner, M. Mayer, A. Marino, A. Soldani Nezhad, H. Zwaan, and I. Malavolta, "On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT," in *Proc. 27th International Conference on Evaluation and Assessment in Software Engineering*, 2023, pp. 72–82.

[23] A. Prabakar and R. Kiran, "WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations," 2024.

[24] A. N. Volkov, A. S. A. Muthanna, A. E. Koucheryavy, A. S. Borodin, A. I. Paramonov, S. S. Vladimirov, et al., "Perspective research on networks and services 2030 in the 6G Meganetlab laboratory of SPbSUT," *Elektrosvyaz*, no. 6, pp. 5-14, 2023, doi: 10.34832/ELSV.2023.43.6.001.

ФРЕЙМВОРК РАЗВЁРТЫВАНИЯ ФЕДЕРАТИВНОГО ОБУЧЕНИЯ НА ОСНОВЕ WEBASSEMBLY ДЛЯ СРЕД ТУМАННЫХ ВЫЧИСЛЕНИЙ

Данг Ван Тханг, *Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича (СПбГУТ), Санкт-Петербург, Россия,* ***dangvanthang_sut@gmail.com***

Кучерявый Андрей Евгеньевич, *Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича (СПбГУТ), Санкт-Петербург, Россия,* ***akouch@mail.ru***

Аннотация

Федеративное обучение в среде туманных вычислений для сетей 6G открывает широкие возможности в различных прикладных областях. Локальность данных используется для защиты конфиденциальной информации, снижения коммуникационных затрат и разгрузки центрального сервера. Вместе с тем развёртывание задач ФО на туманных узлах с гетерогенными аппаратными конфигурациями, различными операционными системами и архитектурами набора инструкций остаётся серьёзной технической проблемой. Жёсткие требования 6G к задержке и тесная интеграция с приложениями реального времени дополнительно усугубляют её. WebAssembly (Wasm) представляет собой переносимый байткод-формат, способный преодолеть указанные ограничения в среде туманных вычислений 6G. Безопасность исполнения обеспечивается изолированной средой и разграничением полномочий на основе возможностей; производительность близка к нативной, объём модулей компактен, а запуск возможен на любой платформе без перекомпиляции. В настоящей работе представлен фреймворк развёртывания федеративного обучения на основе Wasm в среде туманных вычислений для сетей 6G. Практическая применимость и производительность решения оценены в экспериментах на гетерогенном наборе устройств. Полученные результаты свидетельствуют о том, что предложенный фреймворк снижает потребление оперативной памяти и загрузку процессора, сохраняя при этом высокую точность модели, что соответствует требованиям к производительности сетей 6G.

Ключевые слова: *WebAssembly, федеративное обучение, туманные вычисления, сети 6G, гетерогенные устройства.*

Литература

1. *Iftikhar S., Gill S.S., Song C., Xu M., Aslanpour M.S., Toosi A.N.* et al. AI-Based Fog and Edge Computing: A Systematic Review, Taxonomy and Future Directions // Internet of Things. 2023. Vol. 21. Art. 100674. DOI: 10.1016/j.iot.2022.100674.

2. *Ateya A.A., El-Latif A.A.A., Muthanna A., Volkov A., Koucheryavy A.* Enabling Metaverse and Telepresence Services in 6G Networks. New York : River Publishers, 2025. 254 p. DOI: 10.1201/9788770046749.

3. *Stojmenovic I., Wen S., Huang X., Luan H.* An Overview of Fog Computing and Its Security Issues // Concurrency and Computation: Practice and Experience. 2016. Vol. 28, № 10, pp. 2991-3005. DOI: 10.1002/cpe.3485.

4. *Thang D.V., Volkov A., Muthanna A., Koucheryavy A., Ateya A.A., Jayakody D.N.K.* Future of Telepresence Services in the Evolving Fog Computing Environment: A Survey on Research and Use Cases // Sensors. 2025. Vol. 25, № 11. Art. 3488. DOI: 10.3390/s25113488.
5. *Li L., Fan Y., Tse M., Lin K.Y.* A Review of Applications in Federated Learning // Computers & Industrial Engineering. 2020. Vol. 149. Art. 106854. DOI: 10.1016/j.cie.2020.106854.
6. *Hasan B.T., Idrees A.K.* Federated Learning for IoT/Edge/Fog Computing Systems // Federated Learning. Apple Academic Press, 2024, pp. 47-75.
7. *Xia Q., Ye W., Tao Z., Wu J., Li Q.* A Survey of Federated Learning for Edge Computing: Research Problems and Solutions // High-Confidence Computing. 2021. Vol. 1, № 1. Art. 100008. DOI: 10.1016/j.hcc.2021.100008.
8. *Barona Lopez L.I., Borja Saltos T.* Heterogeneity Challenges of Federated Learning for Future Wireless Communication Networks // Journal of Sensor and Actuator Networks. 2025. Vol. 14, № 2. Art. 37. DOI: 10.3390/jsan14020037.
9. *Abreha H.G., Hayajneh M., Serhani M.A.* Federated Learning in Edge Computing: A Systematic Survey // Sensors. 2022. Vol. 22, № 2. Art. 450. DOI: 10.3390/s22020450.
10. Unleashing Edge AI with WebAssembly: Performance, Portability, and a Hands-On Guide [Электронный ресурс] // DEV Community. URL: https://dev.to/vaib/unleashing-edge-ai-with-webassembly-performance-portability-and-a-hands-on-guide-p7o (дата обращения: 28.10.2025).
11. *Zhang Y., Liu M., Wang H., Ma Y., Huang G., Liu X.* Research on WebAssembly Runtimes: A Survey // ACM Transactions on Software Engineering and Methodology. 2025. Vol. 34, № 8, pp. 1-47. DOI: 10.1145/3714465.
12. *Khelifa S.E., Bagaa M., Messaoud A.O., Ksentini A.* Case Study of WebAssembly Runtimes for AI Applications on the Edge // Proc. 2024 Global Information Infrastructure and Networking Symposium (GIIS). 2024. P. 1-6. DOI: 10.1109/GIIS59465.2024.10449907.
13. *Heinrich T., Will N.C., Obelheiro R.R., Maziero C.A.* Uso de Chamadas WASI para a Identifica??o de Amea?as em Aplicacoes WebAssembly // Proc. Simp?sio Brasileiro de Seguran?a da Informa??o e de Sistemas Computacionais (SBSeg). 2023, pp. 237-250. DOI: 10.5753/sbseg.2023.233111.
14. *Garofalo M., Colosi M., Catalfamo A., Villari M.* Web-Centric Federated Learning over the Cloud-Edge Continuum Leveraging ONNX and WASM // Proc. 2024 IEEE Symposium on Computers and Communications (ISCC). 2024. P. 1-7. DOI: 10.1109/ISCC61673.2024.10733614.
15. *Bachmeier J., Yussupov V., Henss J., Koziolek H.* WebAssembly with WASI-NN for Edge Machine Learning Inference: Experiences and Lessons Learned // Software Architecture. Cham : Springer Nature Switzerland, 2025, pp. 343-359. DOI: 10.1007/978-3-032-02138-0_23.
16. *Van Thang D., Volkov A., Muthanna A., Elgendy I.A., Alkanhel R., Jayakody D.N.K., Koucheryavy A.* A Framework Integrating Federated Learning and Fog Computing Based on Client Sampling and Dynamic Thresholding Techniques // IEEE Access. 2025. DOI: 10.1109/ACCESS.2025.
17. Introduction - Wasmtime [Электронный ресурс] // Wasmtime Documentation. URL: https://docs.wasmtime.dev/ (дата обращения: 28.10.2025).
18. *Bosamiya J., Lim W.S., Parno B.* Provably-Safe Multilingual Software Sandboxing using WebAssembly // Proc. 31st USENIX Security Symposium (USENIX Security '22). Boston : USENIX Association, 2022, pp. 1975-1992. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya.
19. *Gackstatter P., Frangoudis P.A., Dustdar S.* Pushing Serverless to the Edge with WebAssembly Runtimes // Proc. 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2022., pp. 140-149. DOI: 10.1109/CCGrid54584.2022.00023.
20. *Berard S.* Part One: Exploring WebAssembly to Power Secure, Portable Applications Spanning the Cloud to Tiny Edge Devices [Электронный ресурс] // Atym. URL: https://www.atym.io/post/exploring-webassembly-to-power-secure-portable-applications-spanning-the-cloud-to-tiny-edge-devices (дата обращения: 28.10.2025).
21. *Hoffman K.* Programming WebAssembly with Rust: Unified Development for Web, Mobile, and Embedded Applications. Raleigh : Pragmatic Bookshelf, 2019. 240 p. ISBN 978-1-68050-636-5.
22. *Wagner L., Mayer M., Marino A., Soldani Nezhad A., Zwaan H., Malavolta I.* On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT // Proc. 27th International Conference on Evaluation and Assessment in Software Engineering (EASE). 2023, p. 72-82. DOI: 10.1145/3593434.3593454.
23. *Prabakar A., Kiran R.* WebAssembly Performance Analysis: A Comparative Study of C++ and Rust Implementations : B.S. thesis. Karlskrona : Blekinge Institute of Technology, 2024. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-26632.
24. *Волков А.Н., Мутханна А.С.А., Кучерявый А.Е., Бородин А.С., Парамонов А.И., Владимиров С.С.* и др. Перспективные исследования сетей и услуг 2030 в лаборатории 6G Meganetlab СПбГУТ // Электросвязь. 2023. № 6. С. 5-14. DOI: 10.34832/ELSV.2023.43.6.001.

Информация об авторах:

Данг Ван Тханг, *аспирант кафедры передачи сигналов, Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича (СПбГУТ), Санкт-Петербург, Россия, ORCID: https://orcid.org/0009-0009-2219-3767*

Кучерявый Андрей Евгеньевич, *профессор, заведующий кафедрой сетей связи и передачи данных, д.т.н., Санкт-Петербургский государственный университет телекоммуникаций им. проф. М.А. Бонч-Бруевича (СПбГУТ), Санкт-Петербург, Россия, ORCID: https://orcid.org/0000-0003-4479-2479*