

Article

An Edge-Cloud Infrastructure for Scalable Personalization of AI Models in Community Health Care

Rakesh Suvvari ¹, Awatif Yasmin ¹, Sana Alamgeer ¹, Tarek Mahmud ² and Anne H. H. Ngu ^{1,*}

¹ Department of Computer Science, Texas State University, San Marcos, TX 78666, USA; rakesh.suvvari@txstate.edu (R.S.); nuc4@txstate.edu (A.Y.); sana.alamgeer@txstate.edu (S.A.)

² Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville, Kingsville, TX 78363, USA; tarek.mahmud@tamuk.edu

* Correspondence: angu@txstate.edu

Abstract

Modern Internet of Things (IoT) systems increasingly rely on machine learning/deep learning models deployed across edge devices and cloud infrastructure. However, many real-world IoT systems require personalized models for each user or device, making scalable retraining, deployment, and real-time inference challenging. To address this, we propose a practical low-cost edge–cloud framework for concurrent personalization of AI models in IoT systems that can support users in a community-dwelling center. The personalization is achieved with a data-efficient machine learning retraining model (MODELXc) coupled with a layered software architecture that separates edge sensing, communication, cloud processing, and model management. First, retraining of AI model jobs runs as Kubernetes pods with pre-configured CPU and memory limits, and users are grouped into batches to prevent resource oversubscription during concurrent training. Second, edge devices communicate with the cloud using NATS.io request-reply messaging with binary-encoded payloads for low-latency communication. Third, a lifecycle pipeline handles versioned model storage, in-memory caching, and background synchronization. We evaluated this framework using two IoT applications: a wearable fall-detection system and a heart-rate-monitoring system. The second application is used to demonstrate the cross application portability of our infrastructure. Using the MODELXc personalization workflow, we executed 40 user-specific training jobs concurrently on a 40-logical-CPU server within one minute. We also demonstrated experimentally that a 40-CPU server can support personalization of 5000 virtual users in a 6–7 h period using a batch-wise processing strategy. Docker- and Kubernetes-based orchestration provided bounded per-job resource usage and resulted in a predictable system load compared with traditional background execution. For real-time inference, NATS.io with binary encoding and controlled throttling achieved a median latency of approximately 110–120 ms, compared with approximately 220 ms using the default without binary encoding.



Academic Editors: Ao Li, Geng Yuan and Na Gong

Received: 1 September 2026

Revised: 22 September 2026

Accepted: 24 September 2026

Published: 26 September 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: internet of things; edge-cloud computing; personalization; scalable AI systems; lightweight messaging; model life-cycle management; community healthcare

1. Introduction

As Internet of Things (IoT) devices become increasingly powerful, they are now capable of supporting continuous sensing and real-time prediction in applications such as wearable systems and industrial monitoring. These systems often use AI/machine learning

models to analyze sensor data streams and deliver timely predictions for safety-critical and context-aware applications [1]. However, many wearable systems need personalized AI models that are adapted to each user or device. For example, one global model usually cannot account for differences in human behavior, movement style, and operating environment. This is especially obvious in wearable fall detection, where model performance depends on how people move, their physical condition, and how the device is worn. Previous research shows that personalized models can lower false positives and make detection more reliable than global models [2,3].

However, implementing personalized IoT applications at scale in a community dwelling center brings major infrastructure challenges [4]. Each user needs an AI model that must be retrained using user-specific data periodically, and the privacy of users' data should be guaranteed. When more users join the system, retraining each model on a modest server can become computationally impossible and difficult to manage. Traditional methods that train models sequentially or as background processes on a server often waste resources and make system performance unpredictable because some resources remain overloaded while others are underused.

Many personalized IoT AI models also require real-time inference with strict response-time constraints [5]. When AI models are hosted on low-cost servers or in a private cloud, the system must maintain low-latency communication between edge devices and cloud services [6]. REST-based HTTPS communication is widely supported and easy to deploy [7]. However, in high-frequency IoT inference scenarios, REST/HTTP with text-based payloads can introduce per-request overhead from HTTP headers, connection handling, and JSON serialization/deserialization. Prior studies comparing messaging protocols and serialization formats show that protocol choice and payload encoding significantly affect latency and throughput in vehicle-to-cloud and IoT-style communication systems [8].

Managing personalized AI models in IoT applications also adds complexity because the system must support model versioning, persistent storage of each person's feedback data with privacy guardrails, in-memory caching, synchronization, and periodic re-deployment of updated models [9]. In our SmartFall lab [10], we conducted an extended user study with three older adults (age over 65) using a fall detection application over three weeks. At the end of each week, the collected activities of daily life data, especially misclassified events, had to be processed for personalization. This required multiple visits to each user's residence to retrain and deploy updated models, which was time-consuming and difficult to scale.

To support personalized AI model management and deployments with ease, this paper introduces a low-cost edge–cloud system for personalization of AI models in IoT systems with controlled usage of resources. The system automates the personalization pipeline, from collecting sensor data on edge devices to retraining, evaluating, deploying, and serving each user's model in real time. Using Docker containerization and Kubernetes orchestration, multiple personalized model retraining jobs are executed concurrently as isolated pods with defined CPU and memory limits. A batch-wise training strategy further groups users into parallel execution batches, enabling predictable resource utilization as the number of users grows.

However, orchestration alone cannot achieve high concurrency in a modest server because the number of parallel jobs still depends on the computational cost of each retraining job. If the AI models were retrained from scratch with feedback data for every user in isolation, each pod would need to load a large training dataset and update all model parameters. Therefore, even with efficient orchestration, only a limited number of users could be personalized concurrently on a modest server.

To reduce this cost, we use MODELXc [11], a two-stage test-time adaptation method for personalizing AI models within each retraining pod. MODELXc freezes a shared population-trained backbone and trains only two lightweight modules using labeled feedback data. The alignment module maps the user's embeddings toward the source feature distribution, while the residual adapter adjusts the backbone's output logits. This design reduces the CPU and memory requirements of each pod. It also allows the system to store one shared backbone for all users and only small adapter weights for each individual user.

Beyond efficient model retraining, our edge-cloud framework supports real-time, low-latency interaction between resource-constrained edge devices and the private cloud inference server. We evaluate HTTPS, Kafka-based messaging, and NATS.io [12] and show that NATS.io achieves the lowest median latency with stable performance. We further optimize NATS.io messages by using binary payload encoding and request throttling, which reduces transmission size and stabilizes the request rate on the edge device.

We validate the framework across two IoT applications: wearable fall detection for end-to-end personalization and deployment, and heart-rate monitoring for cross-application infrastructure reusability testing.

In summary, this paper presents a community-scale deployment framework for personalized AI models on a modest server or private cloud. The framework adopts a layered edge-cloud architecture and uses Kubernetes-based, batch-wise retraining with the MODELXc data efficient personalization strategy. It also incorporates optimized NATS.io communication and a model lifecycle management pipeline that supports versioned model storage, caching, and synchronization of per-user adapters. Our evaluation demonstrates that MODELXc personalization and container orchestration can work together to support personalization and process a population of 5000 virtual users through resource-bounded batch execution within a fixed window period of 6–7 h on a single modest server (PowerEdge R640/R740-class server) that costs around \$3000.

The main contributions of this paper are:

- A practical layered edge–cloud architecture for deploying and managing multiple personalized AI models on a modest server or private cloud. The framework integrates edge sensing, cloud-side personalization, real-time communication, and model management while supporting both on-device and server-side inference.
- An integration of the lightweight MODELXc personalization strategy with Docker- and Kubernetes-based execution, where a shared population-trained backbone remains fixed and only lightweight user-specific adaptation modules are trained. This reduces the computation, memory, storage, and model-transfer requirements associated with maintaining personalized models for many users.
- A resource-aware batch scheduling strategy that admits concurrent personalization jobs according to available CPU and memory capacity while reserving resources for other server services. The framework executed 40 user-specific personalization jobs concurrently on a 40-logical-CPU server and processed 5000 users through resource-bounded batches within approximately 6–7 h. The measured resource profile is further used to estimate practical server configurations and deployment costs for community-dwelling environments with several hundred users.
- A comparative evaluation of HTTPS, Kafka-based messaging, and NATS.io for real-time server-side inference, together with an optimized NATS.io communication pipeline using binary payload encoding and controlled request throttling. The optimized configuration reduces communication overhead and provides low and more stable request–reply latency for resource-constrained IoT devices.
- A model life-cycle management pipeline designed around a shared backbone and lightweight per-user adapters, supporting model versioning, persistent storage, in-

memory caching, background synchronization, and deployment of updated personalized models. The same core infrastructure is evaluated with wearable fall detection and heart-rate monitoring to demonstrate its applicability across different IoT sensing applications.

2. Related Work

2.1. Federated Learning for Personalized IoT Systems

Federated learning (FL) enables multiple clients to collaboratively train a model without sending raw data to a central server. Each client trains a model locally and sends model updates to a central coordinator, which aggregates them into a shared global model [13]. FL is useful for IoT systems that work well with a single global model and where data is distributed across devices and privacy constraints limit centralized data collection and training.

Personalized FL adapts the global model to individual clients; it can scale to many users since re-training happens locally on each client device. However, it requires edge devices having sufficient computing and infrastructure support to maintain independent personalized models for individual users, such as versioning, caching, and serving these models. Thus, while FL is suitable for privacy-sensitive decentralized learning, our framework targets IoT systems where initial centralized training is not a concern, and the main challenge is scalable personalized model training and management.

Besides the need for privacy, IoT systems introduce other challenges such as device heterogeneity, non-uniform data distribution, communication overhead, edge resource constraints, and intermittent connectivity. Prior work shows that differences in device capability, local data distributions, and model requirements make standard FL difficult to deploy in IoT systems [14]. Existing surveys also identify communication cost, resource limitations, and system heterogeneity as major barriers to practical personalized FL deployment in IoT [15].

2.2. Edge-Cloud Architectures for Scalable IoT Systems

Edge-cloud architectures distribute computation between resource-constrained edge devices and more powerful cloud servers [16]. This design is widely used in IoT systems because edge devices continuously collect data but often have limited computation, storage, and energy capacity. By offloading model training and complex inference to the cloud, these systems reduce the burden on edge devices while supporting real-time sensing and interaction.

Ngu et al. introduced an IoT edge computing framework that supports flexible interaction between edge devices and cloud services [17]. Such designs are useful for personalized fall detection because sensing can occur on wearable or mobile devices, while training and coordination can be handled in the cloud. However, this system only demonstrated the seamless edge-cloud interaction and does not address scalability challenges when the number of users increases, such as concurrent retraining, resource allocation, and synchronization between updated models and live inference services.

Large-scale IoT systems such as CrowdQuake demonstrated scalable infrastructure for continuous data streams and distributed communication with a large number of IoT applications [18]. Similarly, AI-as-a-Service frameworks support distributed model deployment and inference in edge-cloud environments [19]. Kubernetes-based edge frameworks such as KubeEdge extend cloud-native orchestration to edge environments [20]. However, these systems mainly focus on a shared global model and distributed inference, rather than on scalable training and management for personalized AI models for users.

2.3. Cloud-Centered Training and Orchestration

Cloud-native technologies support scalable and reproducible workload execution. Docker packages applications and dependencies into isolated environments, improving portability and reproducibility [21]. Kubernetes extends this capability through automated scheduling, resource allocation, scaling, and fault tolerance for distributed workloads [22]. Kubernetes-native machine learning platforms such as Kubeflow further support Machine Learning (ML) workflow orchestration, including training pipelines and model serving [23].

However, most cloud-native ML workflows are designed for training a single large model, running predefined pipelines, or serving shared models across many users. Personalized IoT systems require a different workload pattern: repeated training of many smaller user-specific models. Sequential retraining can cause long delays, while unmanaged concurrent retraining can lead to resource contention and unpredictable performance.

A practical and scalable personalization framework requires coordinated scheduling and model lifecycle management. The system must manage many independent retraining tasks while maintaining predictable resource usage and synchronizing updated models with real-time inference services.

2.4. Communication Frameworks for Real-Time IoT Inference

Communication between edge devices and cloud services is critical for real-time IoT inference. REST-based HTTPS is widely used because it is simple and compatible with many devices and services [7]. However, frequent request–response interactions can introduce overhead from request handling, HTTP headers, connection management, and verbose data serialization.

Apache Kafka supports high-throughput data streaming, backend analytics, and scalable event processing [24]. However, Kafka is better suited for backend stream processing than direct communication with resource-constrained edge devices. MQTT is a lightweight IoT protocol that reduces communication overhead through compact messages and publish–subscribe communication [25], but integrating MQTT with backend inference services requires additional routing or protocol conversion.

Lightweight messaging systems such as NATS.io support publish–subscribe and request–reply communication with low protocol overhead [26]. This makes them suitable for real-time IoT inference, where edge devices frequently send sensor data to cloud services and expect timely responses. In personalized IoT systems, the communication layer must also route requests to the correct user-specific inference service while supporting concurrent device interactions.

3. Methodology

This section presents the operational design of the proposed edge–cloud system for scalable personalization of IoT AI models in a community dwelling center with a modest server/private cloud budget. As shown in Figure 1, the system comprises four functional components: the edge layer, the communication layer, the cloud processing layer, which includes MODELXc’s personalization strategy; and the model management layer. Collectively, these components establish a coordinated pipeline that enables data sensing, communication, training, deployment, and serving of personalized AI models.

The system supports both on-device and server-side inference. For on-device inference, the personalized AI model is deployed directly on the edge device. For server-side inference, the AI model is hosted on the private cloud server to address the computational and memory limitations of edge devices. In both scenarios, retraining for personalization is conducted on the server using each user-specific feedback dataset. The operational details of each component are provided below.

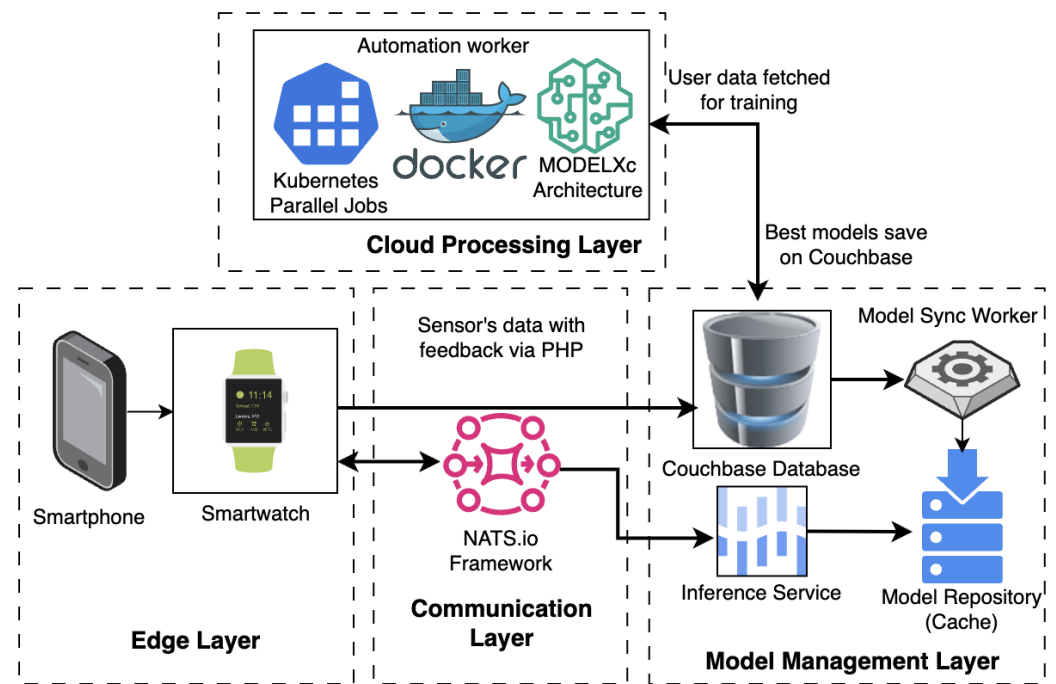


Figure 1. Generalized edge-cloud architecture for scalable personalization of IoT AI models. The system separates sensing, communication, training, and model management through four corresponding components.

3.1. Edge Layer and Model Inference

The edge layer comprises resource-constrained IoT devices, such as smartwatches, smartphones, or embedded sensors in other edge hardware, which continuously collect sensor data, including time-series signals from accelerometers, gyroscopes, or physiological signals. In the fall detection application, which we named SmartFall [10], a smartwatch and a smartphone are paired. The smartwatch continuously collects motion sensor data, while the smartphone is primarily used to establish a detailed profile for each new user during the activation phase.

For model inference, whether on-device or server-side, predictions begin once the number of sensed time-series data points reaches the configured window size of K data points. Every subsequent prediction requires only X additional data points, as the model slides X data points at a time, reusing all previous data points except the least recent X . Consequently, the overlap between consecutive windows is $K - X$. Instead of treating each window prediction as a final decision, the system calculates the average probability across Y consecutive model outputs and compares this value to a predefined threshold. This approach reduces false positives resulting from isolated positive predictions. For server-side inference, the Y windows are transmitted as a batch, prediction and aggregation are performed on the server, and only the final decision is communicated back to the edge device.

Continuous sensing and inference also introduce hardware-level constraints on resource-constrained edge devices, particularly battery consumption and thermal load. Recent studies on power-output optimization and battery thermal management highlight the importance of considering energy efficiency and thermal behavior alongside computational performance [27,28]. Although these studies target larger-scale energy systems, the same design principle motivates reducing unnecessary computation and communication on wearable devices. Accordingly, our framework batches sensor windows, minimizes payload size, controls request frequency, and supports server-side inference to reduce

the workload on the edge device; detailed battery and thermal measurements are left for future work.

3.2. Communication Layer

The communication layer enables efficient data exchange between edge devices and the cloud server. It is designed to minimize latency and protocol overhead while maintaining reliable message delivery for real-time IoT applications.

NATS.io serves as the underlying edge-to-server communication framework, selected due to its demonstrated lowest latency among evaluated options (see Section 4). NATS.io employs a lightweight messaging protocol supporting both publish-subscribe and request-reply communication patterns. In this protocol, edge devices function as producers, publishing messages to specific subjects/topics, while server-side components act as consumers subscribing to those subjects. NATS.io routes messages according to subjects, facilitating efficient communication between multiple edge devices and cloud services with minimal overhead.

3.2.1. Edge-Side Communication

For on-device inference, all data validation, and model inference is conducted on the edge device. The communication client on the edge device periodically transmits sensed data, user feedback, and inference results to the cloud server for archival using a REST-based API (as shown in Figure 2). Because this synchronization with the cloud server does not require real-time communication, data transmission occurs at fixed intervals to conserve edge-device battery life. Batching archival data enables the transmission of large volumes of information in a single request.

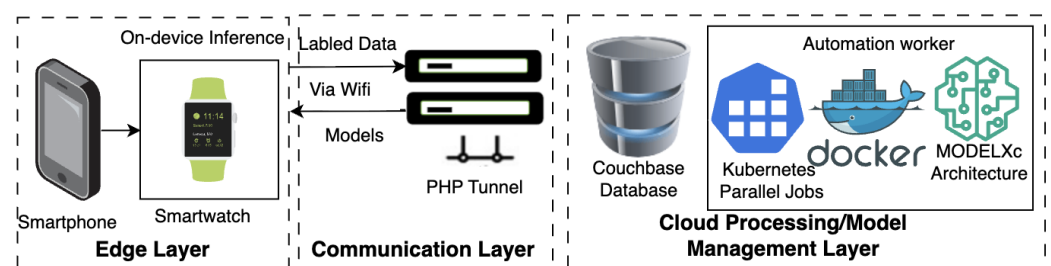


Figure 2. Edge-cloud architecture for on-device inference of IoT AI models. The Cloud handles scalable training and feedback data/model management.

For server-side inference, the communication client on the edge device performs basic preprocessing, such as packaging continuous sensor streams into fixed-size sliding windows, normalizing sensor values, and converting floating-point data into a compact binary representation. The data is encoded using a custom binary format that reduces the sensor-value precision from approximately nine decimal places to three decimal places and packs the values into integer representations. On the server, the integer values are decoded and dequantized back to floating-point values before model inference. A batch of Y windows is subsequently sent to the server for inference.

Transmitting batches of windows, rather than individual windows, reduces communication overhead by amortizing protocol costs such as message headers, network transmission, and request-reply handling, across multiple samples. This approach also decreases the number of network round-trip requests and prevents excessive request generation at high sampling rates, which could otherwise increase CPU, network, and battery usage on the edge device. Consequently, batching enhances bandwidth efficiency and results in more stable data transmission during continuous streaming.

3.2.2. Cloud-Side Communication

On the cloud/server side, during server-side inference, a consumer process receives incoming requests from edge clients. Each request is routed to the appropriate model inference service based on its subject, which includes a unique user ID, and the resulting prediction is returned to the corresponding edge device. This subject-based routing allows the server to manage requests from multiple users within a single consumer process.

Let R represent the incoming request rate and T_{infer} the inference time per request. For stable operation, the system must ensure that $R \cdot T_{infer}$ remains within the processing capacity of the consumer. The current implementation utilizes a single consumer; in our request-rate test, this consumer handled up to 100 inference requests per second without observed overload. For higher aggregate request rates, NATS.io supports horizontal scaling through multiple consumers organized in queue groups. In this configuration, multiple consumers can listen on the same subject, and messages are load-balanced among them. This capability enables the system to scale beyond the limitations of a single consumer as the number of edge devices increases. At the same time, the cloud communication layer integrates with the model inference pipeline to ensure that each request is processed using the latest personalized model.

3.3. Cloud Processing Layer

The cloud processing layer automates the retraining of personalized AI models using user-specific feedback data stored in the database. In the current implementation, a scheduler periodically evaluates the performance of each deployed personalized model using newly collected user feedback. When a configured model-quality criterion is violated, such as an increase in false positives, the scheduler automatically creates a retraining job for the corresponding user to be executed at the pre-configured 6–7 h period at night. Distribution-shift detection can additionally be configured as another retraining policy.

Each retraining task executes within a containerized environment using Docker [29], orchestrated by Kubernetes [30]. Docker encapsulates the training code, dependencies, and runtime configuration into a reusable image. Kubernetes instantiates this image into multiple pods for parallel execution. Each pod trains a user-specific model using a personalized dataset with predefined CPU and memory limits, preventing interference between concurrent jobs.

3.3.1. MODELXc-Based Personalization

Personalization inside each pod is performed using MODELXc [11], a lightweight two-stage adaptation method that keeps the population-trained backbone fixed and learns only user-specific adaptation modules. Figure 3 illustrates how MODELXc is integrated with the SmartFall App feedback loop and the cloud-side personalization workflow. The edge device processes raw sensor streams into overlapping windows, sends them for inference, and prompts for user feedback on the inference results. The system then stores user feedback in the cloud database. During personalization, the system retrieves the user's feedback samples from the cloud database, selects the specific user's feedback data, and trains the lightweight adaptation modules on top of the frozen global model.

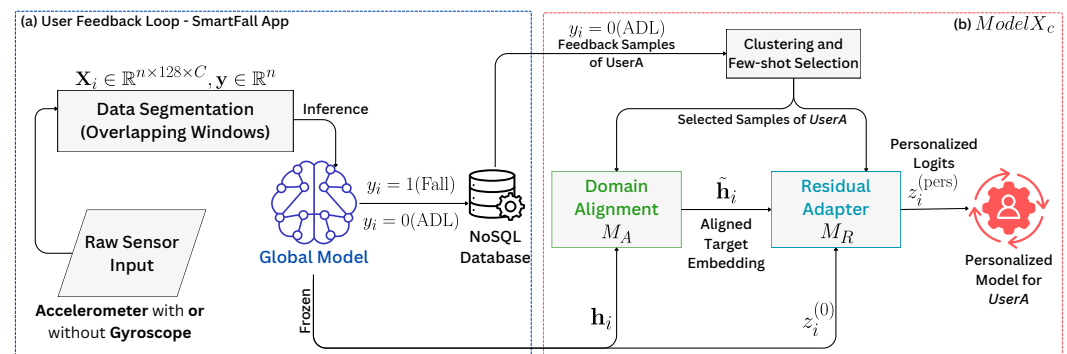


Figure 3. Overview of the MODELXc pipeline: (a) The SmartFall app collects user feedback ($y_i = 0$ (ADL), and $y_i = 1$ (Fall)). (b) Feedback samples are filtered via clustering and few-shot selection, then used to train a Domain Alignment module (M_A) and a Residual Adapter (M_R) on top of the frozen global model (G_θ), producing personalized logits for the target user.

Let the frozen population-trained backbone be $G_\theta(\cdot)$, where θ denotes the fixed backbone parameters. Given the i -th sensor window $\mathbf{X}_i \in \mathbb{R}^{K \times C}$, where K is the window length and C is the number of sensor channels, the backbone produces a latent embedding and an initial global-model logit:

$$\mathbf{h}_i = g_{\text{enc}}(\mathbf{X}_i), \quad z_i^{(0)} = g_{\text{clf}}(\mathbf{h}_i). \quad (1)$$

The backbone is trained once offline using pooled population data and remains fixed during personalization.

MODELXc introduces two user-specific modules: an alignment module $M_A(\cdot; \phi_A)$ and a residual adapter $M_R(\cdot; \phi_R)$, where ϕ_A and ϕ_R denote their trainable parameters. The alignment module maps the target user's embedding toward the source feature space:

$$\tilde{\mathbf{h}}_i = M_A(\mathbf{h}_i; \phi_A). \quad (2)$$

The residual adapter takes the aligned embedding and the original backbone logit as input and predicts an additive correction:

$$\Delta_i = M_R([\tilde{\mathbf{h}}_i; z_i^{(0)}]; \phi_R), \quad z_i^{(p)} = z_i^{(0)} + \Delta_i. \quad (3)$$

Here, $[\tilde{\mathbf{h}}_i; z_i^{(0)}]$ denotes the concatenation of the aligned target embedding vector $\tilde{\mathbf{h}}_i$ and the initial global model logit $z_i^{(0)}$, which together serve as the input to the residual adapter.

The final personalized prediction probability for the i -th window is computed as

$$p_i^{(p)} = \sigma(z_i^{(p)}). \quad (4)$$

The alignment module is optimized using a feature-matching loss over embeddings, while the residual adapter is trained using binary cross-entropy on the personalized logits from the user's labeled feedback windows. The detailed algorithmic design and predictive performance of MODELXc are reported in [11].

This formulation reduces the infrastructure cost of personalization. Instead of updating all backbone parameters θ , each pod trains only the lightweight user-specific parameters ϕ_A and ϕ_R . The raw population dataset and full backbone training pipeline are not required during personalization; each pod only needs the frozen backbone, the user's feedback data, and the compact source-feature reference used by the alignment loss. The per-user output artifact is the adapter-parameter pair (ϕ_A, ϕ_R) rather than a complete model. As a result, model storage, transfer, caching, and synchronization become significantly lighter.

Because the backbone is frozen, each pod performs a single forward pass over the user's feedback windows to obtain embedding-logit pairs $\{(\mathbf{h}_i, z_i^{(0)})\}_{i=1}^n$ and caches them. The remaining training operates on these cached vectors instead of repeatedly processing the original $K \times C$ sensor windows through the backbone. This reduces CPU time and memory usage and makes the resource profile of each personalization pod predictable.

Upon invocation, a pod maps its `JOB_COMPLETION_INDEX` to a specific user, retrieves the user's feedback data, constructs the personalization dataset, caches the backbone embeddings and logits, trains M_A and M_R , evaluates the personalized model on a held-out subset, and uploads the adapter weights and metadata if the new version improves over the deployed model.

3.3.2. Resource-Aware Batch Scheduling

Training jobs are executed in batches, as shown in Figure 4. Each MODELXc personalization task runs in an independent Kubernetes pod. Although MODELXc is lightweight, the server must reserve resources for non-training services, including the operating system, database, communication services, logging, and model synchronization. Therefore, the scheduler admits only the number of personalization pods that fit within the available CPU and memory budget.

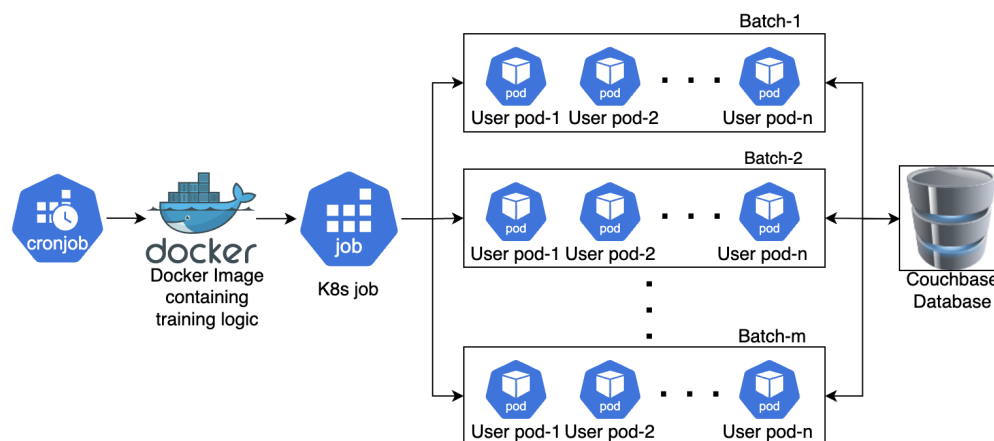


Figure 4. Batch-wise training of personalized models. Users requiring retraining are divided into batches. Each batch trains multiple user-specific models in parallel using Kubernetes pods. After one batch completes, the next batch is scheduled.

Let C and M denote the total CPU and memory capacity of the server. Let C_r and M_r denote the resources reserved for non-training operations, and let c_p and m_p denote the CPU and memory allocation of one personalization pod. The maximum number of concurrent jobs is:

$$P = \min\left(\left\lfloor \frac{C - C_r}{c_p} \right\rfloor, \left\lfloor \frac{M - M_r}{m_p} \right\rfloor\right). \quad (5)$$

For N users requiring personalization, the number of batches is:

$$B = \left\lceil \frac{N}{P} \right\rceil. \quad (6)$$

If T_b is the execution time of one batch and T_c is the optional cooldown interval between batches, the total processing time is:

$$T_{\text{total}} = BT_b + (B - 1)T_c. \quad (7)$$

Thus, MODELXc reduces the duration of each personalization job by training only lightweight adaptation modules, while Kubernetes controls the number of concurrent jobs through resource-aware scheduling. Together, these two mechanisms enable scalable personalization on a modest private-cloud server.

All users within one batch are personalized concurrently. At each pod, the best-performing trained models are stored in the database, synchronized with the inference service cache, and made available for real-time inference. After the batch completes, its resources are released and the next batch is admitted. A configurable cooldown interval can optionally be inserted between consecutive batches to allow transient server activity to return toward its normal operating level before another training wave begins.

3.4. Model Management Layer

The model management layer stores personalized models for each user in a central NoSQL database repository. This repository contains model artifacts and their metadata, including hyperparameters, parent-backbone version numbers, adapter version numbers, evaluation results, and deployment status. Maintaining multiple versions is essential because model performance may vary across retraining cycles due to changes in user data distribution or limited feedback samples. Versioning enables the system to compare models, track improvements, and revert to a previous model if a newly trained model under-performs.

Because the backbone is shared and frozen, the artifact stored per user is not a complete model but the pair of alignment and residual modules produced by MODELXc. The backbone is stored once in the repository, and repository growth across N users is $|\theta| + O(N \cdot (|\phi_A| + |\phi_R|))$ rather than $O(N \cdot |\theta|)$. Since adapter weights are several orders of magnitude smaller than a full model, retaining a longer version history per user is inexpensive, which is useful given that personalization quality can fluctuate across retraining cycles when feedback samples are limited.

Adapter artifacts are valid only with respect to the backbone they were trained against. Each adapter version therefore records the identifier and version of its parent backbone in its metadata, and the inference service refuses to load an adapter whose parent version does not match the resident backbone. When a new backbone is released, existing adapters are marked stale and their users are enqueued for re-personalization. Because a MODELXc retraining job is inexpensive, a fleet-wide re-personalization following a backbone update is itself schedulable as a small number of batches rather than as a full retraining campaign.

Although the proposed lifecycle mechanism maintains explicit dependencies between the shared backbone and per-user adapters, several failure modes can occur during operation. First, a personalization job may fail because of insufficient feedback data, malformed sensor windows, database access errors, or resource exhaustion inside a pod. In this case, the previous validated adapter remains active, and the failed job is logged and rescheduled for a later batch. Second, a newly trained adapter may perform worse than the currently deployed version on the held-out feedback subset. Such adapters are not promoted to production; instead, the system retains the previous best-performing adapter and records the failed candidate for audit. Third, a backbone update can make existing adapters incompatible with the active inference service. To avoid incorrect composition, each adapter stores its parent backbone version, and the inference service rejects adapters whose parent version does not match the resident backbone. The system marks these adapters as stale and queues their users for re-personalization. Finally, cache synchronization may be delayed or interrupted while inference requests are being served. In this case, the inference service continues using the last validated cached adapter until the background synchronization

worker completes a safe adapter swap. These recovery rules ensure that lifecycle failures degrade gracefully rather than interrupting real-time inference.

On-device inference requires compressing the model into TensorFlow Lite format, which may result in some loss of accuracy. The deployed artifact is the frozen backbone composed with the user's alignment and residual modules; these are fused into a single inference graph before conversion so that on-device execution remains one forward pass. Because the backbone is shared and changes only when a new global model is released, routine personalization updates require transferring only the adapter weights to the device rather than a complete model. However, when complex AI models exceed the memory or CPU capacity of edge devices, the model must be executed on the server. Consequently, the model management pipeline supports both on-device and server-side inference modes.

For on-device inference, the system retrieves and downloads the best-performing personalized model when the IoT application is activated on the edge device. A newer model is downloaded during subsequent activations only if an improved version is available. These models are stored locally on the device and reused for inference.

For server-side inference, AI models are cached in memory on the cloud server to enable fast inference. A synchronization process updates the cached models whenever improved versions become available. These updates happen in the background, ensuring that ongoing inference requests are not interrupted.

The structure of the personalized model also determines how the cache scales with the number of users. If each user had an independent full model, the cache would need to hold N complete models, growing as $O(N \cdot |\theta|)$ and bounding the number of users servable without eviction. Under MODELXc the frozen backbone is identical across users and is held in memory exactly once, so serving an additional user requires caching only that user's alignment and residual modules. Cache memory therefore grows as $M_{\text{cache}} \approx |\theta| + N \cdot (|\phi_A| + |\phi_R|)$ which is dominated by a constant term for realistic values of N . Inference for user u executes one forward pass through the shared backbone to obtain (h, z) , followed by that user's two small modules to produce the personalized logit z' . This also simplifies synchronization: an update pushed by the background sync worker is a kilobyte-scale adapter swap rather than the replacement of a full cached model, so updates complete quickly and are unlikely to interfere with concurrent inference requests.

Let T_{load} represent the time required to load a model from the database. Without caching, each inference request incurs this additional latency. In-memory caching eliminates this overhead, reducing per-request latency from $T_{\text{infer}} + T_{\text{load}}$ to approximately T_{infer} . Thus, caching models in memory avoids repeated database access and enables the inference service to use models that are immediately available for prediction. This approach reduces response time and improves throughput, particularly when many edge devices send requests simultaneously.

4. Results and Discussion

This section evaluates the proposed infrastructure with respect to concurrent MODELXc personalization, resource utilization, batch-wise training capacity, community-scale deployment, communication latency, and generalization to other personalized IoT applications. All experiments were performed on the university's internal server, a Dell system equipped with 40 logical CPUs (2 sockets, 10 physical cores per socket, and 2 hardware threads per core), 250 GB of RAM, and an NVIDIA GeForce GTX 1080 GPU.

For the personalization training experiments, each simulated user corresponds to an independent MODELXc personalization task. All compared execution configurations use the same MODELXc implementation, including the frozen backbone, adapter module archi-

textures, personalization data, and optimization settings so that the measured differences reflect the execution and resource-management strategy.

4.1. Personalization Training Infrastructure Evaluation

To evaluate the execution of concurrent personalization jobs, we compare traditional background-process execution with the Docker-containerized and Kubernetes-orchestrated configuration. Both configurations execute the same MODELXc personalization workload and differ only in how the training processes are launched and their resources are managed.

Figure 5 shows the resource utilization when 40 user-specific MODELXc personalization jobs are launched concurrently. The lightweight adaptation performed by MODELXc allows both configurations to remain within the computational capacity of the 40-logical-CPU server. However, the two execution strategies exhibit different resource-utilization patterns.

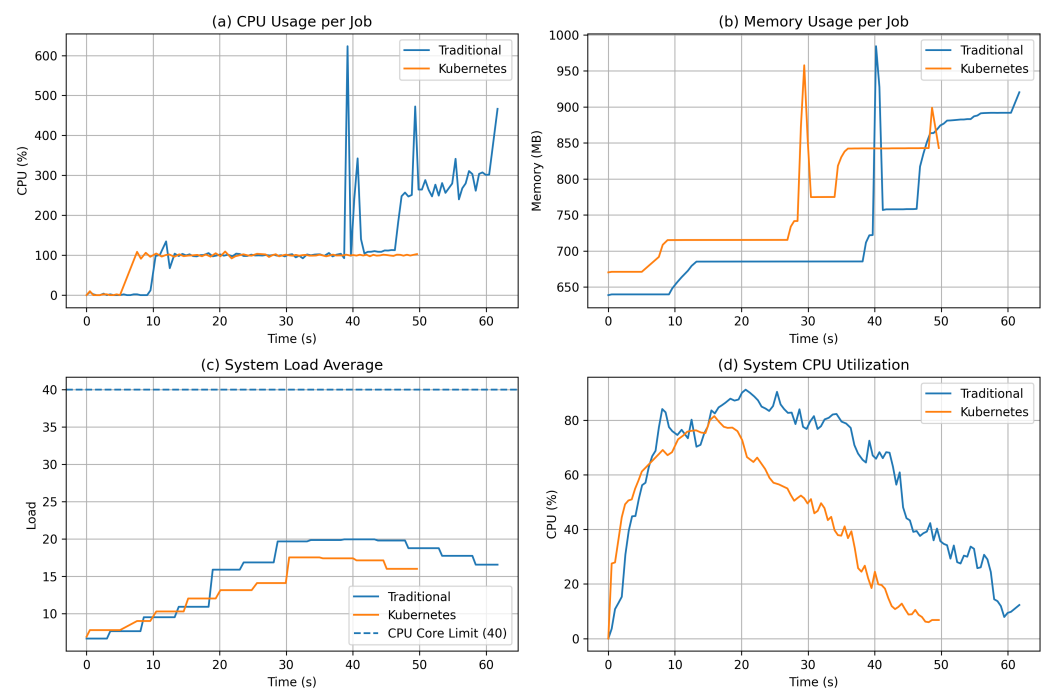


Figure 5. Resource utilization while executing 40 concurrent MODELXc personalization jobs on a server with 40 logical CPUs. Traditional background-process execution is compared with Docker/Kubernetes-isolated execution.

Under traditional background execution, individual processes are free to use additional CPU resources when they become available. During the experiment, the CPU utilization of a training process is therefore more variable, with short periods in which a process uses several logical CPUs. In contrast, Kubernetes applies the configured resource allocation policy to each personalization pod, keeping the per-job CPU utilization close to its assigned CPU-equivalent during most of the execution. As shown in Figure 5a, the nearly flat CPU-usage line indicates more uniform per-job CPU utilization across concurrent users under the Kubernetes configuration. Memory consumption is substantially lower than the available server capacity in both configurations. An individual MODELXc personalization job remains below approximately 1 GB of memory during the experiment, including short initialization peaks. Consequently, memory does not limit the number of concurrent jobs on the evaluated 250 GB server.

The system-level measurements show a similar trend. The one-minute load average remains below 20 for both configurations and therefore stays well below the available capacity of 40 logical CPUs. Traditional background execution reaches a load of approxi-

mately 20, whereas the Kubernetes configuration levels off at approximately 17–18. Total CPU utilization reaches approximately 90% during the background-process run and approximately 80% during the Kubernetes run, with the Kubernetes workload decreasing more gradually as individual personalization pods complete.

These results show that the current MODELXc workload can support a large degree of parallel personalization without oversubscribing the evaluated server. The main benefit of Docker and Kubernetes in this setting is therefore not a reduction in the computation performed by MODELXc, but controlled resource allocation, process isolation, and predictable execution when many independent personalization jobs are scheduled concurrently. These properties are particularly useful when personalization is executed repeatedly across batches of users while the server continues to support other application services.

4.2. Resource-Aware Batch Training Capacity

For the evaluated MODELXc configuration, the training experiment shows that each personalization job can be executed with approximately one CPU-equivalent while requiring less than approximately 1 GB of memory. This is an observed resource profile of the lightweight MODELXc workload and should not be interpreted as a general resource requirement for personalized AI training. In our experiments, each user follows the same MODELXc architecture and personalization procedure, with only the user-specific feedback data changing; therefore, the per-user workload remains similar across users. For practical deployment, however, we do not allocate the complete server capacity to personalization of AI models. One of the 40 available logical CPU-equivalents is reserved for operating-system activity, Kubernetes management, database access, logging, communication, model synchronization, and other server-side tasks. The remaining 39 CPU-equivalents are available for personalization.

Using the resource-admission rule described in Section 3, the production configuration therefore admits up to 39 MODELXc personalization jobs concurrently. Since the evaluated server contains 250 GB of RAM and the memory requirement of a personalization job remains below approximately 1 GB, CPU capacity is the limiting resource in this configuration.

A complete 39-user batch requires approximately one minute to execute, including concurrent personalization and the associated job-management overhead. In the current deployment, a conservative two-minute cooldown interval is inserted after a completed batch before the next training wave is launched. The resulting batch cycle is therefore approximately three minutes. The cooldown is an operational scheduling parameter rather than a requirement of MODELXc and can be adjusted according to the workload and available server capacity.

For a population of N users, the number of required batches is

$$B = \left\lceil \frac{N}{39} \right\rceil. \quad (8)$$

If T_b denotes the execution time of a batch and T_c denotes the cooldown between consecutive batches, the time required to process the population is approximately

$$T_{\text{total}} = BT_b + (B - 1)T_c. \quad (9)$$

In our experiments, we trained 5000 virtual users in batches; personalizing 5000 users requires

$$B = \left\lceil \frac{5000}{39} \right\rceil = 129 \quad (10)$$

batches. The first 128 batches contain 39 users each, corresponding to 4992 users, while the final batch contains the remaining eight users. With an approximately one-minute batch execution time and the conservative two-minute cooldown, the complete population can be processed in approximately 6–7 h.

The 5000-user result represents batch-wise processing rather than simultaneous execution of 5000 personalization jobs. At any instant, no more than the configured resource-bounded number of jobs is admitted. Larger user populations are handled through additional sequential batches while maintaining an upper bound on concurrent CPU and memory consumption. This allows the personalization workload to scale beyond the instantaneous capacity of a single server without resource oversubscription.

4.3. Community-Scale Deployment and Cost Analysis

The preceding experiment uses the full 40-logical-CPU research server to evaluate concurrent execution and large-population batch capacity. A community-dwelling deployment, however, may contain only tens or a few hundred residents and therefore does not require the same server capacity. The measured per-job resource profile can be used to estimate a smaller deployment configuration according to the expected resident population and the available personalization window.

For a small CPU server with four virtual CPUs, we conservatively reserve one CPU-equivalent for non-training services and allocate the remaining three CPU-equivalents to MODELXc personalization. This configuration therefore supports approximately three concurrent personalization jobs. An 8 GB memory configuration provides additional headroom for the operating system, Kubernetes components, communication services, model management, and other server processes in addition to the three active personalization jobs.

Table 1 shows the estimated personalization time for representative community sizes using the same one-minute training batch and two-minute cooldown policy evaluated above.

Table 1. Estimated MODELXc personalization capacity for community-scale deployments.

Users	Configuration	Parallel Jobs	Est. Time
100	4 vCPU/8 GB	3	~1.7 h
250	4 vCPU/8 GB	3	~4.2 h
400	4 vCPU/8 GB	3	~6.7 h
500	4 vCPU/8 GB	3	~8.3 h
500	8 vCPU/16 GB	7	~3.6 h

These estimates illustrate that the server configuration can be selected according to the size of the community and the permitted retraining window. For example, a 4-vCPU/8-GB deployment can process approximately 400 users within a 6–7 h overnight window under the conservative scheduling policy. The same configuration can process approximately 500 users if a longer 8–9 h window is available. Where a shorter personalization window is required, an 8-vCPU/16-GB configuration provides additional concurrency while retaining CPU capacity for other server operations.

The values in Table 1 are analytical estimates derived from the measured MODELXc resource utilization and batch duration. They should therefore be interpreted as deployment-sizing estimates rather than as measurements obtained directly on each listed virtual-machine configuration. The same resource limits can be imposed on the experimental server to validate these configurations before deployment.

To examine the economic feasibility of a community deployment, we also consider representative on-demand Linux virtual machines from Amazon Web Services (AWS),

Microsoft Azure, and Google Cloud Platform (GCP). The Table 2 comparison uses publicly listed on-demand prices and excludes persistent storage, taxes, and committed-use discounts [31–33].

Table 2. Representative cloud compute cost for a small community deployment. Prices are approximate U.S. dollars with on-demand Linux rates checked in August 2026.

Platform	Configuration	Always-On	Training-Only
AWS c7i.xlarge	4 vCPU/8 GiB	~\$130/mo	~\$35/mo
Azure F4s v2	4 vCPU/8 GiB	~\$123/mo	~\$33/mo
GCP e2-standard-4	4 vCPU/16 GiB	~\$98/mo	~\$26/mo

Network transfer costs are expected to remain small for the evaluated time-series workload. In the optimized accelerometer-only configuration, each 128-sample inference request is approximately 825 bytes, while the prediction response is typically below 50 bytes (See Section 4.5). Assuming, for illustration, one inference request per second for each of 500 continuously active users, the system would receive approximately 1.07 TB of sensor data per month but transmit only about 64.8 GB of prediction responses. Internet data ingress is free on the major cloud platforms, while AWS and Azure currently include the first 100 GB of monthly Internet egress at no charge; Google Cloud Standard Tier includes a comparable free egress allowance [34–36]. Therefore, under this workload, the additional network egress cost would be approximately zero. Even at two requests per second, the outbound traffic would be approximately 130 GB per month, resulting in only a few dollars of additional monthly egress cost on typical U.S. cloud pricing.

The always-on estimate assumes approximately 730 compute hours per month. The training-only estimate assumes that the instance is active for approximately 6.5 h per day over 30 days. The training-only configuration is applicable when inference is performed on the edge device and the cloud compute instance is primarily required for scheduled personalization. If server-side inference is required continuously, the always-on estimate is the more appropriate deployment model.

The cost analysis indicates that community-scale personalization does not require a large dedicated training server. A modest CPU-based cloud instance can support scheduled personalization for several hundred users, while the deployment can be scaled to a larger instance when a shorter personalization window or additional server-side workloads are required.

4.4. Communication Framework Evaluation

In this section, we evaluate the communication performance required for real-time server-side inference. We measure the end-to-end latency of different communication approaches, including HTTPS, Kafka-based systems, and NATS.io lightweight messaging. Here, inference latency refers to the end-to-end request–reply time measured from when the edge device sends a prepared batch of Y sensor windows until it receives the final prediction from the server. The server performs inference on the Y windows in parallel, averages their output probabilities as described in Section 3.1, and returns the resulting decision. Therefore, the reported latency includes communication and server-side inference and aggregation, but excludes the time required to collect the sensor windows before transmission. Sensor-window collection time is determined by the sensing configuration, such as the sampling rate and window size, and is common across the communication methods evaluated in this study.

For evaluation, each round-trip exchange consists of a batch of sensor data windows sent with an inference request, followed by the server’s prediction response. The request

payload contains sensor readings encoded in JSON. On average, the JSON request is about 5.5–6.5 KB, whereas the response is very small, usually under 50 bytes, and contains only the predicted label or score. This configuration mirrors a typical real-time IoT inference workload in which edge devices and the cloud consumer continuously exchange relatively small messages.

Figure 6 presents the latency comparison among HTTPS, Kafka-based methods, and NATS.io. HTTPS serves as the baseline due to its widespread support and ease of deployment. NATS.io demonstrates a median latency of approximately 220 ms, which is lower than the median latencies of HTTPS and Kafka-based methods, both of which range from approximately 240 to 280 ms.

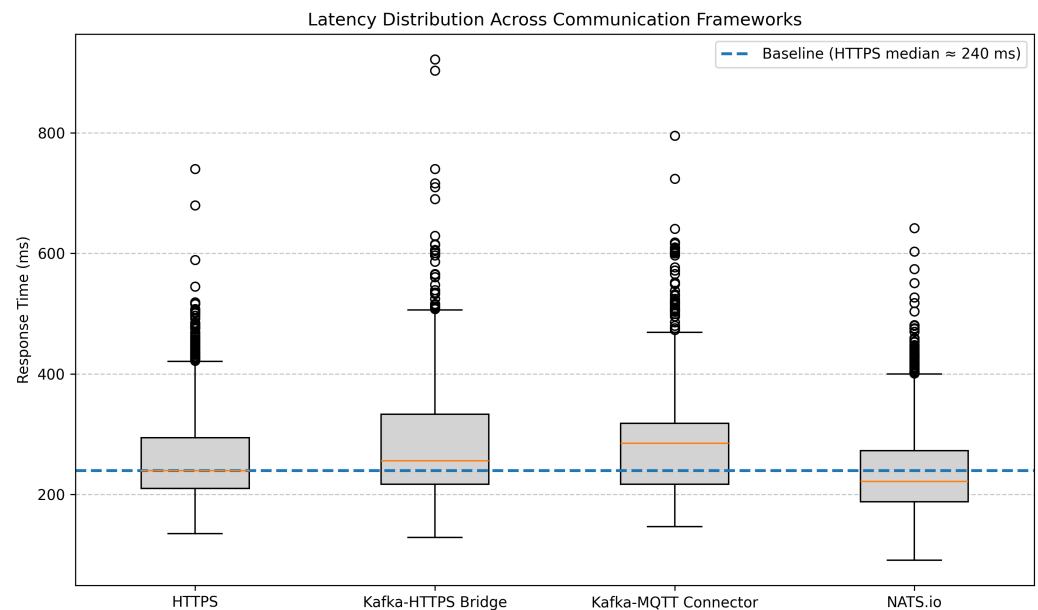


Figure 6. Latency distribution across communication frameworks (HTTPS, Kafka-based approaches, and NATS.io). NATS.io shows a lower median and a tighter spread compared to other approaches.

NATS.io also exhibits a narrower latency distribution with fewer high-latency occurrences. In contrast, HTTPS and Kafka-based methods show broader distributions, indicating less consistent response times.

Kafka is known for its high throughput, fault tolerance, and support for stream processing. However, in an edge-cloud setup, Kafka does not natively support direct communication with the resource-constrained Android edge device and therefore requires an HTTP bridge or MQTT connector. These additional components introduce protocol translation and routing overhead. HTTPS provides a simpler communication path, but it still incurs overhead from connection management, HTTP headers, and JSON serialization and deserialization. The single-consumer inference test is separate from the 5000 virtual user experiment in Section 4.2, which evaluates batch-wise personalization training and does not represent 5000 simultaneous inference clients.

Overall, NATS.io provides lower latency and more stable request–reply performance in our evaluated setup, making it a suitable communication mechanism for real-time IoT inference.

4.5. Optimization of Lightweight Messaging

To further enhance real-time communication between edge devices and the cloud, we minimize transmitted data size via binary encoding and introduce controlled throttling.

Figure 7 illustrates the impact of various optimization strategies on the latency characteristics of NATS.io communication. In the default configuration, the median end-to-end latency is approximately 220 ms and exhibits substantial dispersion, including multiple high-latency outliers. This default latency remains relatively higher for scenarios requiring frequent real-time communication.

To address this, we use a custom binary encoding that converts floating-point sensor values into smaller integers and sends data as an ordered byte array, called SFN1 (SmartFallNATS-Version1). In the fall detection system, each JSON request includes meta-data, 128 timestamps, and 128 samples for every accelerometer axis. This leads to a payload of about 5.5–6.5 KB when sending only accelerometer data. By comparison, the SFN1 binary format represents the same 128-sample accelerometer window with compact integer values, shrinking the payload to roughly 825 bytes. We also verified that using the dequantized three-decimal sensor values did not change the model’s accuracy or precision compared with the original higher-precision floating-point inputs. When both accelerometer and gyroscope signals are transmitted, the SFN1 payload is about 1593 bytes. This smaller payload cuts transmission overhead and eliminates the need for JSON parsing on both the edge device and the server. Overall, SFN1 achieves about a $7\times$ reduction in payload size for accelerometer-only data compared to the JSON format.

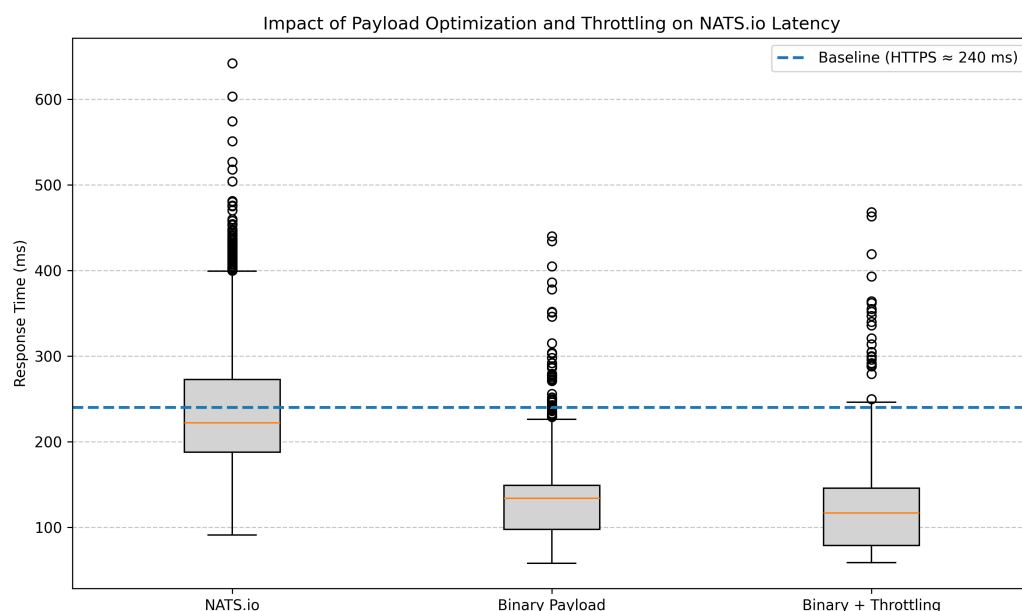


Figure 7. Impact of payload optimization and throttling on NATS.io latency. Binary encoding and controlled throttling reduce median latency and improve consistency.

With binary encoding, the median latency drops to about 130 to 140 ms, and the latency distribution becomes more consistent. Controlled throttling was also introduced to regulate the frequency of requests sent by the edge device. In the absence of throttling, frequent requests increase CPU, network, and watch battery usage, potentially causing unstable transmission due to internal scheduling. When both binary encoding and throttling are applied, the median latency decreases to 110–120 ms, and most responses remain below the HTTPS baseline.

These results indicate that reducing payload size and controlling the request rate improve both latency and response-time stability.

4.6. MODELXc Personalization Effectiveness

To evaluate the effectiveness of MODELXc as the personalization workload in the proposed pipeline, we summarize the participant-level personalization results reported in [11]. In this comparison, the baseline refers to the frozen global model evaluated on each target participant without user-specific adaptation. MODELXc, in contrast, keeps the shared population-trained backbone frozen and updates only the lightweight alignment and residual adaptation modules.

Figure 8 shows the participant-level F1-scores for ten participants. MODELXc consistently achieves higher F1-scores than the non-adapted global baseline across all participants. The baseline model achieves F1-scores in the range of approximately 0.67 to 0.76, while MODELXc achieves F1-scores in the range of approximately 0.87 to 0.92. On average, MODELXc improves the participant-level F1-score from approximately 0.728 to 0.898.

This result demonstrates that MODELXc is suitable for the proposed scalable personalization infrastructure. Instead of retraining a complete model for every user, MODELXc produces user-specific adaptation modules while reusing the same frozen global backbone. Therefore, the system can reduce per-user training cost and model storage overhead while still improving personalized fall-detection performance.

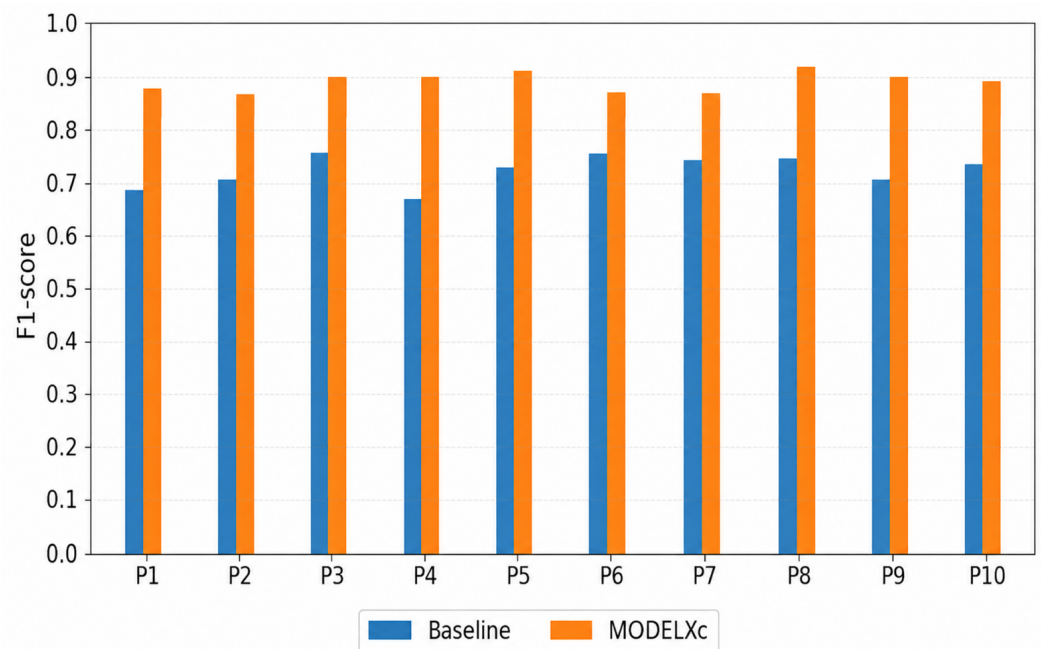


Figure 8. Participant-level F1-score comparison between the personalized model trained from scratch and MODELXc.

4.7. Cross Application Infrastructure Reusability Test

To evaluate the reusability of this framework beyond fall detection, we prototyped a personalized heart rate monitoring App. In this App, the smartwatch collects heart rate readings from the smartwatch sensor of a specific user and transmits them to the cloud via NATS.io. for inference. For each request, the edge device sends a fixed-size heart rate window of K_{hr} samples, represented as a one-dimensional time-series vector: $H = [h_1, h_2, \dots, h_{K_{hr}}]$ where each h_i is a heart rate value measured in beats per minute (BPM). Compared to fall detection, which uses multi-axis motion data, the heart rate monitoring input is lower-dimensional because it uses a single physiological signal.

The cloud's heart-rate inference service checks the incoming data window against the user's historical heart rate patterns and returns either a normal or high-variability response. The internal logic for detecting heart rate variability is outside the scope of this paper

and irrelevant for our evaluation. Figure 9 indicates a median latency of approximately 100 ms, with most responses ranging from 90 ms to 130 ms. The heart-rate latency is lower compared to the fall detection setup. This is expected because the payloads are smaller and the processing is simpler. There are also fewer extreme outliers, which shows that communication performance is stable and consistent. These results demonstrate that the edge–cloud communication design supports additional IoT use cases without any change while maintaining low and predictable latency.

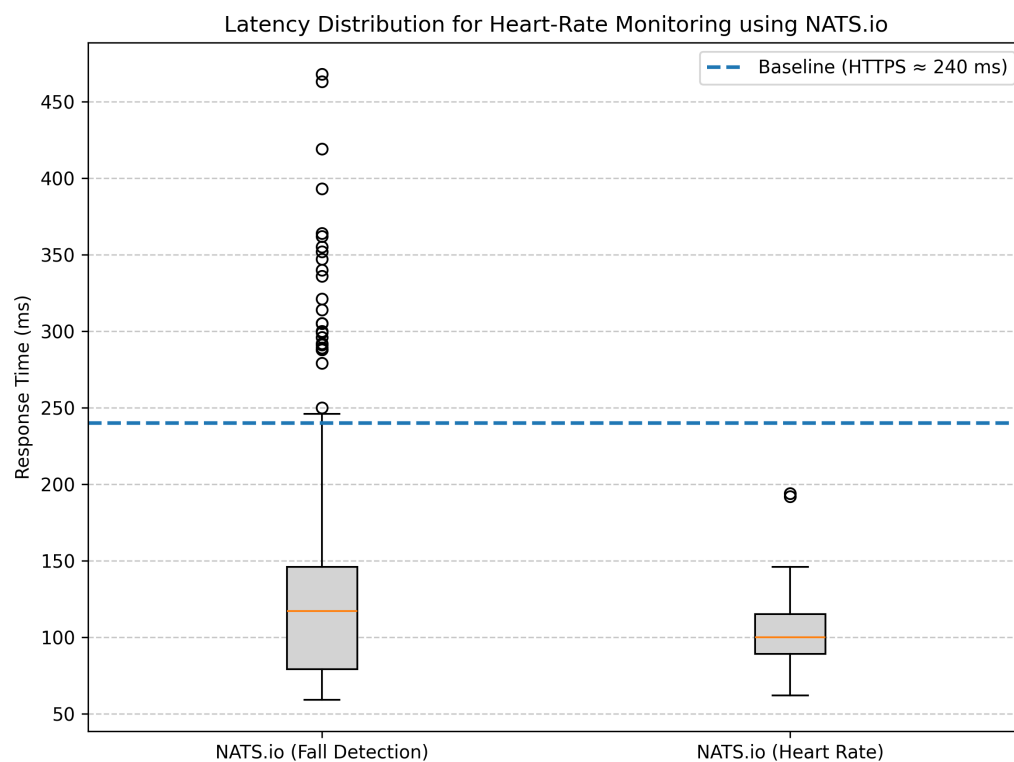


Figure 9. Latency distribution for heart rate monitoring using NATS.io. The system achieves low median latency with a tight distribution, demonstrating stable performance.

This also demonstrates that its core infrastructure is designed to be application-independent. The training pipeline, batch-wise scheduling, NATS.io communication optimization, model storage, caching, and synchronization processes remain consistent across applications. Only the application-specific components require modification. These components include sensor modalities, preprocessing logic, AI model architecture, and feedback labels for personalization. For instance, a gait analysis system may utilize both accelerometer and gyroscope data with a dual-stream AI model, whereas heart rate monitoring relies on lower-dimensional physiological signals. In both scenarios, the Kubernetes-based retraining, NATS.io communication, and model lifecycle pipeline remain reusable.

We did not perform the personalization process for the heart rate monitoring application since a new heart-rate AI model must first be trained offline with the heart rate population data. The fall detection application already demonstrates the scalability of the personalization process. Furthermore, our earlier work with the ModelXc personalization process [11] has already shown that ModelXc is AI model-agnostic.

When fall detection and heart rate monitoring inference services are deployed on the same server, their AI model training jobs are scheduled as independent Kubernetes pods and share CPU and memory resources. Consequently, the effective batch size must be adjusted according to the combined resource demand. If the applications are deployed on separate servers, each application can utilize its own available resources and train additional

users in parallel. These findings demonstrate that the proposed training architecture can support multiple IoT applications while maintaining the same core infrastructure.

5. Conclusions

Deploying personalized AI models for IoT applications on a modest server or private cloud requires efficient user-specific personalization, controlled concurrent execution, low-latency communication, and reliable model life-cycle management. This paper presents a practical edge–cloud framework that integrates MODELXc-based personalization with Docker and Kubernetes orchestration, lightweight NATS.io communication, and versioned model storage, caching, and synchronization. The novelty of our work is the design of a resource-aware infrastructure for systematically training, deploying, and managing large numbers of personalized models under resource constraints, together with an empirical evaluation of its scalability.

MODELXc reduces the computational cost of personalization by keeping the population-trained backbone frozen and training only lightweight user-specific adaptation modules. In our evaluation, 40 user-specific personalization jobs were executed concurrently on a server with 40 logical CPUs without CPU oversubscription, while each job required less than approximately 1 GB of memory. Compared with traditional background-process execution, Docker- and Kubernetes-based orchestration provided bounded per-job resource usage and a more predictable system resource profile. For practical operation, one CPU-equivalent was reserved for non-training services and the remaining capacity was used to execute up to 39 personalization jobs concurrently. Using this resource-aware batch configuration, 5000 users were personalized in 129 batches within approximately 6–7 h. The same measured resource profile was also used to estimate smaller community-scale deployments, showing that a modest CPU-based server can support personalization for several hundred residents depending on the available retraining window.

For server-side inference, NATS.io provided lower and more stable request–reply latency than the evaluated HTTPS and Kafka-based approaches. Further optimization using binary payload encoding and controlled throttling reduced the median fall-detection inference latency from approximately 220 ms with the default NATS.io configuration to approximately 110–120 ms. The framework also maintains a shared backbone with lightweight per-user adapters in the model repository and inference cache, reducing the storage and synchronization requirements as the number of personalized users increases.

Finally, the communication and infrastructure design was evaluated with both wearable fall detection and heart-rate monitoring. The heart-rate monitoring prototype achieved a median communication latency of approximately 100 ms, demonstrating that the same edge–cloud infrastructure can support an additional IoT sensing application with low and predictable communication latency. While application-specific components such as the sensor modality, population-trained model, preprocessing, and personalization logic may differ, the core infrastructure for containerized retraining, resource-aware batching, communication, model storage, and synchronization remains reusable. Overall, the results demonstrate the feasibility of deploying and managing personalized IoT AI models at community scale using modest server or private-cloud resources.

Author Contributions: Conceptualization, R.S. and A.Y.; methodology, R.S., A.Y. and A.H.H.N.; data collection: A.Y., R.S. and S.A.; software, R.S.; validation, R.S., A.Y., T.M. and A.H.H.N.; writing—original draft preparation, R.S., A.Y. and A.H.H.N.; writing—review and editing, A.H.H.N., S.A. and T.M. All authors have read and agreed to the published version of the manuscript.

Funding: The work is funded by the National Science Foundation under the NSF-SCH (21223749).

Institutional Review Board Statement: The study protocols were approved by Texas State University’s Institutional Review Board (IRB), with protocol number 9461.

Informed Consent Statement: Informed consent was obtained from all subjects involved in this study.

Data Availability Statement: The source code of the framework is publicly available at https://github.com/tkst-cs-smartfall/Edge_Cloud_Framework_IoT/, accessed on 10 September 2026.

Acknowledgments: We thank the National Science Foundation for funding the research under the NSF-SCH (21223749).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Singh, R.; Gill, S.S. Edge AI: A Survey. *Internet Things Cyber-Phys. Syst.* **2023**, *3*, 71–92. [CrossRef]
2. Vallabh, P.; Malekian, N.; Malekian, R.; Li, T.M. Personalized fall detection monitoring system based on learning from the user movements. *arXiv* **2020**, arXiv:cs.LG/2012.11195. [CrossRef]
3. Mauldin, T.R.; Canby, M.E.; Metsis, V.; Ngu, A.H.H.; Rivera, C.C. SmartFall: A Smartwatch-Based Fall Detection System Using Deep Learning. *Sensors* **2018**, *18*, 3363. [CrossRef] [PubMed]
4. Yasmin, A.; Mahmud, T.; Debnath, M.; Ngu, A.H.H. An Empirical Study on AI-Powered Edge Computing Architectures for Real-Time IoT Applications. In Proceedings of the 2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC), Osaka, Japan, 2–4 July 2024; pp. 1422–1431. [CrossRef]
5. Bian, J.; Al Arafat, A.; Xiong, H.; Li, J.; Li, L.; Chen, H.; Wang, J.; Dou, D.; Guo, Z. Machine Learning in Real-Time Internet of Things (IoT) Systems: A Survey. *IEEE Internet Things J.* **2022**, *9*, 8364–8386. [CrossRef]
6. Kreković, D.; Krivić, P.; Podnar Žarko, I.; Kušek, M.; Le-Phuoc, D. Reducing communication overhead in the IoT–edge–cloud continuum: A survey on protocols and data reduction strategies. *Internet Things* **2025**, *31*, 101553. [CrossRef]
7. Fielding, R.T. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Thesis, University of California, Irvine, CA, USA, 2000. Available online: <https://dl.acm.org/doi/10.5555/932295> (accessed on 10 September 2026).
8. Proos, D.P.; Carlsson, N. Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV. In Proceedings of the 2020 IFIP Networking Conference (Networking), Paris, France, 22–26 June 2020; pp. 10–18.
9. Yasmin, A.; Mahmud, T.; Haque, S.T.; Alamgeer, S.; Ngu, A.H.H. Enhancing Real-World Fall Detection Using Commodity Devices: A Systematic Study. *Sensors* **2025**, *25*, 5249. [CrossRef] [PubMed]
10. SmartFall Project. SmartFall: Smartwatch-Based Fall Detection. 2023. Available online: <https://smartfall.github.io> (accessed on 5 January 2026).
11. Yasmin, A.; Mahmud, T.; Alamgeer, S.; Islam, T.; Ngu, A.H.H. MODELX_c: A Novel Data-Efficient Personalization for Fall Detection. In Proceedings of the 2026 IEEE International Conference on Digital Health (ICDH), Sydney, Australia, 13–18 July 2026; pp. 261–271. [CrossRef]
12. Sharvari, T.; Sowmya Nag, K. A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming. *arXiv* **2019**, arXiv:1912.03715. [CrossRef]
13. McMahan, B.; Moore, E.; Ramage, D.; Hampson, S. Communication-Efficient Learning of Deep Networks from Decentralized Data. *arXiv* **2016**, arXiv:1602.05629. [CrossRef]
14. Wu, Q.; He, K.; Chen, X. Personalized Federated Learning for Intelligent IoT Applications: A Cloud-Edge Based Framework. *IEEE Open J. Comput. Soc.* **2020**, *1*, 35–44. [CrossRef] [PubMed]
15. Zhang, T.; Gao, L.; He, C.; Zhang, M.; Krishnamachari, B.; Avestimehr, S. Federated Learning for Internet of Things: Applications, Challenges, and Opportunities. *arXiv* **2022**, arXiv:cs.LG/2111.07494.
16. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
17. Ngu, A.H.H.; Eyitayo, J.S.; Yang, G.; Campbell, C.; Sheng, Q.Z.; Ni, J. An IoT Edge Computing Framework Using Cordova Accessor Host. *IEEE Internet Things J.* **2022**, *9*, 671–683. [CrossRef]
18. Huang, X.; Lee, J.; Kwon, Y.W.; Lee, C.H. CrowdQuake: A Networked System of Low-Cost Sensors for Earthquake Detection via Deep Learning. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD ’20, Virtual Event, 6–10 July 2020; pp. 3261–3271. [CrossRef]
19. Andrés Cajas Ordóñez, S.; Samanta, J.; Suárez-Cetrulo, A.L.; Ghosh, R.; Giannopoulos, A.E.; Barceló, A.; Carbajo, R.S. An Open-Source AI-as-a-Service Framework for Federated, Efficient, and Drift-Robust Learning in the Continuum Edge–Cloud. *IEEE Access* **2026**, *14*, 26265–26277. [CrossRef]

20. Saini, L.M.; Ajmani, P.; Asha, V.; Pithamber, K.; Sreevani, N. KubeEdge for Scalable IoT Applications. In Proceedings of the 2024 7th International Conference on Contemporary Computing and Informatics (IC3I), Greater Noida, India, 18–20 September 2024; Volume 7, pp. 1–7. [\[CrossRef\]](#)
21. Pahl, C. Containerization and the PaaS Cloud. *IEEE Cloud Comput.* **2015**, *2*, 24–31. [\[CrossRef\]](#)
22. The Kubernetes Authors. Kubernetes Documentation. 2025. Available online: <https://kubernetes.io/docs/home/> (accessed on 13 October 2025).
23. Bisong, E. Kubeflow and Kubeflow Pipelines. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*; Apress: Berkeley, CA, USA, 2019; pp. 671–685. [\[CrossRef\]](#)
24. Kreps, J. Kafka: A Distributed Messaging System for Log Processing. In Proceedings of the NetDB Workshop, Athens, Greece, 12 June 2011.
25. Thangavel, D.; Ma, X.; Valera, A.; Tan, H.X.; Tan, C.K.Y. Performance evaluation of MQTT and CoAP via a common middleware. In Proceedings of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 21–24 April 2014; pp. 1–6. [\[CrossRef\]](#)
26. NATS.io. NATS: High-Performance Messaging for Cloud Native Systems. 2023. Available online: <https://docs.nats.io/> (accessed on 12 January 2026).
27. Qi, W.; Yu, L.; Zhao, S.; Guo, Y.; Ge, S.; Zhang, Y.; Cheng, J.; He, Z. Mechanism of PEMFC Output Performance Improvement via Non-Uniform Bluff Body Structures in Bipolar Plate Flow Channels: Flow Field Regulation and Mass Transfer Enhancement. *Renew. Energy* **2026**, *271*, 126020. [\[CrossRef\]](#)
28. Qi, W.; Wang, X.; Yu, J.; Liu, Y.; Lan, P.; Cheng, Z.; Ge, S.; He, Z.; Jin, C.; Zhang, Y. Influence of Radial Divergent Coolant Flow Plate Channel Distribution on Thermal Management of Power Batteries. *Int. Commun. Heat Mass Transf.* **2026**, *180*, 112450. [\[CrossRef\]](#)
29. Bernstein, D. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [\[CrossRef\]](#)
30. Zhong, Z.; Xu, M.; Rodriguez, M.A.; Xu, C.; Buyya, R. Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions. *ACM Comput. Surv.* **2022**, *54*, 217. [\[CrossRef\]](#)
31. Amazon Web Services. Amazon EC2 On-Demand Pricing. Available online: <https://aws.amazon.com/ec2/pricing/on-demand/> (accessed on 17 August 2026).
32. Microsoft Azure. Linux Virtual Machines Pricing. Available online: <https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/> (accessed on 17 August 2026).
33. Google Cloud. Compute Engine General Purpose VM Pricing. Available online: <https://cloud.google.com/products/compute/pricing/general-purpose> (accessed on 17 August 2026).
34. Amazon Web Services. AWS Global Network FAQs: Data Transfer Pricing. Available online: <https://aws.amazon.com/about-aws/global-infrastructure/global-network/faqs/> (accessed on 16 September 2026).
35. Microsoft Azure. Bandwidth Pricing. Available online: <https://azure.microsoft.com/en-us/pricing/details/bandwidth/> (accessed on 16 September 2026).
36. Google Cloud. Virtual Private Cloud Pricing. Available online: <https://cloud.google.com/vpc/pricing> (accessed on 16 September 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.