# Performance Analysis of Architectural Patterns for Federated Learning Systems

Ivan Compagnucci
Gran Sasso Science Institute
L'Aquila, Italy
ivan.compagnucci@gssi.it

Riccardo Pinciroli
Zimmer Biomet
Milan, Italy
riccardo.pinciroli@gssi.it

Catia Trubiani
Gran Sasso Science Institute
L'Aquila, Italy
catia.trubiani@gssi.it

***Abstract*—Designing Federated Learning systems is not trivial, as it requires managing heterogeneous and distributed clients' resources, while balancing data privacy and system efficiency. Architectural patterns have been recently specified in the literature to showcase reusable solutions to common problems within Federated Learning systems. However, patterns often lead to both benefits and drawbacks, e.g., introducing a message compressor algorithm may reduce the system communication time, but it may produce additional computational costs for clients' devices. The goal of this paper is to quantitatively investigate the performance impact of applying a selected set of architectural patterns when designing Federated Learning systems, thus providing evidence of their pros and cons. We develop an open source environment by extending the well-established Flower framework; it integrates the implementation of four architectural patterns and evaluates their performance characteristics. Experimental results assess that architectural patterns indeed bring performance gains and pains, as raised by the practitioners in the literature. Our framework can support software architects in making informed design choices when designing Federated Learning systems.**

***Index Terms*—Architectural Patterns, Federated Learning, Performance Evaluation**

## I. Introduction

Federated Learning (FL) systems are increasingly attracting attention from both researchers and practitioners [1]–[3], mainly due to their peculiar characteristic of distributing the learning to multiple clients, thus reducing data privacy risks and computational costs encountered by traditional centralized methods [4], [5]. According to Gartner's 2023 Hype Cycle for Emerging Technologies [6], Federated Learning (FL) has the potential to revolutionize machine learning applications by enabling collaborative model training across distributed data sources without the need to share sensitive data. Recent studies on FL [7], [8] lists *performance optimization* as one of the main challenges in this domain. Computational overhead may be introduced for different reasons: (i) preventing private data from being leaked during model transmission; (ii) guaranteeing high communication efficiency; (iii) tolerating heterogeneous hardware resources and handling fault tolerance in case of offline devices; (iv) verifying the accuracy of local models. All these scenarios motivate the need to instrument FL systems and enable their performance evaluation [9], thus understanding when performance issues may arise.

In the software architecture community, designing FL systems is recognized as an emergent topic, and a reference architecture, namely *FLRA*, has been recently defined in [2]. A follow-up work from the same authors is presented in [10] where a collection of architectural patterns is derived from a systematic literature review, and the design challenges of FL systems are presented. Notably, applying architectural patterns triggers both benefits and drawbacks for the system performance. For instance, the *Client Selector* pattern [10] consists of selecting clients with more power and network bandwidth, thus reducing the chances of clients dropping out and lowering the communication latency. This entails the introduction of a *Client Registry* pattern [10] that is in charge of storing information about the client devices, but the maintenance of this information requires extra communication and storage costs, which is further affected by the number of client devices [10]. Inspired by these challenges, our research focuses on developing a framework that enables the performance analysis of FL architectural patterns, thus supporting software architects in making informed decisions.

To the best of our knowledge, the most closely related state-of-the-art approaches to architecting FL systems are [9], [11]. Di Martino et al. [11] investigate the quantitative impact of architectural patterns applied to the design of Cloud Edge architectures within the healthcare domain. Lai et al. [9] propose a benchmark to emulate FL environments while offering application programming interfaces (APIs) to implement, deploy, and evaluate FL algorithms. As opposite, this paper aims to quantitatively analyze the performance characteristics of some architectural patterns extracted from [10]. Specifically, we develop a novel framework, namely *Architectural Patterns for Federated Learning* AP4Fed, that builds upon the well-consolidated Flower [12], and hosts the implementation of these four selected architectural patterns: (i) the *Client Registry* that stores information on clients contributing to the training; (ii) the *Client Selector* that improves the system efficiency by selecting clients that show some preferred criteria, e.g., more computational power; (iii) the *Client Cluster* that enhances the training efficiency by grouping clients based on their similarities; (iv) the *Message Compressor* that increases communication efficiency by reducing the message data size.

In summary, the main contributions of this paper are: (i) the development of an FL framework hosting the implementation of four architectural patterns indicated relevant for the system performance in a recent literature review [10]; (ii) the design of experiments whose results support software architects in quantitatively understanding the impact of these architectural patterns on the performance of FL systems; (iii) the flexibility in changing the design settings for patterns, thus enabling further quantitative investigations. The developed framework and replication data are publicly available [13].

## II. Background

*Federated Learning.* The exponential growth of computing devices, now equipped with advanced sensors such as cameras, microphones, and GPS, has resulted in the generation of high amounts of data, much of which are often sensitive and highly personal [10], [14]. This vast amount of data has driven the development of machine learning applications, as it represents a valuable resource for training prediction models capturing complex patterns, and improving decision-making [7]. However, while this data holds significant potential for machine learning applications, its sensitive nature raises serious privacy concerns, especially when shared with centralized servers or external organizations [1], [15]. FL is a paradigm introduced by Google in 2016 [14], in which multiple client devices (e.g., mobile phones and laptops) collaborate under the coordination of a central server to train a global machine learning model, while keeping their local data private [1], [7], [10]. It addresses data privacy by enabling machine learning model training directly on clients' devices, ensuring that personal data is locally stored. It also allows computation to be distributed across a network, improving scalability and reducing reliance on centralized processing resources [15].

Figure 1 provides an overview of the FL mechanism. A FL process starts when a central server broadcasts a set of initial global model parameters (i.e., model weights and structure) to all participating clients ①. After receiving the parameters, model training is performed locally across the client devices ②. Then, each participating client device sends its updated model parameters (i.e., trained model weights) to the central server ③. The central server collects all trained models and aggregates them to generate an updated "version" of the global model ④. New parameters of the global model are then distributed to the client devices for the next FL round. This establishes an iterative process for continuously updating and improving the global model until it converges.

Developing FL systems is far from trivial, as it requires robust coordination among system parties (i.e., clients and the central server), overcoming challenges on device heterogeneity, and scalability across networks [14], [15]. To effectively manage this complexity, Flower [12] has emerged as a leading framework for supporting and simplifying the deployment of FL systems. Flower offers a standardized implementation of essential FL components, along with high-level abstractions that enable researchers and practitioners to explore and implement new functionalities [12]. For all these reasons, we make use of Flower as the reference framework to implement and evaluate the performance of FL architectural patterns.

*Architectural Patterns.* The selection of patterns is motivated by our focus on investigating performance-related characteristics of FL systems. We consider all those patterns that are relevant for system efficiency [10]. We exclude the patterns that contribute to other aspects of FL systems, e.g., the so-called *Model Aggregation patterns* include design solutions of model aggregation used for different purposes. The detailed description of selected patterns is provided in [10], in the following, we briefly outline their main characteristics:

i. The *Client Registry* manages and provides information about all clients participating in the training process;
ii. The *Client Selector* samples client devices during the training according to some predefined criteria to increase the *system efficiency*;
iii. The *Client Cluster* groups client devices to improve the *training efficiency*;
iv. The *Message Compressor* reduces the message data size to increase the *communication efficiency*.

## III. Methodology

Our methodology consists of developing a novel FL framework, namely AP4Fed [13], that hosts the implementation of selected patterns and enables their evaluation. We leverage Flower [12], and architectural patterns are implemented by extending its codebase Python library. It is worth remarking that software architects can use AP4Fed by following our detailed instructions (please refer to [13]), i.e., modifying input parameter values and conducting further investigations.

### *A.* AP4Fed *Infrastructure*

The overview of the developed environment is depicted in Figure 2. This setup uses Docker-Compose to create a configuration with one container representing the central server

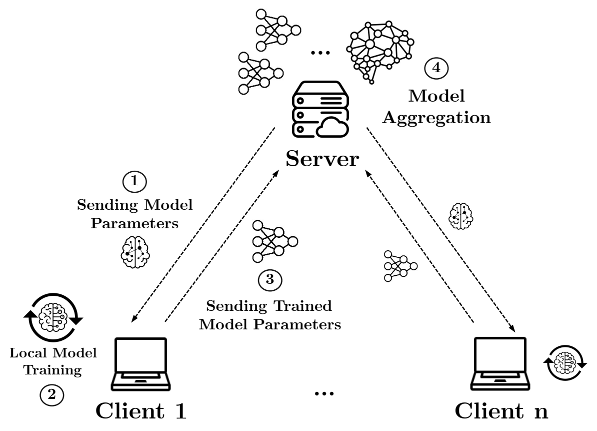


Fig. 1: Federated Learning overview, inspired by [10].

| Parameter | Description |
|---|---|
| `NUM_ROUNDS` | no. of Federated Learning Rounds |
| `nS` | no. of Server Container Instances |
| `nC` | no. of Client Container Instances |
| `n_CPU` | no. of CPUs allocated to each Container |
| `RAM` | memory capacity allocated to each Container |

TABLE I: System parameters summary.

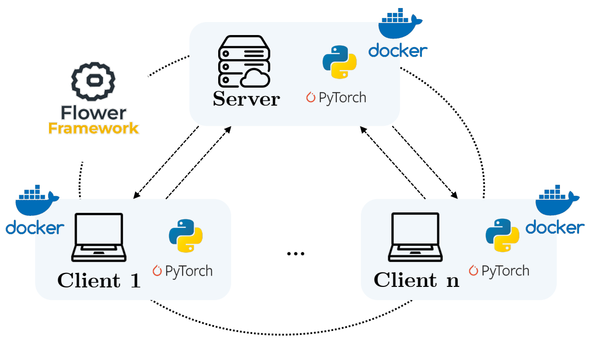


Fig. 2: AP4FED Infrastructure overview.

and multiple containers acting as client devices. This environment simulates a real-world FL scenario, where isolated Docker containers represent a network of clients performing local training, and a central server container handles model aggregation. Our framework extends the Flower Python library version 1.12.0 [12]. Flower provides a set of methods to enable the communication between the server and all clients, handling key aspects such as client-server message exchange, model updates, and round-based coordination. We use PyTorch version 2.5.0 [16] for model training, allowing each client to train on local data. Table I collects the input parameters to set up the environment, i.e., the number of FL rounds (`NUM_ROUNDS`), instances of server and client containers (`nS` and `nC`, respectively), and the allocation of resources per container, specifically the number of physical CPUs (`n_CPU`) and the `RAM` capacity. Note that, these parameters may intentionally vary depending on the experimental setup, which is determined by the architectural pattern under investigation. For instance, the Client Selector pattern selects client devices depending on their available resources, hence we decide to set up clients with a different number of allocated CPU(s).

## B. Client Registry Architectural Pattern

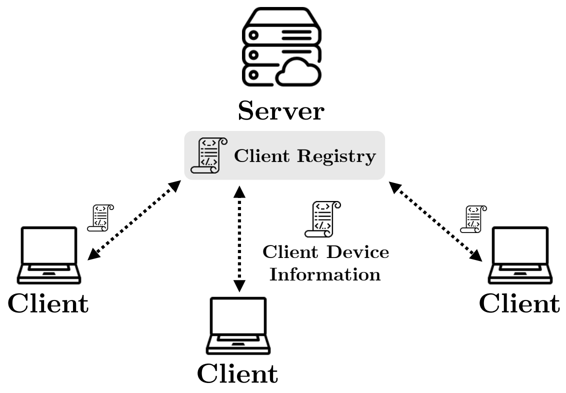


Fig. 3: Client Registry pattern overview.

*Context.* The Client Registry pattern is designed to store relevant information of each client device participating in an FL process [10]. As illustrated in Figure 3, the central server maintains a centralized data registry for storing client device data. At the beginning of each round, the server sends a request to each client, that responds with a set of attributes including device ID, connection uptime and downtime, and resource availability (e.g., computational power, communication capacity, and storage). This architectural pattern plays a key role in ensuring efficient client management, enabling the server to monitor all connected devices and gather useful insights, such as identifying clients that are actively participating, have disconnected, or exhibit untrustworthy behavior [10].

| Parameter | Description |
|---|---|
| `cID` | Client Unique Identifier |
| `n_CPU` | no. of Container CPU |
| `cluster_Type` | Cluster associated to the Client |
| `training_time` | Client Training Time |
| `communication_time` | Client Communication Time |
| `total_round_time` | Client Total Round Time |

TABLE II: Extended parameters for Client Registry.

*Our Implementation.* This architectural pattern is already partly implemented in Flower through the `ClientManager`[1] abstract base class. This class enables the Client Registry to track each client's information and interactions with the server. However, relevant parameters regarding the client performance (i.e., number of CPUs) or the device system metrics (i.e., training or communication time) are missing. Such attributes provide key information that can be leveraged to exploit new features, such as dynamic client selection based on resource availability or prioritizing clients with higher stability for critical rounds. For this reason, we extend the standard `ClientManager` class to implement additional parameters. Table II lists the added parameters. We consider a unique client identifier (`cID`), that is generated when each client is instantiated. This identifier helps to distinguish clients within the system. The `n_CPU` parameter, which indicates the number of CPUs assigned to each container. The `cluster_Type` parameter is a label reassigned at each round to group clients based on a predefined client clustering strategy (see Section III-D). Timing parameters are collected using `psutil`, i.e., a Python library for retrieving system metrics. The `training_time` parameter records the time each client spends on the local model training. The `communication_time` measures the time required to exchange model parameters with the server. The `total_round_time` parameter represents the overall time each client takes to complete a round, combining both training and communication durations.

## C. Client Selector Architectural Pattern

*Context.* The Client Selector pattern introduces a mechanism for selecting a subset of client devices to participate in the FL round according to specific criteria [10], [17]. Such criteria can be categorized as follows: (i) *resource-based* factors, which evaluate the computational and network capabilities of devices; (ii) *data-based* factors, which assess the quality,

[1]Additional information is available at https://flower.ai/docs/framework/ref-api/flwr.server.ClientManager.html

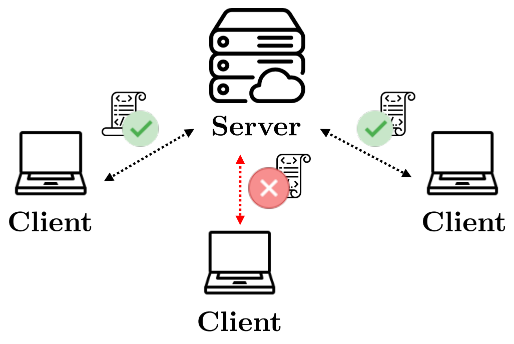


Fig. 4: Client Selector pattern overview.

heterogeneity, and volume of the data each client holds; and (iii) *performance-based* factors, which assess each client's contributions to the global model enhancement based on their performance in the latest round [17]. Overall, this assessment involves evaluating resources, data quality, and performance metrics to determine if and which clients are best suited for contributing to the global model aggregation. As depicted in Figure 4, the server evaluates each client by analyzing the information stored in the Client Registry, to exclude or include clients based on selection criteria. Evaluating and selecting clients based on their characteristics may mitigate challenges associated with limited computational resources, unbalanced data distribution, and unstable client connectivity [10], [17].

*Our Implementation.* To implement the Client Selector pattern, we apply a filtering procedure to the Flower `configure_fit`[2] method, thus assessing the eligibility of client devices for the FL round. The `configure_fit` method sets the essential information (i.e., participating clients, model parameters) required to conduct the FL round. At the beginning of each round, the server accesses the Client Registry to review clients' information. Our current implementation considers the *resource-based* selection strategy [10], in which the server selects clients on the basis of their computational power, e.g., the number of allocated CPUs. This process ensures that only clients meeting specific criteria are chosen for each round, thus optimizing the FL process.

## D. Client Cluster Architectural Pattern

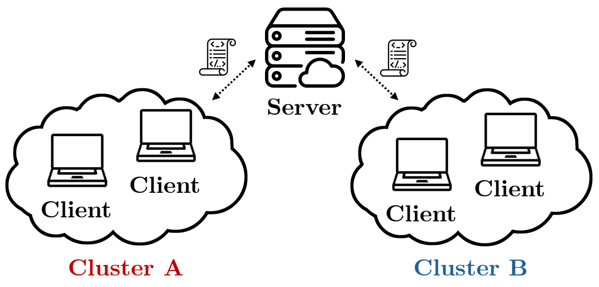


Fig. 5: Client Cluster pattern overview.

[2]Additional information is available at https://flower.ai/docs/framework/how-to-implement-strategies.html

*Context.* The Client Cluster pattern groups client devices based on the similarity of their characteristics. They can be categorized as follows: (i) *computational resources*, referring to the device's processing power and memory capacity; (ii) *network capabilities*, i.e., the speed and stability of the device's network connection; and (iii) *data partitioning*, considering how data is distributed across devices [10]. The graphical representation of the Client Cluster architectural pattern is reported in Figure 5. The server uses the information from the *Client Registry* to group clients into distinct clusters based on their characteristics. By clustering clients with similar features, the server can apply tailored aggregation and training strategies. This is performed to enhance training efficiency by reducing training round time and increasing the model accuracy [10]. For instance, the server can address the data heterogeneity problem [18] by clustering clients based on data distribution, leading to improved model performance and faster convergence [7], [10].

*Our Implementation.* Similarly to the Client Selector pattern, we implement the Client Cluster mechanism through the `configure_fit` method. Clients are assigned to a cluster depending on their local data partitions, which may vary between independent and identically distributed (IID) and non-independent and identically distributed (non-IID) forms [18]. In IID data settings, each client has a balanced set of training samples representing all classes and ensuring that data are uniformly distributed. In contrast, non-IID data present an unbalanced distribution, meaning that some clients have more training examples of a specific class than others [7], [18].

## E. Message Compressor Architectural Pattern

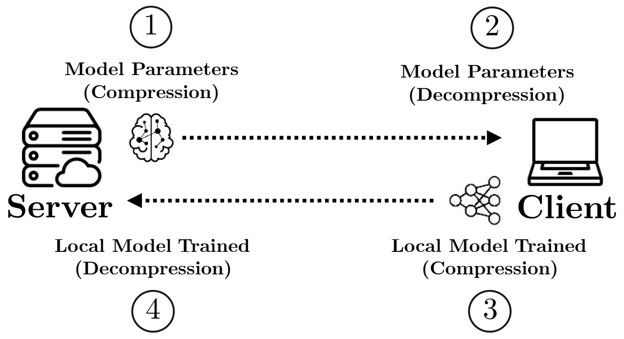

1
Model Parameters (Compression)
2
Model Parameters (Decompression)
Server
Client
Local Model Trained (Decompression)
4
Local Model Trained (Compression)
3

Fig. 6: Message Compressor pattern overview.

*Context.* The Message Compressor pattern introduces a procedure to reduce the size of message data exchanged (e.g., model weight and structure) between the central server and client devices in an FL environment [10], [19]. As illustrated in Figure 6, this pattern operates on both ends of the system. Initially, the server compresses the model parameters ① before sending them to client devices. The data is transmitted in a compressed format, hence reducing communication overhead. Upon receiving compressed parameters, client devices decompress the data ② and prepare it for local training. After completing local training, client devices compress their updated model parameters ③ before transmission. Then, the server decompresses received messages ④ to aggregate all models. In FL, multiple rounds of model exchange occur between the server and client devices to collaboratively train the global model [15]. By minimizing the volume of data transmitted, the Message Compressor significantly decreases communication time and preserves bandwidth, which could be particularly beneficial for bandwidth-constrained client devices [10]. However, the time needed for compression and decompression can overcome the saving in communication time, especially when the exchanged data are low.

*Our Implementation.* We implement the Message Compressor pattern by introducing a compression mechanism to optimize data exchanges between clients and the server. Specifically, our approach embeds the *zlib* library [20] to implement two-stage compression and decompression (server-to-client and client-to-server), effectively reducing the overhead and improving the communication efficiency. *zlib* belongs to the *LZ77* family of compression algorithms and is valued for its high-speed compression with minimal resource consumption, making it a good choice among available algorithms [21].

## IV. Experiments

*Subject Systems.* We train a Convolutional Neural Network (CNN) as a global model on the CIFAR-10 dataset [22], which consists of $60,000$ $32x32$ color images ($50,000$ images for training and $10,000$ for testing) divided into 10 distinct classes. Each class includes images representing objects and animals (i.e., trucks, dogs), challenging the CNN to effectively recognize and classify images into one of the 10 classes. The main characteristics of the global model are reported in Table III. The CNN consists of two convolutional layers (Conv) with ReLU activations, followed by max-pooling layers (Pool) to reduce spatial dimensions. Specifically, the first convolutional layer (Conv1) has 6 filters with a $5x5$ kernel, and the second convolutional layer (Conv2) has 16 filters with a $5x5$ kernel. Each convolutional layer is followed by ReLU activation. The model then transitions to two fully connected layers (FC), where FC1 has 120 units, FC2 has 84 units, and the final layer (FC3) has 10 units corresponding to the classes of CIFAR-10. The model is trained with a batch size of 32, a learning rate of $0.001$, and a Stochastic Gradient Descent (SGD) with a momentum equal to $0.9$.

*Hardware Setup.* Experiments are conducted on a commodity machine with an Apple M3 Pro chip featuring an `11-core CPU @4.02GHz` and `36GB` memory. Docker Compose enables the manual allocation of resources to each container (e.g., CPU core and RAM) allowing flexible experiment configurations tailored to the architectural pattern being evaluated. For instance, different CPU allocations are assigned to client containers to evaluate system performance when implementing the *Client Selector* pattern. This approach provides a scenario with computational disparity between clients to analyze how different resource allocations impact the system performance. It is worth remarking that the maximum number of concurrently running containers is limited by the host machine's capacity (i.e., 11 core) to prevent CPU overcommitment. This is crucial, as exceeding devices' processing limits can lead to resource contention, potentially invalidating the replicability of experimental results [23]. All experiments are repeated 10 times and reported results are the average of all outputs. The 99% confidence interval is represented by shaded areas in all graphs.

| Parameter | Value |
|---|---|
| `Dataset` | CIFAR-10 |
| `Training Samples` | $50,000$ |
| `Test Samples` | $10,000$ |
| `Model Type` | Convolutional Neural Network |
| `Model Structure` | *Conv1*: 6 filters, $5x5$ kernel, ReLU activation |
| | *Pool*: Max pooling, $2x2$ kernel |
| | *Conv2*: 16 filters, $5x5$ kernel, ReLU activation |
| | *FC1*: 120 units, ReLU activation |
| | *FC2*: 84 units, ReLU activation |
| | *FC3*: 10 units (for CIFAR-10's 10 classes) |
| `Batch Size` | 32 |
| `Learning Rate` | 0.001 |
| `Optimizer` | SGD (momentum = 0.9) |

***Conv:** Convolutional Layer; **Pool:** Pooling Layer; **FC:** Fully Connected Layer.*

TABLE III: Global model parameters.

| Parameter | Description |
|---|---|
| `Training Time` | Time Spent on Local Training |
| `Communication Time` | Time for Client-Server Communication |
| `Total Round Time` | Total Time for each FL Round |
| `F1 Score` | F1 score of the global model |

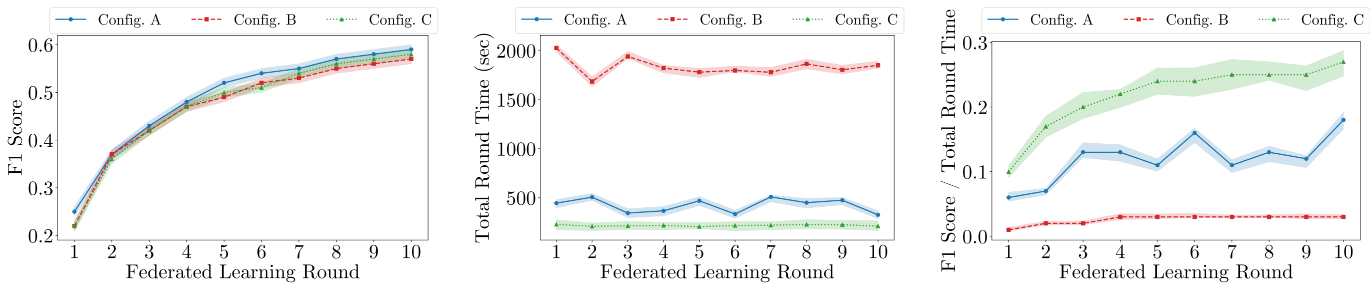
TABLE IV: Evaluation metrics used in the experiments.

*Evaluation Metrics.* Table IV lists metrics used in our experiments, as proposed in [24]. We evaluate key performance indicators, including training, communication, and total round times, as well as model accuracy using the F1 score [25]. These metrics provide round-by-round performance tracking, offering quantifiable insights into the trade-offs associated with the design solutions of architectural patterns.

### A. Performance Analysis: Client Selector

Table V presents input parameters used for the Client Selector pattern experiment. We consider 10 training rounds, a single server, and 4 clients. To test the effectiveness of this pattern, clients are divided into *High-Spec* and *Low-Spec* categories based on their computational capacity. High-Spec clients are equipped with 2 CPUs, providing larger processing power, while Low-Spec clients have 1 CPU only, reflecting more limited computational capacity. Since we implement a *resource-based* selection strategy, we establish specific evaluation criteria requiring clients to have a physical CPU allocation greater than one to participate in each FL round.

As reported in Table VI, we evaluate the performance of the Client Selector pattern through three different experiment configurations labeled $A$, $B$, and $C$. In each configuration, we keep an number of 4 Clients and 1 Server. Config. $A$ includes 4 High-Spec clients without applying the Client Selector pattern. In Config. $B$, we introduce a Low-Spec client and 3 High-Spec ones, and we do not make use of the selection strategy.



(a) F1 Score per FL round (b) Total Time per FL round (c) F1 Score over Total Round Time

Fig. 7: Performance analysis of the Client Selector pattern.

TABLE V: Input parameters for Client Selector experiments.

| **Parameter** | **Value** |
|---|---|
| `NUM_ROUNDS` | 10 |
| `nS` | 1 |
| `nC` | 4 |
| `n_CPU` | 1 *for Low-Spec Clients*; 2 *for High-Spec Clients* |
| `RAM` | 2GB |
| `Selection Strategy` | Resource-based |
| `Selection Criteria` | no. of CPU >1 |

TABLE VI: Experiment configurations for Client Selector.

| | **Config. *A*** | **Config. *B*** | **Config. *C*** |
|---|---|---|---|
| `Client Selector` | ✗ | ✗ | ✓ |
| *no. of High-Spec Clients* | 4 | 3 | 3 |
| *no. of Low-Spec Clients* | - | 1 | - |
| `Total Clients` | 4 | 4 | $4 \rightarrow 3$ |

✗: *Without Client Selector pattern;* ✓: *With Client Selector pattern.*

Config. $C$ follows the same setup as Config. $B$; however, we introduce the Client Selector to exclude the Low-Spec client, limiting the learning process to 3 High-Spec clients.

Figure 7 depicts the performance analysis of an FL system implementing the Client Selector pattern. Starting from the global model accuracy, Figure 7a depicts the F1 Score evolution across FL rounds for each experiment configuration. The results of all experiments indicate that, after 10 rounds, the accuracy ranges between 0.57 and 0.59, with stable model convergence. Figure 7b shows the total time required to complete an FL round for each configuration. Results indicate that Config. $A$ and Config. $C$ exhibit similar round time, with Config. $C$ being slightly faster. This behavior is expected as Config. $C$, compared to Config. $A$ and Config. $B$, excludes one client from the FL round, thus reducing the workload for model training and aggregation. Configuration $B$ shows a high FL total round time, approximately $9\times$ longer than Config. $C$, due to the bottleneck introduced by the Low-Spec client. We aggregate accuracy and total round time to derive an efficiency metric, represented by the ratio of F1 Score and Total Round Time across rounds. Values assumed by this ratio are shown in Figure 7c, where we can notice that the best trade-off between accuracy and efficiency is achieved by Config. $C$ that is the one implementing the pattern. Config. $A$ achieves moderate efficiency with some fluctuation across rounds. As expected, Config. $B$ maintains the lowest efficiency ratio, reflecting the overhead introduced by the Low-Spec client in FL rounds.

*Architectural Implications.* Software architects can employ this architectural pattern when there is a disparity among clients in terms of resource capabilities. Experimental results show that the system response time, expressed as the total round time in FL contexts [24], deteriorates when Low-Spec clients participate in FL rounds, thus leading to bottlenecks and high waiting time.

## B. Performance Analysis: Client Cluster

Table VII reports input parameters used in the Client Cluster experiment. The setup involves 10 rounds of FL with 1 server and 8 clients with 1 CPU and 2GB of RAM. We implement the Client Cluster architectural pattern by adopting a *data distribution-based* strategy [10], thus creating two distinct clusters: Cluster $A$ and Cluster $B$ based on how client data is partitioned. Clients in Cluster $A$ show an equal distribution of data samples across classes, representing IID data, while clients in Cluster $B$ contain non-IID data. To partition the non-IID data for clients in Cluster $B$, we use the Dirichlet distribution [26][3], effectively replicating variations in data distribution. This approach enables a realistic and accurate test of the Client Cluster pattern, effectively simulating typical real-world scenarios where non-IID datasets are common [15]. The Dirichlet Distribution is a multivariate probability distribution, commonly used in FL experiments [18], [27], allowing for non-IID data's partitions. For completeness, we report the Dirichlet distribution formula:

$$p(x_1, x_2, \ldots, x_k) = \frac{1}{B(\alpha)} \prod_{i=1}^{k} x_i^{\alpha_i - 1}$$

$p(x_1, x_2, \ldots, x_k)$ represents the probability density function of the Dirichlet distribution for $k$ variables. The constant $B(\alpha)$, known as the beta function, normalizes the distribution, calculated based on the parameters $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ to ensure the total probability sums to one. The term $\prod_{i=1}^{k} x_i^{\alpha_i - 1}$ represents the product of each variable $x_i$ raised to the power $\alpha_i - 1$, capturing the non-uniformity in probability distribution

[3]https://flower.ai/docs/datasets/ref-api/flwr_datasets.partitioner.DirichletPartitioner.html

TABLE VII: Input parameters for Client Cluster experiments.

| Parameter | Value |
|---|---|
| `NUM_ROUNDS` | 10 |
| `nS` | 1 |
| `nC` | 8 |
| `n_CPU` | 1 |
| `RAM` | 2GB |
| `Training Samples` | $50,000 \rightarrow 25,000$ |
| `Cluster Strategy` | Data Partitioning Type |
| `Cluster Criteria` | IID vs non-IID |

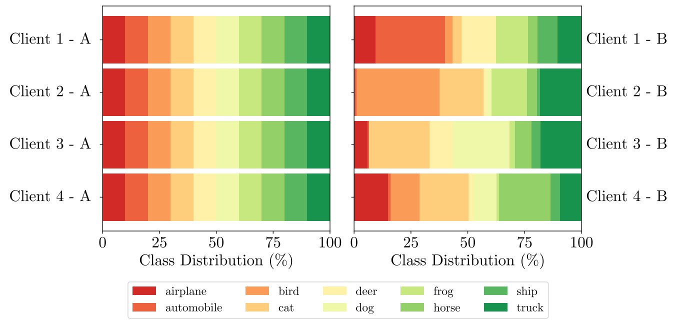


Fig. 8: IID (Cluster A) vs non-IID (Cluster B) distribution.

across variables. Here, $\alpha$ defines the degree of imbalance, allowing adjustments to test varying levels of data non-uniformity in partitioning. The number of training examples in these experiments is reduced from $50,000$ to $25,000$ to allow a different distribution of non-IID data. This change is due to the CIFAR-10 dataset maximum number of samples being $5,000$ per class (across 10 classes). This way, we assign a different amount of samples across classes to Cluster B, ensuring fair and comparable experiment configurations. Figure 9a depicts the difference in data partitioning for Clusters A and B, highlighting the percentage of examples for each class. In Cluster A, each class in the CIFAR-10 dataset has $2,500$ training examples, while Cluster B displays a non-uniform distribution, such as 500 examples for *dog* and $4,500$ for *ship* classes, ensuring that the total number of training examples remains equal between the two clusters.

Table VIII collects the three different analyzed configurations, i.e., $4a4b$, $5a3b$, and $6a2b$. Each configuration label, such as $4a4b$, represents the specific number of Clients A with IID data and Clients B with non-IID data (e.g., $4a4b$ means 4 Clients A and 4 Clients B). Each configuration is evaluated with and without applying the Client Cluster pattern. The clustering methodology generates two distinct global models: model A, trained with Clients A, and model B, trained with Clients B. This approach allows for a detailed analysis of how clustering influences the model performance across diverse data distributions. Non-clustered configurations remove the clustering component, leading all clients to collaborate on a single global model A, mixing IID and non-IID data. By progressively decreasing the proportion of non-IID clients (i.e., from $4a4b$ to $5a3b$ and $6a2b$), this setup enables a comprehensive assessment of how changes in the number of non-IID clients impact model training efficiency (i.e., better accuracy, lower training time). This highlights the impact of the *Client Cluster* pattern in effectively managing heterogeneous data distributions across clients.

TABLE VIII: Experiment configurations for Client Cluster.

| | **Config. *4a4b*** | **Config. *5a3b*** | **Config. *6a2b*** |
|---|---|---|---|
| `Client Cluster` | ✓, ✗ | ✓, ✗ | ✓, ✗ |
| `Dirichlet` $\alpha$ | 0.5 | 0.5 | 0.5 |
| `Data Distribution Type` | IID, non-IID | IID, non-IID | IID, non-IID |
| `Total Clients` | 4 A + 4 B | 5 A + 3 B | 6 A + 2 B |

✗*: Without Client Cluster pattern;* ✓*: With Client Cluster pattern.*

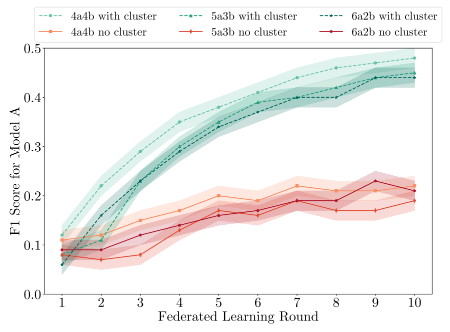
Results of this experiment are shown in Figure 9, where we report the accuracy (F1 Score), the average training time, and the efficiency metric evaluated on the ratio between accuracy and training time. Figure 9a depicts the F1 Score of each global model against FL rounds. We can notice that all the configurations with clustering (represented by dashed lines), show an overall higher F1 Score (except for the first round), demonstrating improved model accuracy and faster convergence over FL rounds. In contrast, non-clustered configurations (solid lines) yield lower F1 Scores, showing low model accuracy and slower convergence across FL rounds. Figure 9b presents the clients average training time during FL rounds. Configurations using the Client Cluster pattern exhibit shorter and more stable training time across rounds, e.g., the $5a3b$ configuration (with cluster) shows an observed training time varying between 110 and 118 seconds across the ten rounds. In contrast, configurations without the Client Cluster pattern exhibit significantly higher and more variable training times. For example, the $5a3b$ configuration (without cluster) reaches peaks from 186 seconds in round 3 up to 263 seconds in round 8. This aligns with practitioners' expectations, confirming that training times can increase when rounds include clients with non-IID data [18], [27]. Figure 9c depicts the F1 Score against the average training time, providing a metric for assessing the system performance efficiency. As highlighted by the trends, clustered configurations exhibit a higher efficiency ratio. In contrast, values assumed by configurations without the Client Cluster pattern, show lower efficiency.

*Architectural Implications.* Software architects may benefit from adopting this pattern in scenarios where clients exhibit high heterogeneity in their local data distribution, such as in the case of non-IID distributions. Experimental results indicate that using a clustering approach to group clients based on data partition characteristics accelerates global model convergence and reduces the average training time in FL systems.

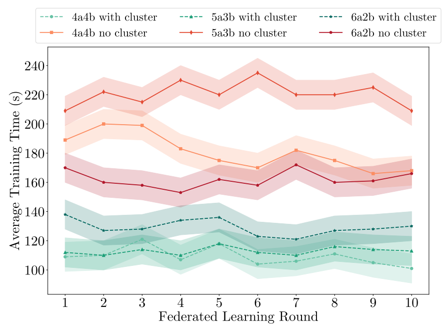
## C. Performance Analysis: Message Compressor

Table IX reports input parameters for the Message Compressor pattern experiment. This setup includes 10 FL rounds, with a single server and 8 clients, each one with 1 CPU and 2GB of RAM. This pattern implements a compression mechanism for exchanging global model data between clients and the central server, using version 1.3.1 of the *zlib* Python library [20] for both compression and decompression steps.

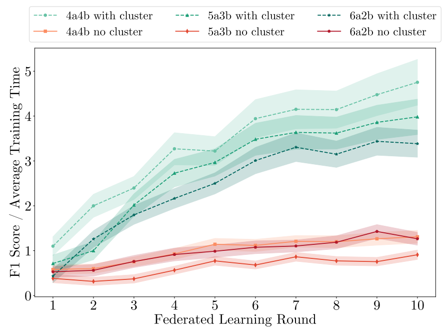
Table X reports three different experiment configurations for the Message Compressor pattern, specifically we progressively



(a) F1 Score per round



(b) Average Training Time per round



(c) F1 Score over Average Training Time

Fig. 9: Performance analysis of the Client Cluster Pattern.

| **Parameter** | **Value** |
|---|---|
| `NUM_ROUNDS` | 10 |
| `nS` | 1 |
| `nC` | 8 |
| `CPU` | 1 |
| `RAM` | 2GB |
| `Compression Library` | *zlib* [20] |

TABLE IX: Input parameters for Message Compressor experiments.

increase the global model size by adjusting the number of convolutional, pooling, and fully connected layers to scales of $n/2$, $n$, and $n * 2$. Here, $n$ represents the global model size determined by the structure described in Section III. Each experiment is conducted with and without the Message Compressor pattern, thus contributing to assessing its impact.

TABLE X: Experiment configurations of Message Compressor.

| | **Config. n/2** | **Config. n** | **Config. n ∗ 2** |
|---|---|---|---|
| `Message Compressor` | ✓, ✗ | ✓, ✗ | ✓, ✗ |
| `Model Structure` | | | |
| `Conv1` | 3 filters, 5$x$5 kernel | 6 filters, 5$x$5 kernel | 12 filters, 5$x$5 kernel |
| `Pool` | Max pooling, 2$x$2 kernel | Max pooling, 2$x$2 kernel | Max pooling, 2$x$2 kernel |
| `Conv2` | 8 filters, 5$x$5 kernel | 16 filters, 5$x$5 kernel | 32 filters, 5$x$5 kernel |
| `FC1` | 60 units | 120 units | 240 units |
| `FC2` | 42 units | 84 units | 168 units |
| `FC3` | 10 units | 20 units | 30 units |
| `Batch Size` | 32 | 32 | 32 |
| `Learning Rate` | 0.001 | 0.001 | 0.001 |
| `Optimizer` | SGD | SGD | SGD |

✗*: Without Message Compressor pattern;* ✓*: With Message Compressor pattern.*

Figure 10 illustrates the results of the experiments. For each configuration, we depict the average communication time. The shaded areas capture the reduction or the increasing in the estimated time, in plain or framed colored areas, respectively. This enables a direct comparison of performance with and without the Message Compressor architectural pattern, facilitating an assessment of its impact using the same model size. Figure 10a shows the communication time obtained with the $n/2$ configuration. In this case, we can observe an increase in communication time when the pattern is used, suggesting a possible overhead introduced by the compression and decompression process. Figure 10b shows the communication time for model size $n$. From round 4 onward, using compression slightly reduces communication time, showing a marginal benefit compared to the previous experiment. Figure 10c shows the results for the $n * 2$ configuration. Applying the pattern here results in a reduction in communication time across all rounds, suggesting higher gains in terms of reducing communication time when applying compression to larger models. In Figure 10d, we compare the relative improvement in communication time across the three configurations. To quantify this impact, we calculate the relative improvement in communication time by comparing configurations with and without compression. The formula is as follows:

$$\text{Improvement (\%)} = \left( \frac{T_{\text{no compression}} - T_{\text{compression}}}{T_{\text{no compression}}} \right) \times 100$$

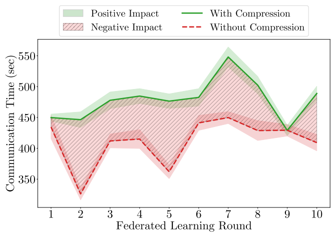
Here, $T_{\text{no compression}}$ represents the average communication time for clients without compression, while $T_{\text{compression}}$ indicates the average communication time for clients with the compression mechanism being applied. The formula provides a metric for improvement, i.e., the percentage reduction in communication time achieved through compression. Positive values indicate a reduction in communication time, while negative values reflect an increase. For the smaller model size ($n/2$), enabling the Message Compressor pattern results in a negative impact on communication time, indicating that compression may be less effective or counterproductive with smaller models. The experiment with model size set to $n$ initially shows a negative impact during the first rounds, it turns to an improvement in later rounds. In contrast, the larger model size ($n*2$) consistently shows improvement in communication time across all rounds, indicating that the Message Compressor is particularly effective for larger models.

*Architectural Implications.* Software architects can consider this architectural pattern when managing large global models. Experimental results indicate that, as the global model size grows, implementing compression becomes increasingly relevant, leading to a significant reduction in communication time that improves the overall system latency.

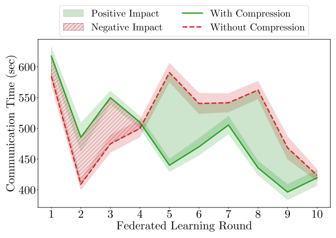
## V. Discussion

### A. *Limitations on Architectural Patterns*

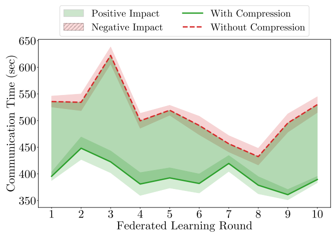
*Client Registry.* This pattern is fundamental in our work to implement other architectural patterns we consider within FL.



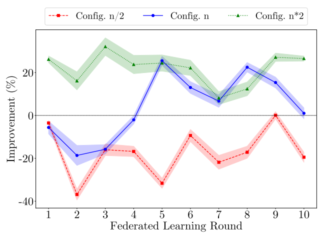
(a) Communication Time for $n/2$



(b) Communication Time for $n$



(c) Communication Time for $n \times 2$



(d) Configurations Comparison

Fig. 10: Performance analysis of the Message Compressor pattern.

It is assessed to enhance both maintainability and reliability in FL systems by centralizing client devices' information [10]. However, centralizing sensitive client data on the server can lead to privacy and security risks in case of malicious attacks [10]. As future work, we can mitigate these risks, e.g., an SSL/TLS authentication system can ensure secure communication between clients and the server [15].

*Client Selector.* The client selection process optimizes resource usage and system scalability by sampling only clients that are best suited for contributing to global model improvements [7], [15]. In our experiments, we test a resource-based selection strategy, which confirms that this pattern significantly reduces total round time in FL. By including one Low-Spec client, we observe that the total round time increases up to $9\times$, as Low-Spec clients generate significant system bottlenecks. Indeed, when "slow" clients are included in the FL round, other nodes of the system (i.e., clients and server) must wait for slow devices to complete local training and model parameters communication before starting a new round. While this is true for the specific parameters set in our experiments (see Section IV), we are confident that similar behavior can be observed whenever there are further disparities in computational client resources. For instance, data-based selection criteria select only clients with high-quality data (i.e., rich and varying training samples). A limitation of the Client Selector pattern is that excluding clients may lead to a loss of valuable and unique data, potentially reducing the final global model accuracy and limiting its generalization. This is due to some clients holding exclusive portions of data from which the model could learn unique patterns. Additionally, the need to frequently assess clients (i.e., by verifying if they are eligible for the FL round) introduces communication and computational costs, that can significantly increase in systems with a high number of participating devices. Both data-based selection strategy and the overhead of continuously evaluating clients require further investigation, which we plan to address in future work.

*Client Cluster.* Our experiments on the Client Cluster pattern focus on evaluating its impact on the system performance using a data-partition clustering strategy. By grouping clients with similar data distributions (e.g., IID vs. non-IID clusters), this pattern improves the efficiency of the training process in two main directions. First, clustering clients based on data similarity mitigates the impact of non-IID data, which reduces gradient variance and leads to a faster training process. Second, the training process benefits from enhances in local model convergence, resulting in improved global model accuracy [10]. Our results show that clustered configurations significantly outperform non-clustered configurations, achieving higher accuracy while also reducing the average round training time. However, there are some drawbacks that software architects need to consider when implementing a Client Cluster pattern. The central server needs additional computational costs and processing time for clustering clients and quantifying their relationships, which can impact system efficiency in large scenarios [10]. Furthermore, clustering requires the server to access extra client information, increasing the risk of exposing sensitive data. As future work, we can use aggregated or anonymized metrics for clustering, thus mitigating the risk of revealing sensitive data [15].

*Message Compressor.* Experiments conducted in this paper show a varying impact of the message compression mechanism, depending on the size of the global model. For smaller models, as in the $n/2$ configuration, the compression introduces a notable overhead, resulting in increased communication time. Here, the computational cost associated with compression outweighs the benefits, as evidenced by consistently higher times across most rounds. In the $n$ configuration, the reduction in communication time with compression, although small, shows an improvement. At this scale, the compression benefits start being visible and lead to a slight improvement in communication time. When the model size doubles (e.g., in the $n$*2 configuration) the advantage of compression becomes more evident. In this case, compressing data significantly shortens the communication time, fully compensating for the overhead introduced by compression and decompression steps. Given that the compression (and decompression) process may introduce additional computational costs, as future work we plan to introduce a trade-off analysis between the size of the global model and the overhead introduced by (de)compression mechanisms, thus enabling compression only when beneficial.

## B. Threats to Validity

Besides inheriting all limitations of architectural patterns and performance analysis [28], [29], our approach exhibits the following threats to validity [30].

*External validity*, i.e., generalization of results, is not guaranteed, since our experimental setup is subject to certain limitations due to the available physical resources. Our commodity hardware comes with a limited number of physical processors, thus restricting the actual number of client devices in our experiments. Consequently, our quantitative analysis is representative for a small client population, which may not fully reflect the dynamics of large-scale FL networks. Nevertheless, experimental results provide valuable insights into the potential applicability of our methodology.

*Internal validity*, i.e., settings and parameters used for performance analysis, is also exposed to potential threats, since we use numerical values that provide evidence of performance variations for each pattern. For instance, we vary CPU allocations among clients to highlight the client selector's response to differing resource availability. Additionally, by maintaining consistent input settings across experiments for each pattern, we minimize misleading effects, ensuring that observed outcomes could be reliably attributed to specific architectural choices. However, determining accurate numerical values for input parameters is an ever existing challenge in software performance [31] and other settings could be further investigated. As an example, clustering clients considering different configurations (i.e., clients holding different data distribution types) may reveal varying performance characteristics. Accordingly, we remark that experiment setup instructions for testing different input parameter values are publicly available [13], and software architects can modify parameters to conduct experiments as needed.

*Construct validity*, i.e., the statistical validity of the experimental results, is established by repeating each experimental configuration 10 times, averaging output values, and calculating the 99% confidence interval to assess the accuracy of the reported numerical results.

## VI. Related work

*Architectural Patterns.* The motivation of our work is supported by a large literature underscoring the importance of adopting architectural patterns in designing software systems [32]–[34]. Besides, the software architecture community recently enforce the adoption of patterns to address design and operational challenges in machine learning systems [35]–[38]. Washizaki et al. [35] conduct a study to classify architectural patterns specifically designed for machine learning systems, providing developers with a framework for selecting suitable architectural solutions. This work highlights the need for systematic performance evaluation of these patterns, i.e., the main contribution of our research. Leest et al. [39] and Takeuchi et al. [40] present structured approaches to address real-world design challenges in machine learning systems.

*Federated Learning.* Compared to centralized machine learning, FL offers additional advantages regarding efficiency and privacy aspects [10], [11]. However, FL needs to face additional architectural challenges, i.e., managing server-client interactions. Di Martino et al. [11] and Lo et al. [2] propose reference architectures to design FL systems. These architectures integrate the patterns discussed in this work, underscoring the value of structured approaches for improving system robustness. However, they lack quantitative performance evaluation, leaving a gap in the empirical assessment of proposed systems, as we do in this paper. Ma et al. [41] present a framework for FL that uses the Client Selector pattern to identify and exclude unreliable clients, thus preserving system performance by avoiding faulty updates. Pavlidis et al. [42] expand on client selection by introducing an algorithm that prioritizes clients based on computational power and network conditions, aiming to reduce delays from slower clients while achieving good model accuracy. Both studies highlight the advantages of client selection strategies, suggesting that further research is needed to evaluate their impact on performance, which aligns with the contribution of our work.

The closest related methodologies are reported hereafter. Lai et al. [9] propose a FL benchmark engine. Differently from our investigation, the authors focus on building a standardized benchmarking suite for FL systems rather than providing an empirical assessment of how architectural patterns affect system performance. Casalicchio et al. [43] propose a workbench platform for the performance evaluation of FL systems implementing patterns proposed by [10]. We initially considered this work for comparison, but it lacks instructions on how to reproduce the experiment results and access to the proposed tool, making the correlation impossible.

To summarize, to the best of our knowledge, most related studies emphasize the importance of implementing architectural patterns in designing FL systems. Our work addresses the lack of publicly available methodologies that evaluate their impact and implications on system performance.

## VII. Conclusion

This paper supports software architects with an investigation of how architectural patterns affect the performance of FL systems, thus providing a quantitative evaluation of a subset of design decisions. To this end, we implement and evaluate four architectural patterns deemed relevant for system performance in the literature [10]. Our findings indicate that architectural patterns are valid candidates to address FL design challenges, although pros and cons need to be considered, i.e., the performance improvement comes at the cost of introducing overhead to actually apply these patterns. Specifically, the *Client Registry* pattern centrally stores client data, enhancing client management and serving as a foundation for other patterns. The *Client Selector* reduces total round time by exploiting the information collected on clients. The *Client Cluster* reduces training time and increases model accuracy by grouping clients on the basis of data similarities. The *Message Compressor* reduces communication time for large models, although a trade-off between compression overhead and model size is necessary to achieve optimal efficiency.

In future work, besides addressing all limitations and threats to validity discussed above, we also plan to extend our experiments by investigating potential combinations of different patterns, e.g., applying the Client Selector to clustered clients.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work has been partially funded by the MUR-PRIN project 20228FT78M DREAM (modular software Design to Reduce uncertainty in Ethics-based cyber-physicAl systeMs), MUR Department of Excellence 2023 - 2027 for GSSI, and PNRR ECS00000041 VITALITY.

## References

[1] S. Banabilah, M. Aloqaily, E. Alsayed, N. Malik, and Y. Jararweh, "Federated Learning Review: Fundamentals, Enabling Technologies, and Future Applications," *Information Processing & Management*, vol. 59, no. 6, p. 103061, 2022.

[2] S. K. Lo, Q. Lu, H. Paik, and L. Zhu, "FLRA: A Reference Architecture for Federated Learning Systems," in *European Conference on Software Architecture (ECSA)*, vol. 12857, 2021, pp. 83–98.

[3] 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.

[4] G. Drainakis, K. V. Katsaros, P. Pantazopoulos, V. Sourlas, and A. Amditis, "Federated vs. Centralized Machine Learning Under Privacy-Elastic Users: A Comparative Analysis," in *International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–8.

[5] J. Liu, J. Huang, Y. Zhou, X. Li, S. Ji, H. Xiong, and D. Dou, "From Distributed Machine Learning to Federated Learning: A Survey," *Knowledge and Information Systems*, vol. 64, no. 4, pp. 885–917, 2022.

[6] L. Perri, "What's New in the 2023 Gartner Hype Cycle for Emerging Technologies," https://www.gartner.com/en/articles/what-s-new-in-the-2023-gartner-hype-cycle-for-emerging-technologies, 2023, Gartner, Inc.

[7] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, "A Survey on Federated Learning," *Knowledge-Based System*, vol. 216, p. 106775, 2021.

[8] L. Baresi, G. Quattrocchi, and N. Rasi, "Open challenges in federated machine learning," *IEEE Internet Comput.*, vol. 27, no. 2, pp. 20–27, 2023.

[9] F. Lai, Y. Dai, S. Singapuram, J. Liu, X. Zhu, H. Madhyastha, and M. Chowdhury, "FedScale: Benchmarking Model and System Performance of Federated Learning at Scale," in *Proceedings of Machine Learning Research (PMLR)*, 2022, pp. 11 814–11 827.

[10] S. K. Lo, Q. Lu, L. Zhu, H.-Y. Paik, X. Xu, and C. Wang, "Architectural Patterns for the Design of Federated Learning Systems," *Journal of Systems and Software*, vol. 191, p. 111357, 2022.

[11] B. D. Martino, D. D. Sivo, and A. Esposito, "Architectural Patterns for Software Design Problem-Solving in the Implementation of Federated Learning Structures Within the E-Health Sector," in *International Conference on Advanced Information Networking and Applications (AINA)*, vol. 203, 2024, pp. 347–356.

[12] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, and N. D. Lane, "Flower: A Friendly Federated Learning Research Framework," *CoRR*, vol. abs/2007.14390, 2020.

[13] I., Compagnucci and R., Pinciroli and C., Trubiani, "Open Science Artifact: Performance Analysis of Architectural Patterns for Federated Learning Systems," https://zenodo.org/records/14539962, 2025.

[14] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 54, 2017, pp. 1273–1282.

[15] P. Kairouz *et al.*, "Advances and open problems in federated learning," *Foundations and Trends in Machine Learning*, vol. 14, no. 1-2, pp. 1–210, 2021.

[16] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *International Conference on Neural Information Processing Systems (NeurIPS)*, 2019, pp. 8024–8035.

[17] C. Briggs, Z. Fan, and P. Andras, "Federated Learning with Hierarchical Clustering of Local Updates to Improve Training on Non-IID Data," in *International Conference on Neural Network*, 2020, pp. 1–9.

[18] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang, "On the Convergence of FedAvg on Non-IID Data," in *International Conference on Learning Representations, (ICLR)*, 2020.

[19] A. Nilsson, S. Smith, G. Ulm, E. Gustavsson, and M. Jirstrand, "A Performance Evaluation of Federated Learning Algorithms," in *International Workshop on Distributed Infrastructures for Deep Learning (DIDL)*, 2018, pp. 1–8.

[20] P. Deutsch and J. Gailly, "ZLIB Compressed Data Format Specification version 3.3," *RFC*, vol. 1950, pp. 1–11, 1996.

[21] M. Adler and J.-L. Gailly, "zlib: A Data Compression Library," 2024. [Online]. Available: https://github.com/madler/zlib

[22] A. Krizhevsky, G. Hinton *et al.*, "Learning Multiple Layers of Features from Tiny Images," Tech. Rep., 2009.

[23] M. C. Cohen, P. W. Keller, V. S. Mirrokni, and M. Zadimoghaddam, "Overcommitment in Cloud Services: Bin Packing with Chance Constraints," *Management Science*, vol. 65, no. 7, pp. 3255–3271, 2019.

[24] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, "Model Pruning Enables Efficient Federated Learning on Edge Devices," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 12, pp. 10 374–10 386, 2022.

[25] N. W. S. Wardhani, M. Y. Rochayani, A. Iriany, A. D. Sulistyono, and P. Lestantyo, "Cross-Validation Metrics for Evaluating Classification Performance on Imbalanced Data," in *International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, 2019, pp. 14–18.

[26] M. Yurochkin, M. Agarwal, S. Ghosh, K. H. Greenewald, T. N. Hoang, and Y. Khazaeni, "Bayesian Nonparametric Federated Learning of Neural Networks," in *International Conference on Machine Learning (ICML)*, vol. 97, 2019, pp. 7252–7261.

[27] H. Yu, S. Yang, and S. Zhu, "Parallel Restarted SGD with Faster Convergence and Less Communication: Demystifying Why Model Averaging Works for Deep Learning," in *International Conference on Artificial Intelligence, (AAAI)*, 2019, pp. 5693–5700.

[28] A. Shokri, J. C. S. Santos, and M. Mirakhorli, "ArCode: Facilitating the Use of Application Frameworks to Implement Tactics and Patterns," in *International Conference on Software Architecture (ICSA)*, 2021, pp. 138–149.

[29] Y. Zhao, L. Xiao, X. Wang, Z. Chen, B. Chen, and Y. Liu, "Butterfly Space: An Architectural Approach for Investigating Performance Issues," in *International Conference on Software Architecture (ICSA)*, 2020, pp. 202–213.

[30] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén *et al.*, *Experimentation in Software Engineering*, 2012, vol. 236.

[31] A. B. Bondi, *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice.*, 2015.

[32] R. C. Martin, "Design Principles and Design Patterns," *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[33] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., ser. SEI series in software engineering. Addison-Wesley Professional, 2012.

[34] Z. Wan, Y. Zhang, X. Xia, Y. Jiang, and D. Lo, "Software Architecture in Practice: Challenges and Opportunities," in *Conference and Symposium on the Foundations of Software Engineering, (FSE).* ACM, 2023, pp. 1457–1469.

[35] H. Washizaki, H. Uchida, F. Khomh, and Y.-G. Guéhéneuc, "Studying Software Engineering Patterns for Designing Machine Learning Systems," in *International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2019, pp. 49–495.

[36] K. A. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards Federated Learning at Scale: System Design," in *Conference on Machine Learning and Systems*, 2019.

[37] S. J. Warnett and U. Zdun, "Architectural Design Decisions for Machine Learning Deployment," in *International Conference on Software Architecture, (ICSA)*, 2022, pp. 90–100.

[38] E. Ntentos, S. J. Warnett, and U. Zdun, "Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures," in *International Conference on Software Architecture, (ICSA)*, 2024, pp. 90–100.

[39] J. Leest, I. Gerostathopoulos, and C. Raibulet, "Evolvability of Machine Learning-based Systems: An Architectural Design Decision Framework," in *International Conference on Software Architecture (ICSA)*, 2023, pp. 106–110.

[40] H. Takeuchi, T. Doi, H. Washizaki, S. Okuda, and N. Yoshioka, "Enterprise Architecture based Representation of Architecture and Design

Patterns for Machine Learning Systems," in *Enterprise Distributed Object Computing Workshop, (EDOC)*, 2021, pp. 245–250.

[41] C. Ma, J. Li, M. Ding, K. Wei, W. Chen, and H. V. Poor, "Federated Learning With Unreliable Clients: Performance Analysis and Mechanism Design," *IEEE Internet Things J.*, vol. 8, no. 24, pp. 17 308–17 319, 2021.

[42] N. Pavlidis, V. Perifanis, T. P. Chatzinikolaou, G. C. Sirakoulis, and P. S. Efraimidis, "Intelligent Client Selection for Federated Learning using Cellular Automata," *CoRR*, vol. abs/2310.00627, 2023.

[43] E. Casalicchio, S. Esposito, and A. A. Al-Saedi, "FLWB: a Workbench Platform for Performance Evaluation of Federated Learning Algorithms," in *International Workshop on Technologies for Defense and Security (TechDefense)*, 2023, pp. 401–405.