Experimenting Architectural Patterns in Federated Learning Systems

Ivan Compagnucci[a], Riccardo Pinciroli[b], Catia Trubiani[a]

[a]*Gran Sasso Science Institute, L'Aquila, 67100, Italy*
[b]*Zimmer Biomet, Milan, 20054, Italy*

Abstract

Federated Learning has emerged as a promising paradigm that enables collaborative model training while preserving data privacy, thus contributing to enhance user trust. However, the design of Federated Learning systems requires non-trivial architectural choices to address several challenges, such as system efficiency and learning accuracy. Architectural patterns for Federated Learning systems have been defined in the literature to handle these challenges, but their experimentation is limited. The objective of this paper is to empower software architects in their task of evaluating the design of FL systems while deciding which architectural alternatives are more beneficial in their context of adoption. Our methodology consists of a tool-based approach that embeds the implementation of six architectural patterns defined in the literature. The advantage is that software architects can select design alternatives either in isolation or in a combined fashion, and the subsequent analysis provides the evaluation of some metrics of interest. The experimental results indicate that architectural patterns can enhance system efficiency, although we found a combination of patterns that added overhead and turned to limit its benefit. By quantifying these trade-offs, we aim to support software architects in designing Federated Learning systems by evaluating the benefits and drawbacks of applying architectural patterns.

Keywords: Architectural Patterns, Federated Learning, Quantitative Evaluation

1. Introduction

Modern digital devices generate massive volumes of data that grow exponentially and contribute to feed advanced Machine Learning (ML) tech-

niques [25]. Data represents a crucial aspect in this context, since several factors play a key role during learning, e.g., data representativeness directly influences predictive performance [3]. With the increasing reliance of ML on sensitive information, data privacy has become a critical concern [28]. Consequently, software solutions must ensure both trustworthiness and compliance with legal requirements, such as the European General Data Protection Regulation (GDPR) [22]. To face these challenges, in 2016 Google introduced Federated Learning (FL), i.e., a paradigm that enables devices to collaboratively train a shared global model while keeping sensitive data locally [44]. This approach has rapidly emerged as a practical and effective solution to handle data privacy and enhance user trust [73, 57], fostering its adoption across several domains, from healthcare to mobile applications [29]. However, recent studies argue that designing FL systems is still challenging [5]. For instance, the optimization of system performance emerges as a relevant concern, as FL systems must balance data protection, communication efficiency, optimal resource management, and model accuracy [70].

State-of-the-art research on FL primarily focuses on optimizing learning algorithms [45, 37], model compression [76, 58], and parameters' aggregation strategies [52, 27]. The system performance is instead handled by a few approaches [33, 14], even if it is well assessed that system performance problems may hurt the functioning [29], especially in real-world deployments aiming at guaranteeing system efficiency along with reliability and user trust [73]. In this context, the design of FL systems represents the primary abstraction that contributes to the subsequent development choices, the architectural decisions become of key importance since they constitute the foundations of the FL process [41, 33]. Besides, architectural inefficiencies may lead to training failures and resource depletion, potentially undermining the trustworthiness properties that FL aims to ensure [29, 73]. To address these challenges, in the literature there is a recent effort of adopting architectural patterns specifically perceived for FL systems [41]. In software engineering, architectural patterns represent reusable solutions to a problem that occurs commonly within a given context in software design [7]. The benefit of relying on the specification of patterns is that they provide a pool of design alternatives that may represent valid options for software architects [41].

The overarching research question we address in this paper is: *What is the quantitative impact of applying different design alternatives on FL systems?* We propose an approach that evaluates the behavior of FL systems, thereby supporting software architects in taking informed design decisions.

In our previous work [18], we presented a quantitative evaluation of architectural patterns for FL systems using Flower [9], i.e., a Python library for FL. By extending its codebase, we implemented and evaluated four architectural patterns from [41]: the *Client Registry*, which provides a centralized storage of client data enabling efficient management, the *Client Selector*, which reduces training round time by intelligently filtering clients, the *Client Cluster*, which improves both training time and model accuracy by grouping clients based on data similarities, and the *Message Compressor*, which optimizes communication overhead while balancing compression costs.

The objective of our research is to support software architects in understanding the impact of design alternatives on FL systems. To this end, we propose AP4Fed, a benchmark framework that enables the selection and the combination of six architectural patterns. The outcome of the analysis consists of reports that estimate some metrics of interest, specifically we target the following two main indicators: (i) *system performance*, that represents the computational overhead generated by the FL process, and (ii) *predictive performance*, that refers to the learned model accuracy. In this paper, two novel architectural patterns are implemented and evaluated: (*i*) *Heterogeneous Data Handler*, which aims to improve the quality of datasets owned by clients participating in the FL process; (*ii*) *Multi-Task Model Trainer*, which is designed to train separated but related ML tasks simultaneously. In addition, we experiment with the combination of previously defined patterns [18] with the newly introduced ones, specifically: (*i*) *Heterogeneous Data Handler* is combined with *Message Compressor*, thus studying the joint optimization of data quality and communication efficiency; (*ii*) *Multi-Task Model Trainer* is combined with *Client Selector*, thus investigating the joint effort of optimizing model accuracy and system efficiency. Summarizing, the main contributions of this manuscript are:

1. The development of AP4Fed [19], i.e., a tool that supports software architects in experimenting different design decisions through the selection and combination of six architectural patterns;

2. The implementation of two architectural patterns and their experimentation either in isolation or in combination with other patterns, while measuring system performance and predictive performance;

3. Empirical results showcase benefits and drawbacks of architectural patterns, thus providing quantitative evidence that supports software architects in understanding pattern(s) trade-offs.

The rest of the manuscript is organized as follows. Section 2 provides background knowledge on FL systems and architectural patterns. Section 3 describes the methodology and the framework used for the experiments. Section 4 reports the analysis of architectural patterns, both in isolation and in a combined fashion. Section 5 argues on the obtained experimental results and their implications. Section 6 discusses related work and highlights the main differences with our research. Section 7 concludes the paper by outlining the main contributions and directions for future research.

2. Background

2.1. Federated Learning

Federated Learning (FL) emerged as a solution to the growing challenge of utilizing a large amount of sensitive and personal data [44, 41]. While this data is invaluable for training ML models capable of improving decision-making in software systems [70], its sensitive nature raises significant privacy concerns when shared with centralized servers [4]. To address this, FL enables multiple client devices to collaboratively train a global ML model using their local data under the coordination of a central server [41, 4, 70]. This distributed system allows to preserve data privacy and distributing computation across a network, FL not only addresses privacy concerns but also enhances scalability and reduces reliance on centralized processing resources [29].

Figure 1 depicts the main phases of the FL paradigm. It begins with a central server broadcasting the initial global model parameters (e.g., model weights and structure) to all participating clients ①. Once all clients receive these parameters, each device performs local model training using its private data ②. Upon completing the training, clients send their updated model parameters (i.e., the trained model weights) back to the central server ③. The server then aggregates these updates to produce a refined version of the global model ④. The updated global model is subsequently redistributed to the clients, initiating the next round of training. The process repeats iteratively, with the global model continuously improving until convergence is achieved. This mechanism ensures collaborative learning while preserving data privacy, as data remains decentralized throughout the process.

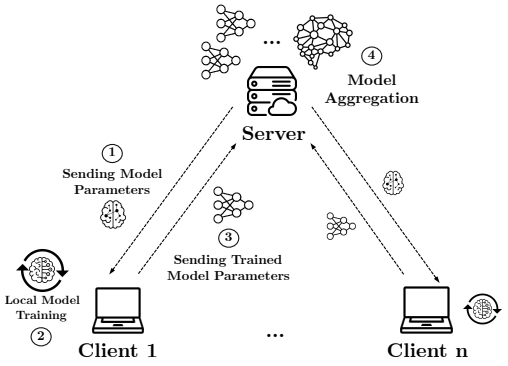


Figure 1: Federated Learning Overview, inspired by [41].

2.2. Architectural Patterns in Federated Learning Systems

Architectural patterns provide reusable solutions to recurring design challenges in complex systems [55]. They represent best practices in design, enabling developers and software architects to address concerns such as scalability and modularity while preserving system efficiency [41]. In our previous work [18], we implemented and evaluated a subset of architectural alternatives defined by Lo et al. [41], specifically: the (*i*) *Client Registry* that stores information on clients contributing to the training; (*ii*) the *Client Selector* that improves the system efficiency by selecting clients based on 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 this work, we keep focusing on quantitatively evaluating FL systems, and investigate two additional architectural patterns that are considered essential for enhancing system efficiency [41]. The first pattern, namely *Heterogeneous Data Handler*, performs pre-processing operations on local datasets to enhance data quality across all clients participating in the FL process. The second pattern, i.e., the so-called *Multi-Task Model Trainer*, enables the simultaneous training of multiple ML tasks. These two patterns are detailed in the following.

2.2.1. Heterogeneous Data Handler Architectural Pattern

Context. The Heterogeneous Data Handler pattern aims to improve the quality of datasets owned by clients participating in the FL process. This

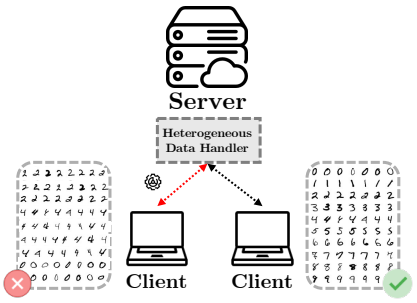


Figure 2: Heterogeneous Data Handler Pattern Overview.

architectural alternative consists of employing techniques such as: (*i*) *data augmentation*, which creates synthetic data to increase the diversity and volume of local datasets, and (*ii*) *federated distillation*, that periodically provides knowledge from other devices to clients (without directly accessing other devices' data) [41]. These techniques help balance data distributions across clients while preserving local data privacy and avoiding the need for centralized data collection [70]. The primary benefit of the Heterogeneous Data Handler pattern lies in its ability to enhance the generalization and accuracy of the global model by improving the quality of client datasets [41]. Through techniques like Generative Adversarial Networks (GANs), the architectural alternative consists of addressing challenges related to the imbalanced nature of the non-Independent and Identically Distributed (non-IID) dataset [38] by generating synthetic examples that increase diversity and fill gaps in underrepresented classes. This leads to a more effective training process and reduces reliance on uniform data distributions. However, this design alternative may introduce computational overhead from performing local operations, as well as privacy concerns since dataset analysis is required during augmentation and knowledge transfer.

Our Implementation. To implement the Heterogeneous Data Handler pattern, we leverage the `PyTorchGAN` [49] Python library, exploiting its capabilities to augment data through synthetic sample generation and thereby mitigating the imbalance of clients local non-IID datasets. Specifically, we employ a conditional GAN model trained to produce class-specific data samples, effectively augmenting underrepresented classes in each client dataset. The Heterogeneous Data Handler pattern is integrated by first analyzing the class distribution of each client dataset, and then generating synthetic data to balance class proportions. After applying GAN-based augmentation,

updated datasets exhibit reduced class imbalance, enabling more effective training of the global model. Results show that this approach mitigates challenges posed by non-IID data while preserving local data privacy.

2.2.2. Multi-Task Model Trainer Architectural Pattern

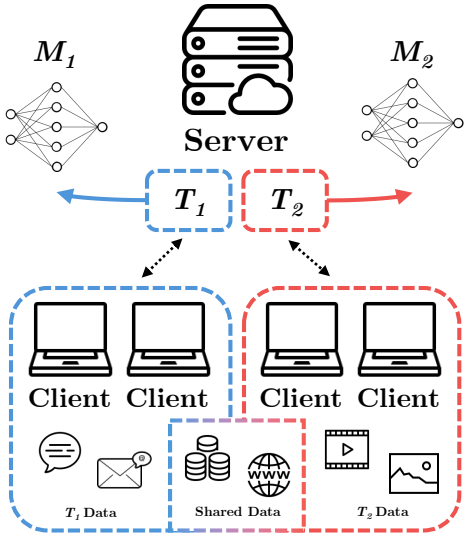


Figure 3: Multi-Task Model Trainer Pattern Overview.

Context. The Multi-Task Model Trainer pattern is designed to train separated but related ML tasks [41]. A ML task is a defined problem that a system addresses by identifying similarities and relationships from data [72]. It encompasses the model objective, the type of predictions or decisions it needs to make, and the structure of the data it processes. Examples may include image classification, speech recognition or time series forecasting [2, 72]. As shown in Figure 3, this architectural alternative consists of enabling scenarios where clients share data to train two ML tasks, $\mathtt{T}_1$ and $\mathtt{T}_2$. These tasks can subsequently result in two possible outcomes: (*i*) the generation of two distinct global models, which we will refer to as $\mathtt{M}_1$ and $\mathtt{M}_2$, or (*ii*) the combination of tasks into a single unified global model, referred to as $\mathtt{M}_3$ [41]. $\mathtt{M}_3$ is designed to tackle both tasks simultaneously by leveraging shared data, ensuring the model effectively achieves the objective of both tasks.

Challenges of statistical heterogeneity in FL systems, particularly when dealing with non-IID data across clients, often lead to poor model generalization and thereby lower accuracy [29, 60]. By considering additional clients

with relevant data for a specific task, this design alternative can partially mitigate the non-IID data problem by providing additional samples, thereby enhancing the global model accuracy. To better illustrate this approach, consider the example depicted in Figure 4, which shows a robot performing two interconnected computer vision tasks, i.e., the semantic segmentation and the depth estimation. Semantic segmentation allows the robot to identify and classify objects within its environment [56], while depth estimation determines the distance and spatial structure of the surrounding area [30]. By combining semantic segmentation and depth estimation, the robot enhances its ability to navigate and interact with the environment more precisely. The multi-task learning approach builds upon shared knowledge, thus the global model can adapt to heterogeneous data across clients [41, 60].

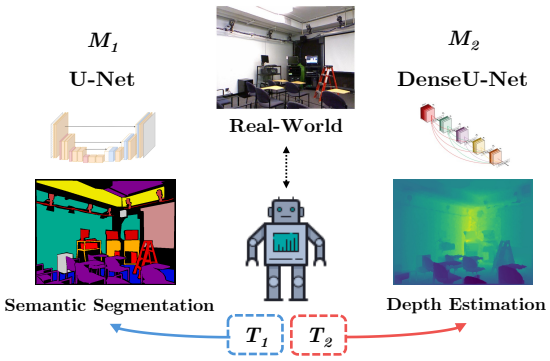


Figure 4: The Multi-Task Model Trainer Pattern in the Robot Use Case.

Our Implementation. To implement the Multi-Task Model Trainer pattern, we design a FL simulation capable of simultaneously training multiple tasks by leveraging both shared and task-specific data. The training process is organized as follows: each client performs local updates for both $\texttt{T}_1$ and $\texttt{T}_2$ by computing gradients and weights independently for each task. These updates are then aggregated at the server level where a single unified model ($\texttt{M}_3$) is generated. To achieve this, AP4Fed extends the Flower standard `configure_fit`[1] method, which is responsible for configuring the upcoming training round [9]. AP4Fed incorporates a strategy that dynamically directs the updated weights provided by clients to the corresponding task during the

[1]Further details are available at `https://flower.ai/docs/framework/how-to-implement-strategies.html`

training process. This ensures that each client's local training data effectively contributes to the global model for its respective task(s). After completing their local training, clients send to the server their updated weights, which are then aggregated using a weighted averaging mechanism based on the number of samples processed by each client.

3. Methodology

Introduction. Our methodology consists of developing AP4Fed, a flexible FL benchmark framework that enables software architects to design, configure, and evaluate FL systems through the composition of mixed architectural patterns, enabling quantitative evaluation of their impact. The open source code and the documentation on how to use the developed tool is publicly available [19].

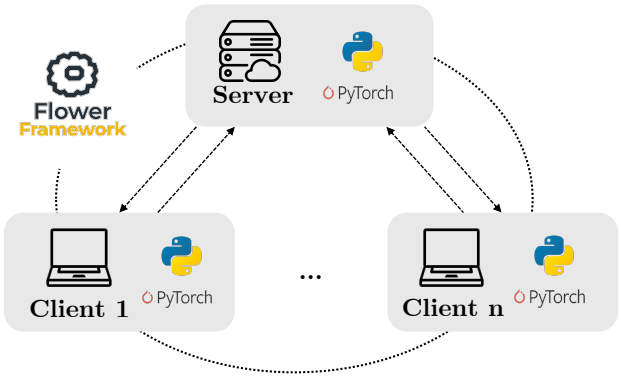


Figure 5: AP4Fed Infrastructure Overview.

The AP4Fed infrastructure, depicted in Figure 5, builds on the Flower Python library (v1.12.0) [9], a framework designed to support the configuration and simulation of FL systems. Flower provides core functionalities such as client-server communication, global model updates, and round-based coordination, which serve as the foundation for AP4Fed. Beyond these core features, AP4Fed introduces a set of architectural patterns and metrics to benchmarking FL simulations. These additions enable the extraction of advanced metrics and provide greater flexibility for configuring and evaluating diverse FL scenarios. The framework leverage the PyTorch library (v2.5.0) [50] for ML tasks, allowing each client to perform local training on

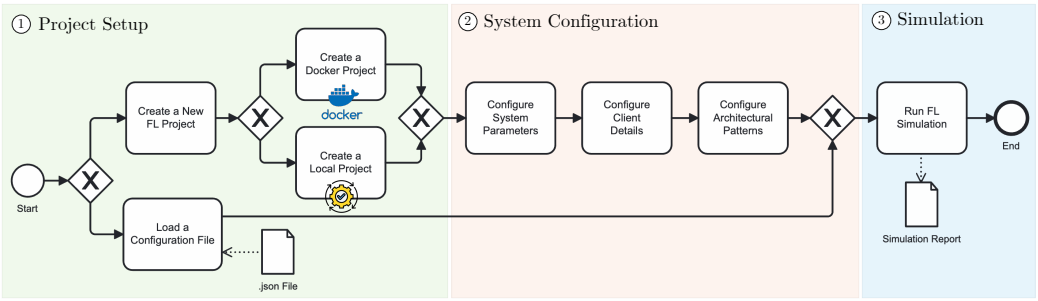


Figure 6: BPMN Graphical Representation of the AP4Fed Framework.

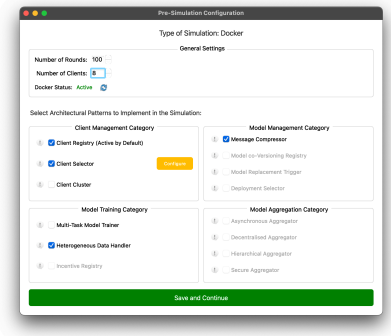
its data. AP4Fed uses `PyQt5` (v5.15.11) to provide a user-friendly Graphical User Interface (GUI) that simplifies the process of configuring simulation parameters and modeling FL systems for experimentation. This interface guides users through the setup process, thus enabling the emulation of typical real-world FL architectures.

Methods. Figure 6 presents a BPMN diagram [17] that illustrates how AP4Fed works. Its core workflow is split into three phases.

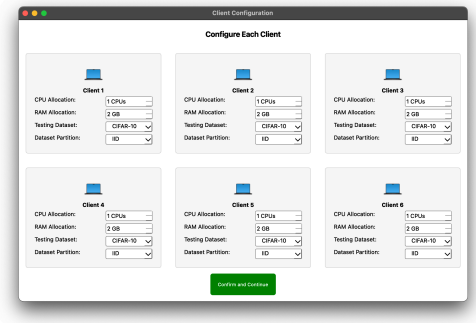
The ① *Project Setup* phase enables users to either start a new FL simulation or load an existing configuration from a `JSON` file. In the former case, the simulation can be configured to run in a local environment (to experiment with different ML strategies and algorithms) or in a container-based setup (to emulate real-world systems, ensure efficient resource allocation, and prevent resource overcommitment [16]). In the latter one, users can customize predefined settings defined in the `JSON` file.

The second phase of the AP4Fed workflow is ② *System Configuration*. Figure 6 shows that the configuration involves three main tasks: (i) system parameters, (ii) client details, and (iii) architectural patterns. Figure 7 reports the configuration GUI of AP4Fed, specifically: system parameters (e.g., the number of rounds), see Figure 7a; client details (e.g., number of allocated CPUs), see Figure 7b; architectural patterns (e.g., the client selector requires a selection strategy), see Figure 7c. A graphical overview of all parameters is presented in Figure 7d.

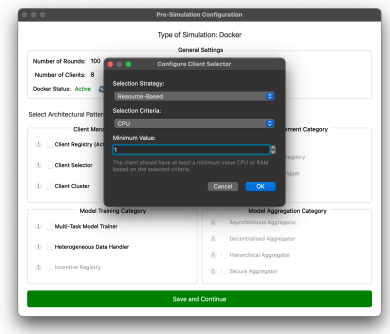
Table 1 summarizes the parameters in a tabular format. The system parameters are: the `simulation_Type` that can be Local or Docker, the number of FL rounds (`num_Rounds`) that can vary between 1 and r, and the number of clients (`nC`) whose minimum value is 2 and the maximum can be set to c. Each client is detailed with an identifier (`client_ID`), the resource



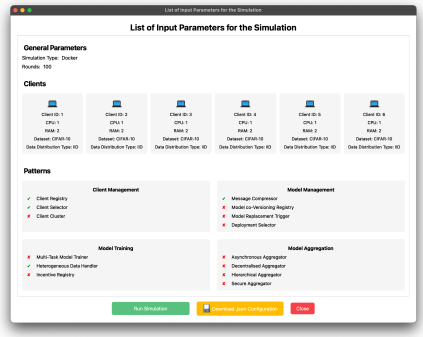
(a) Configuration of System Parameters.



(b) Configuration of Client Details.



(c) Configuration of Architectural Patterns.



(d) Parameters Overview.

Figure 7: AP4Fed: Simulation Configuration GUI.

Parameter	Description	Possible Value
System Parameters		
`simulation_Type`	Type of FL Simulation	"Local","Docker"
`num_Rounds`	no. of FL Rounds	1, r
`nC`	no. of Clients	2, c
Client Details		
`client_ID`	Incremental Client ID	1, ..., c
`n_CPU`	no. of allocated CPUs	1, n
`RAM`	Allocated Memory Capacity	1, m
`dataset`	Testing Dataset	"CIFAR-10", ...
`data_Distribution`	Data Distribution Type	"IID", "non-IID", "Random"
Architectural Patterns		
✎ `CLIENT REGISTRY`	It stores relevant client data	
`enabled`	Activation	"True"
✎ `CLIENT SELECTOR`	It selects clients based on criteria	
`enabled`	(De)Activation	"True", "False"
`selection_strategy`	Setup the selection strategy	"Resource-based"
`selection_criteria`	Setup the selection criteria	"CPU", "RAM"
`criteria_value`	Setup the threshold value	"1, n", "1, m"
✎ `CLIENT CLUSTER`	It groups clients according to similar data	
`enabled`	(De)Activation	"True", "False"
`clustering_strategy`	Setup the clustering strategy	"Data-based"
`clustering_criteria`	Setup the clustering criteria	"Data Distribution Type"
✎ `MESSAGE COMPRESSOR`	It compress Clients-Server exchanged data	
`enabled`	(De)Activation	"True","False"
✎ `MULTI-TASK MODEL TRAINER`	It trains multiple tasks simultaneously	
`enabled`	(De)Activation	"True","False"
`M1`	Setup the first model	"U-Net", ...
`M1_dataset`	Setup the dataset of the first model	"CIFAR-10", ...
`M2`	Setup the second model	"DenseU-Net", ...
`M2_dataset`	Setup the dataset of the second model	"NYUv2", ...
✎ `HETEROGENEOUS DATA HANDLER`	It generates synthetic data to balance dataset classes	
`enabled`	(De)Activation	"True","False"

Table 1: System Configuration Parameters Summary (✎ : Architectural Pattern).

allocation (i.e., `n_CPU`, `RAM`), the testing `dataset`, e.g., CIFAR-10, and the `data_Distribution` type that can be IID, non-IID, or random. The architectural patterns present the following settings. The `CLIENT REGISTRY` pattern is always enabled since it provides key client data on which other patterns rely on. The `CLIENT SELECTOR` pattern, when activated, requires the following settings: `selection_strategy` (e.g., Resource-based), `selection_criteria` (e.g., CPU), and `criteria_value` (e.g., >1). For instance, only clients with more than one CPU resource are selected, whereas all others are excluded. The `CLIENT CLUSTER` pattern, when enabled, necessitates these settings: `clustering_strategy` (e.g., Data-based), and `clustering_criteria` (e.g., Data Distribution Type). For example, clients are clustered via their data type. The `MESSAGE COMPRESSOR` pattern, when activated, reduces communication costs by compressing data exchanged between clients and server. The `MULTI-TASK MODEL TRAINER` pattern, when enabled, needs the specification of both the model architectures `M1` and `M2` (e.g., U-Net) and the corresponding datasets `M1_dataset` and `M2_dataset` (e.g., CIFAR-10). For instance, two datasets are used for training, both in isolation than in combination. The `HETEROGENEOUS DATA HANDLER` pattern, when activated, generates synthetic data that balance the representativeness of classes in the datasets. We recall that these parameters are set by users and they feed the `JSON` configuration file, see a concrete example in Listing 1. This way, users can download the entire configuration as a `JSON` file, that can later be uploaded when creating a new FL project, thereby enabling easy reuse of previously defined setups.

The ③ *Simulation* phase allows to run the simulation and collect results. AP4Fed embeds a prompt that provides an overview of the simulation, detailing aspects such as the training time or other information on implemented patterns, as shown in Figure 8a. This offers a detailed explanation of the current state of the simulation, aiding the reasoning of the ongoing process. Upon completion, the tool generates a set of summary plots, as shown in Figure 8b, and it offers the option to download a complete report of the simulation in a `CSV` format.

```
{
  SYSTEM_PARAMETERS: [
    {
      simulation_Type: "Docker",
      num_Rounds: 100,
      nC: 10,
    },
  ]
  CLIENT_DETAILS: [
    {
```

```
      client_id: 1,
      n_CPU: 2,
      RAM: 2,
      dataset: "FMNIST",
      data_Distribution: "IID"
    },
    {
      client_id: 2,
      n_CPU: 1,
      RAM: 2,
      dataset: "CIFAR-10",
      data_Distribution: "non-IID"
    }
    ...
  ],
  ARCHITECTURAL_PATTERNS: [
    client_registry: {
        enabled: "true",
    },
    client_selector: {
        enabled: "true",
        selection_strategy: "Resource-based",
        selection_criteria: "CPU",
        criteria_value: ">1"
    },
    client_cluster: {
        enabled: "true",
        clustering_strategy: "Data-based",
        clustering_criteria: "IID"
    },
    message_compressor: {
        enabled: "true"
    },
    multi_task_model_trainer: {
        M1: "U-Net",
        M1_dataset: "CIFAR-10",
        M2: "DenseU-Net",
        M2_dataset: "NYUv2"
    },
    heterogeneous_data_handler: {
        enabled: "true"
    }
  ]
}
```

Listing 1: Example of a JSON Configuration File for AP4Fed.

Results. Table 2 provides the description of the implemented evaluation metrics, building on approaches proposed in the literature [27, 56, 30]. Those metrics are selected to capture system performance and predictive performance variations, thus providing insights for different architectural choices. System performance metrics are collected at different granularity levels. The *training time* and the *communication time* are calculated for each client. The

(a) Prompt of the Simulation.

(b) Dashboard of the Simulation.

Figure 8: AP4Fed: Simulation GUI.

Target	Parameter	Description
System Performance	`Training Time`	Time Spent on Local Training
	`Communication Time`	Time Spent on Client-Server Communication
	`Total Round Time`	Total Time Spent for each FL Round
	`CPU Utilization`	Percentage of Time the CPU is Utilized
	`RAM Utilization`	Percentage of Time the RAM is Utilized
Predictive Performance	`Validation Loss`	Global Model's Error on Validation Data
	`Validation Accuracy`	Correct Predictions on Validation Data
	`F1 Score`	F1 Score of the Global Model
	`MAE`	Mean Absolute Error

Table 2: AP4Fed Evaluation Metrics.

total round time instead represents the time required to complete one round considering all the participating clients. Regarding resource utilization, the tool monitors two key metrics for federated learning [61, 1]: the Central Processing Unit (CPU) usage and Random Access Memory (RAM) consumption. We also evaluate the predictive performance metrics, following standard advices from the literature [27, 30, 56]. For classification problems, we derive the F1 score to balance precision and recall, while for regression tasks, we employ the Mean Absolute Error (MAE) to quantify the prediction accuracy.

Discussion. The outcome of our methodology consists of a report that captures a set of evaluation metrics from simulations. The produced report is aimed to collect knowledge on the FL system under analysis, along with information on the different architectural alternatives and their quantitative impact. We think this knowledge may represent a precious support for software architects that are in charge of comparing different design alternatives,

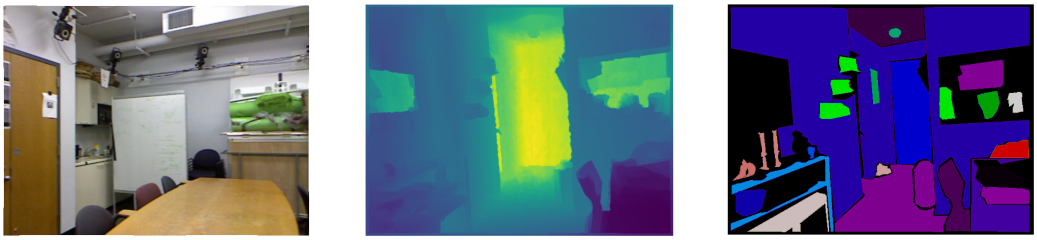
and deriving insights for improving the system performance and the predictive performance. This way, our quantitative evaluation can be used by software architects to take informed architectural decisions.

4. Experiments

4.1. Experiments Configuration

In this subsection, we describe the experimental setup used to evaluate the proposed architectural patterns, including the Heterogeneous Data Handler Pattern, the Multi-task Model Trainer one, and the use of combined patterns. We outline the system configuration, datasets, and the structure of the global models employed in our analysis. The goal is to assess the impact of these design alternatives on system performance and predictive performance.

Subject Systems. To fairly evaluate experiment output metrics, we report mean results obtained over 10 iterations, along with a 99% confidence interval, represented using lines and shaded areas in the plots. Each experiment is conducted using different ML models tailored to the specific ML task under evaluation. A Convolutional Neural Network (CNN) is employed as the global model for image classification tasks. For semantic segmentation, a U-Net architecture was utilized, leveraging its well-established effectiveness in capturing fine-grained spatial details. In the case of depth estimation, the DenseU-Net [30] is adopted to exploit feature reuse and enhance depth prediction accuracy. For training global models, we use the CIFAR-10 [32] and NYUv2 [47] datasets. The CIFAR-10 dataset consists of $60,000$ 32×32 color images, with $50,000$ images for training and $10,000$ for testing, divided into 10 distinct classes. Each class includes images of objects or animals (e.g., trucks, dogs), providing a challenging task for the CNN to accurately classify test images into one of the 10 categories. The NYUv2 dataset consists of $1,449$ 480×640 different images captured from a variety of indoor environments. As depicted in Figure 9, the dataset comprises three types of data: $1,449$ RGB images, $1,449$ depth maps, and $1,449$ `.npy` files. The RGB image in Figure 9a capture the visual details of a scene, while the corresponding depth map in Figure 9b reveal spatial information by encoding the distance of surfaces from the camera perspective. The `.npy` file stores 2D arrays, with each value representing a specific object class (e.g., 0 for background, 1 for table, etc.) that corresponds to a pixel in the RGB image. This information is visually represented in Figure 9c.



(a) RGB Image. (b) Depth Map. (c) Labeled Image.

Figure 9: Random Samples of the NYUv2 Datasets.

Hardware Setup. Experiments are conducted using a workstation machine with an `Intel Xeon W5-2445` featuring a `20 Core CPU @3.1GHz` and `64GB` of RAM alongside an `NVIDIA RTX A4500` GPU. Note that, AP4FED first checks if a Compute Unified Device Architecture (CUDA) GPU [36] is available, and if not, it automatically uses the CPU. Using Docker Compose, resources such as processing units (i.e., CPU cores) and RAM can be manually allocated to each container, enabling flexible experiment setups tailored to specific needs. This setup enables the emulation of heterogeneous clients, incorporating variations in the dataset distribution and clients specifications, to evaluate the impact of different configurations on the system performance and the predictive performance. Notably, the maximum number of concurrently running containers is constrained by the host machine's capacity (i.e., 10 cores) to avoid CPU overcommitment. This limitation is essential, as surpassing the processing limits of the host can result in resource contention, potentially invalidating the replicability and validity of experimental results [16].

4.2. Quantitative Evaluation of Architectural Patterns

In this subsection, we present the configurations and results of the experimental analysis conducted to evaluate the two newly implemented architectural patterns. Then, we report the experimental results from combining these architectural alternatives together in two distinct configurations to assess their impact on system and predictive performance.

4.2.1. Analysis of the Heterogeneous Data Handler Pattern

Table 3 presents input parameters used for the Heterogeneous Data Handler Pattern experiment. The experiment is repeated for 10 iterations, and result metrics are then averaged. The setup involves 10 rounds of FL with 1 server and 8 clients with 1 CPU and 2GB of RAM. To quantitatively

Parameter	Value
simulation_Type	Docker
num_Rounds	10
nC	8
n_CPU	1
RAM	2GB
dataset	CIFAR-10
Training Samples	$50,000 \rightarrow 25,000$
Data Handler Technique	Data Augmentation
Data Augmentation Strategy	GAN
GAN Library	*PyTorchGAN* [49]

Table 3: Input Parameters for Heterogeneous Data Handler Experiments.

evaluate this pattern, we use the Dirichlet distribution [69] to partition non-IID datasets for some clients[2]. This method, widely used in FL experiments [38, 68], allows for emulating varied data distributions, replicating the data heterogeneity commonly observed in real-world scenarios [29]. For completeness, we report the Dirichlet distribution formula:

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

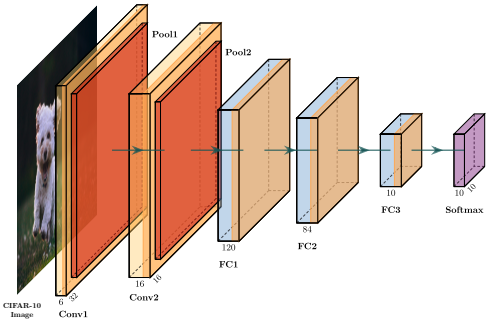
The Dirichlet distribution's probability density function, $p(x_1, x_2, \ldots, x_k)$ is normalized by the beta function $B(\alpha)$, ensuring the total probability equals one. The parameters $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k)$ control the level of classes imbalance, while the term $\prod_{i=1}^{k} x_i^{\alpha_i - 1}$ reflects the non-uniform distribution across variables. Here α defines the degree of unbalance, allowing adjustments to test different levels of data non-uniformity in the partitioning. To accommodate the CIFAR-10 dataset's limit of $5,000$ samples per class (across 10 classes), the number of training examples is reduced from $50,000$ to $25,000$ enabling a different allocation of non-IID data. This way, we assign a different number of samples across classes to each client while ensuring that the total number of samples remains equal across all clients, allowing for comparable experiment configurations. For instance, in IID clients, each class in the CIFAR-10 dataset has $2,500$ training examples, while non-IID clients exhibit imbalanced distributions, such as 500 examples for *cat* and $4,500$ for *truck*, ensuring the total number of training examples remains consistent across

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

clusters. The characteristics of the global model used in the experiment are reported in Figure 10. The CNN architecture, described in Figure 10a, consists of two convolutional layers (*Conv1* and *Conv2*) with ReLU activations, followed by max-pooling (Pool1 and Pool2) layers. *Conv1* has 6 filters (55 kernel), and *Conv2* has 16 filters (55 kernel). The model then transitions to three fully connected (FC) layers: *FC1* (120 units), *FC2* (84 units), and a *FC3* (10 units). Training uses a batch size of 32, a learning rate of 0.001, and SGD with 0.9 momentum. The graphical representation of the model is displayed in Figure 10b.

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
	Pool1: Max pooling, $2x2$ kernel
	Conv2: 16 filters, $5x5$ kernel
	Pool2: Max pooling, $2x2$ kernel
	FC1: 120 units
	FC2: 84 units
	FC3: 10 units
`Batch Size`	32
`Learning Rate`	0.001
`Optimizer`	SGD (momentum = 0.9)

(a) Global Model Parameters.



(b) Global Model Architecture.

Figure 10: Overview of the CNN Used in the Experiment.

The Heterogeneous Data Handler architectural pattern is implemented following the data augmentation strategy [41] based on Generative Adversarial Networks (GANs). GANs, introduced by Goodfellow et al. [24], operate through a competitive process between two neural networks: a *generator*, which creates synthetic data samples mimicking real data, and a *discriminator*, which aims to distinguish between generated and real samples. Figure 11 depicts the general use of a GAN.

Through adversarial training, the generator creates synthetic data from the real dataset while the discriminator evaluates authenticity. The GAN is considered well-trained when the discriminator can no longer differentiate between real and synthetic data, enabling high-quality synthetic data generation [24]. For instance, considering a facial images dataset, a GAN can produce variations by altering features like hair color or expression while maintaining core identity traits. This capability makes GANs especially use-

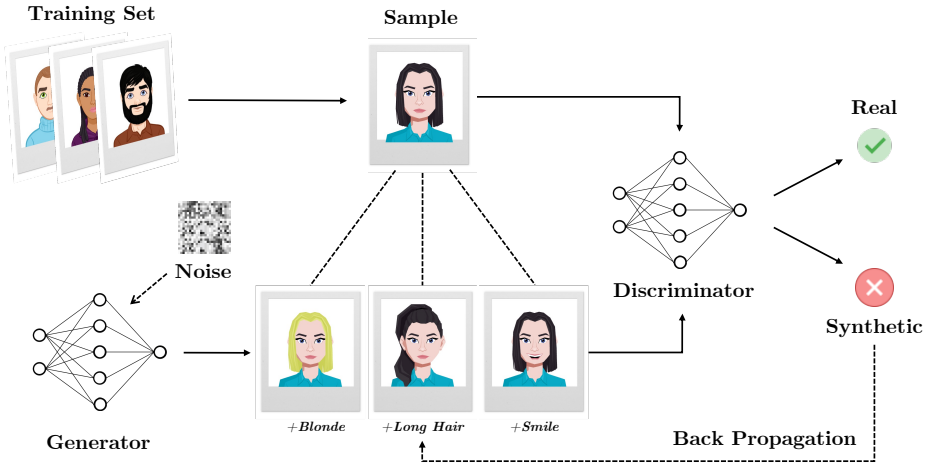


Figure 11: GAN Applications.

ful for addressing dataset imbalances, as they can generate high-quality examples for underrepresented classes. In FL contexts, GANs are recognized as an effective solution for non-IID data challenges, generating additional samples to reduce disparities among clients' local datasets [26, 2]. The effectiveness of GANs for data augmentation is well-documented [2], with successful implementations across various domains including image generation [13], image-to-image translation [75], and Natural Language Processing [71].

As reported in Table 4, we evaluate the Heterogeneous Data Handler pattern through four different experiment configurations labeled A, B, C, and D. In each configuration, we consider 8 Clients and 1 Server. In Config. A all clients use datasets with an IID data distribution, while in Config. B, clients are configured with datasets following a non-IID data distribution. In Config. C 4 clients use IID datasets and the other 4 use non-IID datasets. Similarly, Config. D also consists of 4 clients with IID datasets and 4 with non-IID datasets. However, in Config. D, the Heterogeneous Data Handler pattern is applied to the set of non-IID clients, adjusting their data distribution to closely resemble that of IID clients. Figure 12 highlights the impact of the *Heterogeneous Data Handler* pattern in effectively augmenting the non-IID clients datasets, showing the transition of the class distribution from a non-IID to an IID configuration.

Figure 13 shows the results of this experiment. Figure 13a depicts the F1 Score of each global model across the FL rounds. Config. A and Config. D,

Table 4: Experiment Configurations for Multi-Task Model Trainer.

	Config. A	**Config.** B	**Config.** C	**Config.** D
✎ `Heterogeneous Data Handler`	✗	✗	✗	✓
no. of IID Clients	8	-	4	$4 \rightarrow 8$
no. of non-IID Clients	-	8	4	$4 \rightarrow 0$
`Total Clients`	8	8	8	8

✗: *Without Heterogeneous Data Handler pattern;* ✓: *With Heterogeneous Data Handler pattern.*

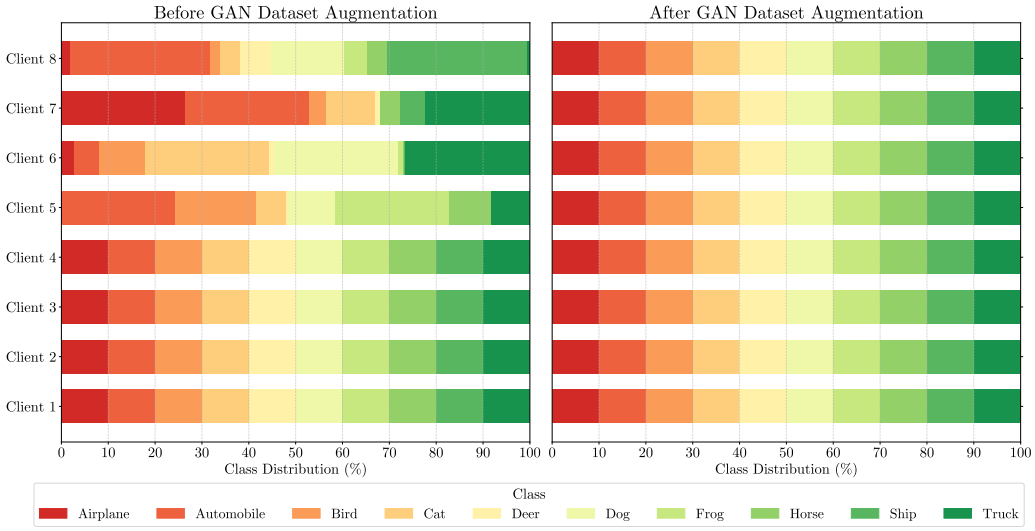
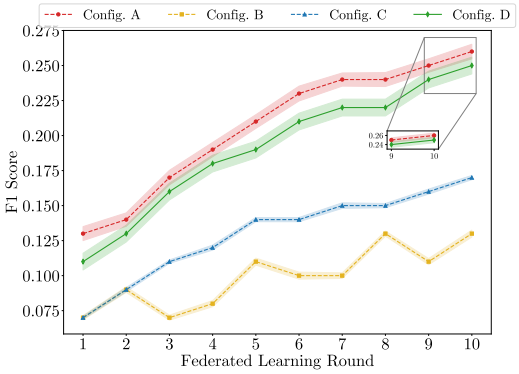
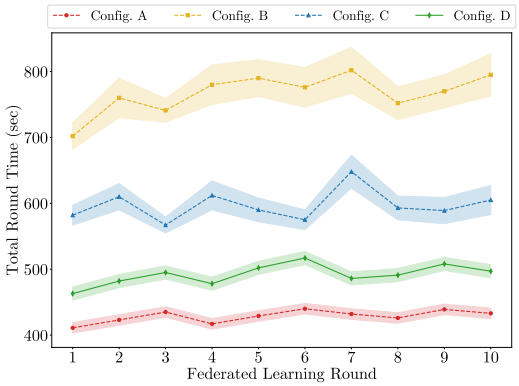


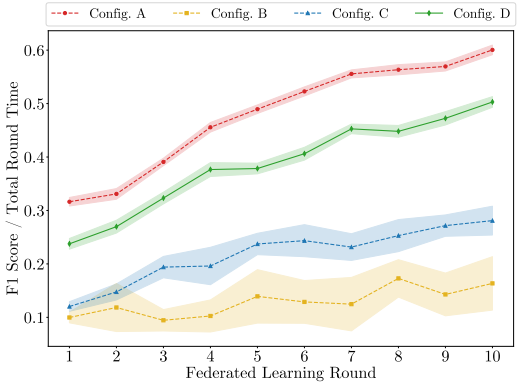
Figure 12: Client Dataset Partition Before and After applying GAN in Config. D.



(a) F1 Score per FL Round.



(b) Total Time per FL Round.



(c) F1 Score over Total Round Time.

Figure 13: Analysis of the Heterogeneous Data Handler Pattern.

which both perform the FL process without clients using non-IID datasets, exhibit similar trends. However, Config. A shows faster convergence, reaching a higher F1 Score earlier in the process. Config. B and Config. C, which include clients with non-IID datasets, show notably lower F1 Scores, indicating reduced model accuracy and slower convergence. Among these, Config. B, composed exclusively of clients with non-IID datasets, performs significantly worse. Notably, its F1 Score even declines during some rounds, dropping from 0.13 to 0.11 between rounds 8 and 9, highlighting the negative impact of non-IID data on the global model accuracy. Figure 13b presents the average total round time during FL rounds. The configurations involving clients with non-IID datasets (i.e., Config. B and Config. C) show longer and less stable average round times, as reflected by the wider confidence intervals derived from 10 experiment iterations. Notably, Config. B, which consists exclusively of clients with non-IID datasets, records the longest average round times, ranging between 700 and 800 seconds per FL round, making it the most time-consuming configuration analyzed. This highlights the significant impact of data non-IIDness on the duration of FL rounds. This aligns with practitioners' expectations, confirming that training times can increase when rounds include clients with non-IID data [38, 68]. Figure 13c depicts the ratio of F1 Score to the average total round time that we define as *efficiency*, thereby capturing the trade-off between predictive performance and system performance. Results indicate that configurations involving clients with non-IID datasets exhibit substantially lower efficiency, underscoring challenges associated with heterogeneous data distributions [41]. In contrast, configurations composed only by IID clients (Config. A) or employing the Heterogeneous Data Handler (Config. D) achieve higher efficiency, pointing out the positive impact of balanced data distributions and tailored handling strategies.

Architectural Implications. Software architects can consider this architectural pattern when dealing with a relevant number of clients that exhibit non-IID dataset characteristics. Experimental results demonstrate that applying GANs [24] technique to clients participating in the FL rounds improve model accuracy by balancing heterogeneous data distributions.

4.2.2. Analysis of the Multi-Task Model Trainer Pattern

Table 5 reports input parameters used for the Multi-Task Model Trainer Pattern experiment. Output metrics are evaluated by considering the average values over the 10 simulation iterations. The setup includes 10 FL rounds, with a single server and 4 clients collaborating for each training task, each

Parameter	Value
simulation_Type	Docker
num_Rounds	10
nC	4 for Each Task (T_1,T_2)
n_CPU	1
RAM	2 GB
dataset	NYUv2
T_1_TASK	Semantic Segmentation
T_2_TASK	Depth Estimation

Table 5: Input Parameters for Multi-Task Model Trainer Experiments.

equipped with 1 CPU and 2GB of RAM. To quantitatively evaluate this pattern, we use the NYUv2 dataset, focusing on two distinct ML tasks labeled T_1 and T_2. Figure 14 illustrates how these tasks are implemented in practice. Semantic segmentation (T_1) classifies each pixel into one of 40 predefined object classes (e.g., 0 for background, 1 for table, etc.), with distinct colors denoting different objects, as shown in Figure 14a. Similarly, depth estimation (T_2) predicts the distance of objects from the camera, generating a per-pixel depth map that enhances spatial perception, as shown in Figure 14b. Figure 15 presents the architectural characteristics of both global models used in our experiments. For T_1, we employ a U-Net architecture with an encoder-decoder structure. The parameters of the model are depicted in Table 15a and the graphical structure is shown in Figure 15b. The encoder features three blocks of double 3×3 convolutions with increasing filters (32, 64, 128), while the decoder mirrors this structure with decreasing filters (128, 64, 32). A 256-filter bottleneck connects these components, concluding with a 40 filters and a 1×1 convolution layer. For T_2, we implement a DenseU-Net architecture characterized by dense connectivity. The parameters of the model are depicted in Table 15c and the graphical structure is shown in Figure 15d. Its encoder combines three dense blocks (64, 128, 256 filters) with transition layers (128, 256, 512 filters), connected to the decoder through a 512-filter bottleneck. The decoder reduces filters from 512 to 128 via transition layers and dense blocks (256, 128, 64 filters), ending with a 1×1 convolution. Both models are trained with a batch size of 2 and learning rate of 0.001, using AdamW optimizer for U-Net and SGD with 0.9 momentum for DenseU-Net.

As reported in Table 6, we evaluate the Multi-Task Model Trainer Pattern considering 3 different configurations. In each configuration, we maintain a set of 4 clients for training the related ML task. Note that, Configurations M_1

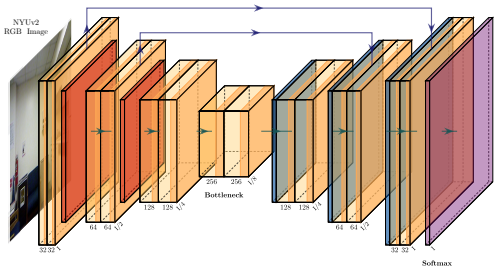
Figure 14: Sample of an RGB Image overlapped with Labeled and Depth Images.

(a) Overlap of RGB and Labeled Images ($\texttt{T}_1$).

(b) Overlap of RGB and Depth Images ($\texttt{T}_2$).

Parameter	Value
`Dataset`	NYUv2
`Training Samples`	1 150 RGB Images 1 150 Labeled Images
`Test Samples`	299 RGB Images 299 Labeled Images
`Model Type`	U-Net with Encoder and Decoder
`Model Structure`	*Encoder*: [32 → 64 → 128] filters × 2, 3×3 kernel *Bottleneck*: 256 filters × 2, 3×3 kernel *Decoder*: [128 → 64 → 32] filters x 2, 3×3 kernel *Final*: 40 filters, 1×1 kernel
`Batch Size`	2
`Learning Rate`	0.001
`Optimizer`	AdamW

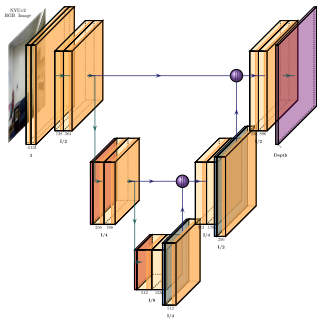
(a) Global Model Parameters of the U-Net ($\texttt{M}_1$).



(b) Architecture of the U-Net Model ($\texttt{M}_1$).

Parameter	Value
`Dataset`	NYUv2
`Training Samples`	1 150 RGB Images 1 150 Depth Maps
`Test Samples`	299 RGB Images 299 Depth Maps
`Model Type`	DenseU-Net with Encoder and Decoder
`Model Structure`	*Encoder*: Dense blocks [64 → 128 → 256] × 2 layers *Transition Down*: [128 → 256 → 512] filters *Bottleneck*: 512 × 2 layers *Transition Up*: [512 → 256 → 128] filters *Decoder*: Dense blocks [256 → 128 → 64] × 2 layers *Final*: 1 filter, 1×1 kernel
`Batch Size`	2
`Learning Rate`	0.001
`Optimizer`	SGD (momentum = 0.9)

(c) Global Model Parameters of the DenseU-Net ($\texttt{M}_2$).



(d) Architecture of the DenseU-Net Model ($\texttt{M}_2$).

Figure 15: Global Models Used in the Multi-Task Model Trainer Experiments.

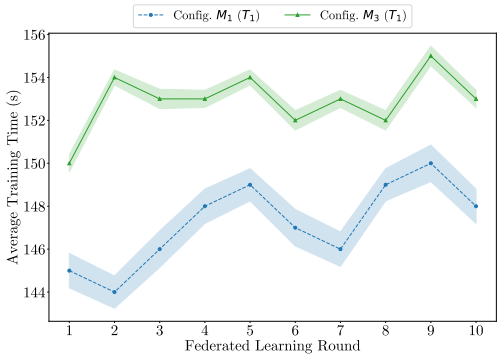
	Config. M_1	Config. M_2	Config. M_3
✎ Multi-Task Model Trainer	✗	✗	✓
no. of Clients for T_1	4 (2 IID, 2 non-IID)	-	4 (2 IID, 2 non-IID)
no. of Clients for T_2	-	4 (2 IID, 2 non-IID)	4 (2 IID, 2 non-IID)
Total Clients	$4 \rightarrow M_1$	$4 \rightarrow M_2$	$4+4 \rightarrow M_3$
Samples Distribution			
RGB Images	724	724	1 449
Labeled Images	-	724	1 449
Depth Maps	724	-	1 449

✗: Without Multi-Task Model Trainer pattern; ✓: With Multi-Task Model Trainer pattern.

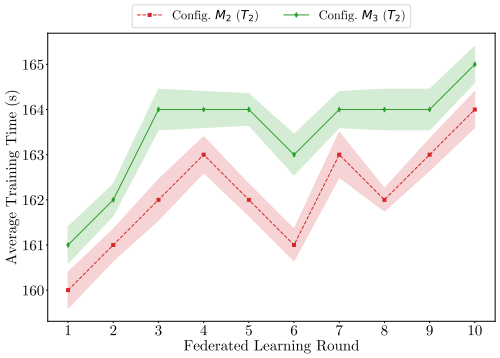
Table 6: Experiment Configurations for Multi-Task Model Trainer.

and M_2 each involve four clients collaborating to train their respective global models, M_1 and M_2. In the configuration M_3, every client participates in each task through sharing data, thereby enabling the simultaneous training of a unified global model M_3.

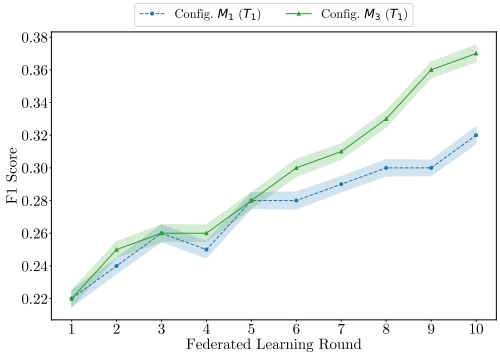
Figure 16 depicts the results of the experiments. For each configuration, we conduct a comparative analysis of the three models: M_1 and M_2 compared against M_3. When referring to M_3, we evaluate both T_1 and T_2 independently, as these represent distinct evaluation scenarios. Figure 16a shows the average client training time required to complete an FL round for model M_1 and M_3. The clients of M_1 participating in the training complete their FL rounds in 147 seconds on average, achieving a reduction of approximately 5.7 seconds per training round compared to M_3. In Figure 16b we compare the average training time of M_2 and M_3, and again we observe that M_2 completes training rounds more efficiently, with an average of 162.1 seconds compared to 163.5 seconds for M_3, achieving a reduction of approximately 1.4 seconds per training round. Considering the global model accuracy, Figure 16c depicts the F1 Score comparison between M_1 and M_3, where M_3 shows a consistent improvement in predictive performance, reaching higher F1 scores (0.35) compared to M_1 (0.32) by the final FL round. Figure 16d depicts the MAE comparison between M_2 and M_3, where M_3 demonstrates lower error rates, stabilizing around 0.16 compared to M_2 higher MAE of 0.18. Then, we consider accuracy and average training time to derive an efficiency metric. For M_1, we evaluate the ratio between F1 Score and average training time, multiplied by 100 to obtain a percentage efficiency score. For M_2, since both MAE and training time are metrics where lower values indicate better efficiency, we calculate their product. These metrics are defined as:



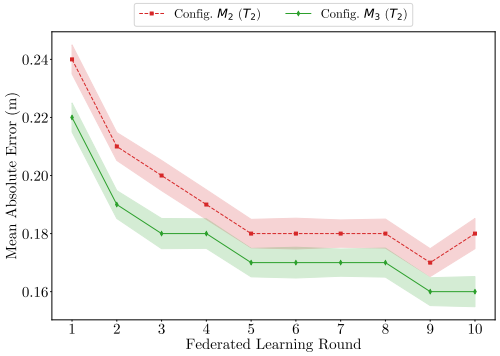
(a) Average Training Time per FL Round.



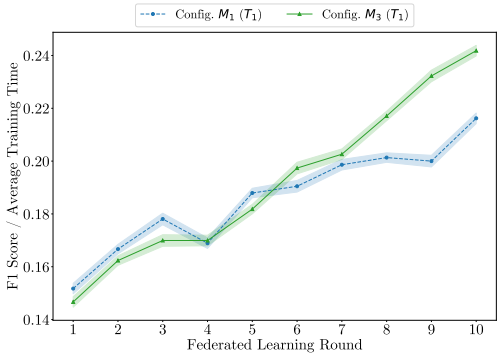
(b) Average Training Time per FL Round.



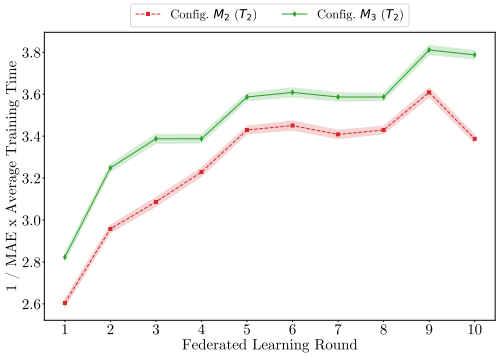
(c) F1 Score per FL Round.



(d) MAE per FL Round.



(e) Efficiency Comparison of $\texttt{M}_1$ and $\texttt{M}_3$.



(f) Efficiency Comparison of $\texttt{M}_2$ and $\texttt{M}_3$.

Figure 16: Analysis of the Multi-Task Model Trainer Pattern.

$$\text{Efficiency } \mathtt{M_1} = \left(\frac{\text{F1 Score}}{\text{Avg. Training Time}}\right) \times 100 \qquad \text{Efficiency } \mathtt{M_2} = \left(\frac{1}{\text{MAE} \times \text{Avg. Training Time}}\right) \times 100$$

The value of such ratios are shown in Figure 16e and Figure 16f, where we can notice that the best trade-off between accuracy and efficiency (i.e., in term of reducing training time) is achieved when the multi-Task Model Trainer pattern is implemented (Config. $\mathtt{M_3}$) in both ML tasks. This improvement is particularly evident in $\mathtt{M_2}$, where despite a minimal difference in training time, the pattern achieves better accuracy. For $\mathtt{M_1}$, while the difference in training time is slightly more evident, applying the pattern still shows higher accuracy throughout the FL rounds.

Figure 17 compares different configurations by reporting their average CPU utilization. The configuration with the Multi-Task Model Trainer pattern (Config. $\mathtt{M_3}$) shows a higher average CPU usage compared to $\mathtt{M_1}$ and $\mathtt{M_2}$, reflecting the increased computational demand due to the simultaneous management of multiple tasks. Specifically, $\mathtt{M_3}$ average CPU usage is approximately 44%, whereas the average utilization of $\mathtt{M_1}$ and $\mathtt{M_2}$ is lower, i.e., approximately 23% and 27%, respectively.

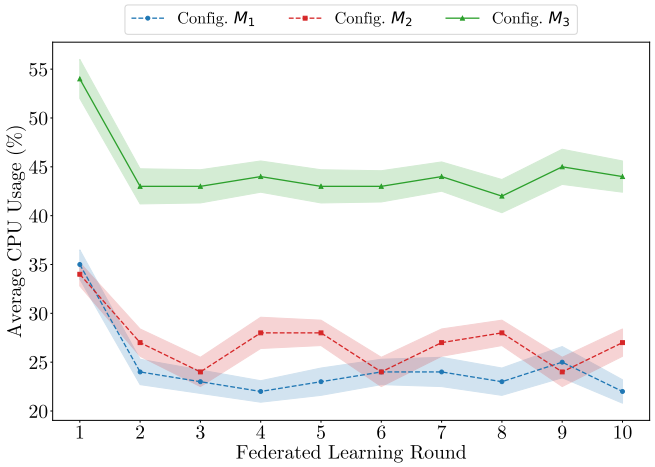


Figure 17: Average CPU Usage per FL Round.

Architectural Implications. Software architects may benefit from adopting this pattern when clients possess datasets containing features relevant to multiple ML tasks. The pattern key advantage emerges when clients participate in training multiple global models simultaneously, i.e., their local data can contribute to improving the accuracy of all involved models by leveraging shared data across ML tasks. However, the larger accuracy is achieved at the cost of increasing the utilization of hardware resources.

4.3. Combining Architectural Patterns

This section explores the combination of multiple patterns to study if this may yield additional benefits or unexpected drawbacks compared to their isolated use. This way, we investigate how these design alternatives interact when combined, thus supporting software architects to make informed decisions about system design and leverage potential synergies [41, 29]. In the following, we report the most interesting interactions we found, specifically the Multi-Task Model Trainer and Heterogeneous Data Handler patterns combined with two patterns from our previous work [18], i.e., the Client Selector and the Message Compressor.

4.3.1. Heterogeneous Data Handler and Message Compressor.

Table 7 reports our investigation on comparing another combination of architectural patterns, i.e., Heterogeneous Data Handler and Message Compressor: the baseline configuration (Config. *A*) implements Heterogeneous Data Handler with a mixed client pool (4 IID and 4 non-IID clients), while the combined configuration integrates both the Heterogeneous Data Handler and Message Compressor patterns. The latter combination integrates data augmentation through GAN-based generation with data compression techniques for client-server communication. The Heterogeneous Data Handler uses GANs to generate synthetic data to balance non-IID datasets, while the Message Compressor aims to reduce the communication overhead by compressing model parameters exchanged between clients and server.

Rationale. The combination is meant to reduce communication overhead while handling non-IID data distributions. However, this approach can introduce significant computational overhead, as the system needs to handle both the resource-intensive GAN operations and the additional compression/decompression processes. This is interesting since the increased total round time may lead to outweigh benefits.

Experiment. Figure 18 depicts the results of the experiment. Figure 18a shows the F1 Score progression across FL rounds. Config. *A* maintains slightly better accuracy throughout the FL rounds, reaching a final F1 Score of 0.25 compared to 0.24 of the combined configuration. This suggests that the compression process may introduce a small degradation in model accuracy [29]. Figure 18b presents the total round time per FL round. While both configurations follow similar patterns, the combined approach shows a higher total round time, averaging about 500 seconds per round versus 490 seconds for Config. *A*. Figure 18c depicts the ratio between F1 Score and total round

	Config. *A*	Config. Combined
✎ `Message Compressor`	✗	✓
`- Compression Library`	-	*zlib* [20]
✎ `Heterogeneous Data Handler`	✓	✓
no. of IID Clients	4	4
no. of non-IID Clients	4	4
`Total Clients`	8	8

✗*:Without Architectural Pattern;* ✓*:With Architectural Pattern.*

Table 7: Combined Configuration Comparison: Client Selector and Heterogeneous Data Handler.

time. Config. *A* shows better efficiency across all rounds, indicating that the additional computational overhead from combining the Message Compressor with the Heterogeneous Data Handler outweighs any potential benefits in communication efficiency. We think that this experimental result is relevant, since it makes evident that it is not always beneficial to combine architectural patterns, there exist some interactions that do not lead to advantages, thus confirming the usefulness of a tool-based approach that quantifies the impact of different design alternatives.

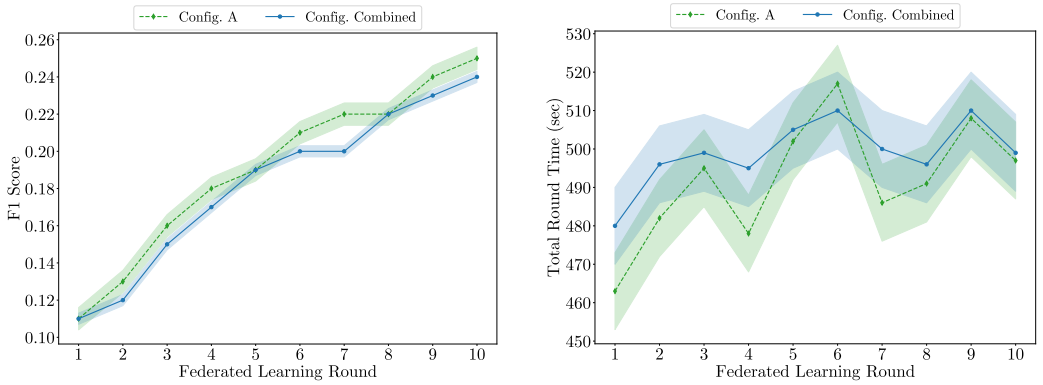
4.3.2. Client Selector and Multi-Task Model Trainer

Table 8 shows our intent to compare two distinct architectural choices: the baseline configuration (Config. $\mathtt{M_3}$) implements the Multi-Task Model Trainer pattern only with a mixed client pool (4 clients per task: 2 with IID and 2 with non-IID data). The combined configuration applies both the Client Selector and the Multi-Task Model Trainer, filtering participants to select only those with IID data distributions while preserving the concurrent multi-task training capability.

	Config. $\mathtt{M_3}$	Config. Combined
✎ `Client Selector`	✗	✓
`- Selection Strategy`	-	Data-Based
`- Selection Criteria`	-	IID Dataset
✎ `Multi-Task Model Trainer`	✓	✓
no. of Clients for $\mathtt{T_1}$	4 (2 IID, 2 non-IID)	$4 \rightarrow 2$ (2 IID)
no. of Clients for $\mathtt{T_2}$	4 (2 IID, 2 non-IID)	$4 \rightarrow 2$ (2 IID)
`Total Clients`	$4 + 4 \rightarrow \mathtt{M_3}$	$4 \rightarrow \mathtt{M_3}$

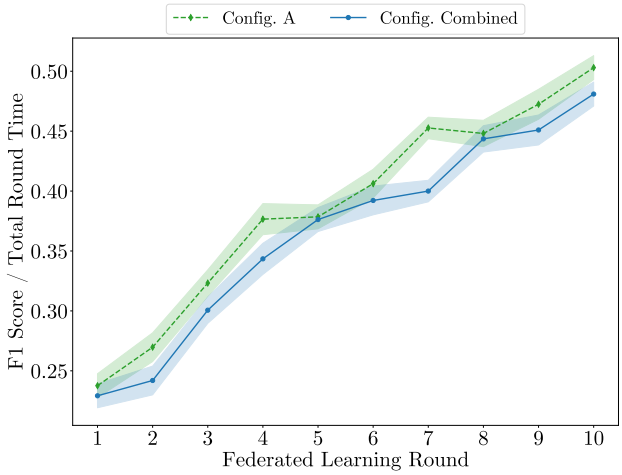
✗*:Without Architectural Pattern;* ✓*:With Architectural Pattern.*

Table 8: Combined Configuration Comparison: Client Selector and Multi-Task Model Trainer.



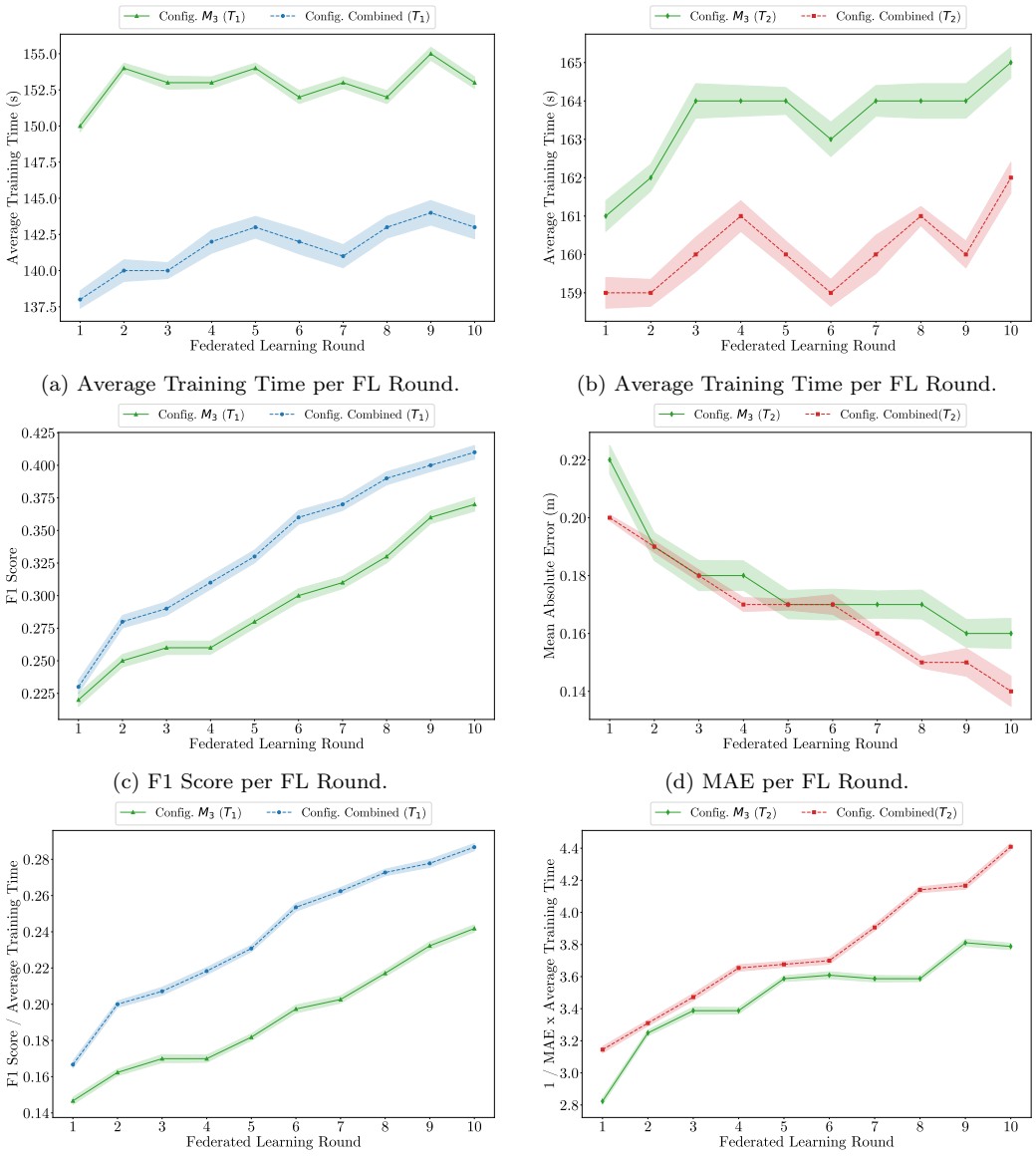
(a) F1 Score per FL Round.

(b) Total Time per FL Round.



(c) F1 Score over Total Round Time.

Figure 18: Analysis of the Heterogeneous Data Handler Pattern combined with the Message Compressor.



(a) Average Training Time per FL Round.

(b) Average Training Time per FL Round.

(c) F1 Score per FL Round.

(d) MAE per FL Round.

(e) Efficiency Comparison of $\mathtt{T}_1$.

(f) Efficiency Comparison of $\mathtt{T}_2$.

Figure 19: Analysis of the Multi-Task Model Trainer Pattern combined with the Client Selector.

Rationale. The combination of the Client Selector and the Multi-Task Model Trainer patterns aims to support multi-task learning by ensuring that selected clients hold well-distributed data across all classes. This combination assumes that clients with IID data provide more stable and reliable contributions across multiple learning tasks, potentially leading to better convergence in the multi-task training process.

Experiment. Figure 19 shows the results comparing two architectural choices: (*i*) the baseline configuration ($\mathtt{M_3}$) implementing only the Multi-Task Model Trainer with a heterogeneous client pool (2 IID and 2 non-IID), and (*ii*) a combined configuration that integrates the Client Selector pattern with the baseline, which systematically filters participants to retain only those with IID data characteristics. Figures 19a and 19b report the average training time per FL round for both tasks $\mathtt{T_1}$ and $\mathtt{T_2}$. The combined configuration with Client Selector shows consistently lower training times compared to the baseline Config. $\mathtt{M_3}$, particularly for $\mathtt{T_1}$ where the reduction is more pronounced. Figure 19c depicts the F1 Score evolution across FL rounds, with the combined configuration achieving higher model convergence. Figure 19d shows the MAE progression, where lower values indicate better predictive performance. Both configurations show decreasing error rates as training progresses, with the combined configuration showing better predictive performance in later rounds. Figures 19e and 19f present efficiency metrics combining accuracy (F1 Score for $\mathtt{T_1}$, MAE for $\mathtt{T_2}$) over training time. It is worth remarking that the combined configuration achieves a higher efficiency in both tasks, thus assessing the positive impact of combining these patterns.

5. Discussion

5.1. Analysis of Architectural Patterns

Heterogeneous Data Handler. It makes use of GAN-based data augmentation to enhance local client datasets, improving global model accuracy and reducing average round time [41]. Balancing datasets with high-quality synthetic samples mitigates the effects of non-IID data, reducing gradient variance and accelerating convergence [41]. In addition, as practitioners note [41, 18], training on non-IID datasets can significantly extend the training phase duration and overall FL round time. Our experiment confirms both these assumptions. Configurations with non-IID clients (Config. B and C) demonstrate reduced global model accuracy and slower convergence, characterized by longer and inconsistent round times. In contrast, the Heteroge-

neous Data Handler (Config. D) mitigates these challenges through GAN-based data augmentation, balancing local datasets and accelerating convergence [41]. However, this design alternative introduces substantial computational overhead due to the time required for training the GAN. The process of generating and validating synthetic data demands significant computational resources, while potentially revealing sensitive information about client data distributions that could compromise the client privacy.

Multi-Task Model Trainer. It provides the capability to train global models that can handle different yet related tasks simultaneously [41]. In our implementation, clients participate in both classification ($\mathtt{T_1}$) and regression ($\mathtt{T_2}$) tasks, demonstrating that shared data characteristics contribute to improved predictive performance for both global models while keeping training time reasonable. Although training global models separately may lead to a slightly shorter training time, employing the Multi-Task Model Trainer pattern results in higher accuracy for both models. This is due to the Multi-Task Learning approach that allows the model to learn shared representations across different but related tasks [41]. When data distributions among clients are skewed, a single global model struggles to capture unique data patterns, especially with non-IID data [38, 68]. By incorporating clients with similar data characteristics, the global model can learn from a broader dataset, providing more samples for underrepresented classes and mitigating class imbalance. However, the Multi-Task Model Trainer pattern leads to a higher consumption of hardware resources. As shown in Figure 17, the configuration where this pattern is enabled (Config. $\mathtt{M_3}$) exhibits significantly higher CPU usage (i.e., 44% on average) than other configurations without this patterns (i.e., $\mathtt{M_1}$ and $\mathtt{M_2}$, whose CPU utilization is 23% and 27% on average, respectively). This increase reflects the additional computational demand required to concurrently manage and aggregate training results from multiple tasks. Moreover, model portability is more complex, as each client may use a model tailored to its specific task. A robust and reliable implementation must carefully manage the aggregation of different model weights in the server [41].

5.2. Combining Multiple Architectural Patterns

Heterogeneous Data Handler and Message Compressor. The integration of the Heterogeneous Data Handler and the Message Compressor is initially conceived to improve system efficiency by enhancing data quality and reducing the communication time. However, from combining these patterns, we learn that the overall system efficiency may decrease. Experimental results

show lower efficiency compared to when the Heterogeneous Data Handler is implemented in isolation. The additional processing time from data compression and the decreasing global model accuracy appear to offset advantages gained through improved data quality. The system performance degradation observed when combining the Heterogeneous Data Handler with the Message Compressor can be attributed to the pipeline of sequential data transformation they introduce. GAN-based data augmentation operations, which are computationally intensive, may introduce significant system bottlenecks when followed by compression procedures. This experience points out the reason why the combination of design alternatives requires some cautious. The synthetic data generated to balance class distributions is subsequently compressed, potentially degrading the statistical properties established during augmentation. This double transformation introduces information loss that negatively impacts the model accuracy. Software architects should carefully evaluate combinations of data-processing patterns, as their cumulative effects can distort data characteristics, resulting in reduced model accuracy rather than improved system performance.

Client Selector and Multi-Task Model Trainer. Combining Client Selector with Multi-Task Model Trainer shows a successful synergy offering improved efficiency and enhanced model accuracy. By filtering out non-IID clients, the Client Selector reduces noise during gradient aggregation, leading to faster convergence, while the Multi-Task Model Trainer effectively leverages shared data across tasks. While using fewer total clients, the combined configuration achieves a slightly higher accuracy with significantly reduced training time. The impact of this combination stems from two complementary factors. First, the Client Selector eliminates clients with non-IID data, which typically have unbalanced datasets and slow down model convergence and learning. Second, the Multi-Task Model Trainer leverages shared data across tasks, thereby enriching the training set with additional data that may be missing in individual clients. This synergy not only reduces the computational overhead by limiting non-IID clients, it also enhances training effectiveness by ensuring a more balanced and comprehensive data distribution.

5.3. Integration with existing MLOps techniques

Machine Learning Operations (MLOps) is defined in the literature as a paradigm including a variegate set of aspects, such as best practices, con-

cepts, and development culture, that contribute to the implementation, monitoring, deployment, and scalability of machine learning products [31].

MLOps tools are reviewed by Recupito et al. [54] that highlight some important features of interest. For instance, our approach is aligned with the so-called predictive *performance monitoring* feature, since it raises the need to control the model's predictive performance. Two further features that align with our research are: (i) *open source*, which ensures the software code is publicly accessible for use, modification, and distribution; and (ii) *scalability*, which highlights the importance of monitoring the system performance to support reliable continuous delivery.

MLOps pipelines are reviewed by Eken et al. [21] who foresee automation as a crucial aspect in ML systems. Tasks such as data collection, model building, deployment, and monitoring should be automated to minimize the need of human intervention. Pipeline management is recognized as a complex process, involving multiple triggers to initiate task execution. For example, training may be triggered when the predictive performance of a model declines [21]. This highlights the importance of generating reports (as those provided in our approach) that offer insights on the status of the ML product throughout its development and operation.

The MLOps lifecycle consists of managing multiple versions of experiments, datasets, models, and their associated metadata, and optimizing workflows across these lifecycle stages is crucial [21]. Our research methodology supports this optimization by leveraging architectural patterns that facilitate the generation of multiple experiment versions. Resulting quantitative metrics are then used to compare a wide range of design alternatives.

Our approach can be integrated with existing MLOps techniques (e.g., Kubeflow [10], Amazon SageMaker [39], or Metaflow [8]) leveraging the simulation reports produced by AP4Fed. Our quantitative evaluation includes metrics that assess both system and predictive performance, making them valuable for analyzing ML products within existing tools and pipelines.

5.4. Threats to Validity

Besides inheriting limitations of architectural patterns and system performance analysis [59, 74], our approach presents the following threats to validity [67].

External validity. The generalization of our findings to different scenarios cannot be ensured due to the constraints imposed by our experimental setup. For instance, the available hardware resources, limited by the number of

physical processors, restrict the scale of client devices in our experiments. As a result, the quantitative analysis primarily reflects the behavior of a small client population, which may not capture complexities and interactions of large-scale FL systems. Despite these limitations, experimental outcomes offer meaningful insights into the potential applicability and scalability of the proposed methodology. In addition, AP4Fed enables the execution of large-scale experiments, offering high flexibility to the specific technical requirements of software architects aiming to emulate real FL settings.

Internal validity. Settings and parameters adopted for our analysis may pose certain risks. First, the proposed evaluation metrics capture a limited set of FL characteristics, and the obtained results reflect that selection. As future work, we plan to extend the evaluation and include additional metrics to explore further aspects of FL systems. Second, our experiments use only image-based datasets, excluding tabular data. We acknowledge that this choice limits the applicability of our approach, as it remains unclear if implemented patterns perform similarly with other type of data. This also leads to questioning our findings, since they might be influenced by the specific pattern implementation. As future work, we aim to investigate data of different nature to study the flexibility of AP4Fed. Third, numerical values assigned to input parameters are selected to point out quantitative differences across architectural alternatives. For example, we adjust the data partitioning strategy (e.g., IID versus non-IID) among clients to evaluate the impact of the heterogeneous data handling mechanism. To mitigate misleading influences, we ensure consistent input settings across experiments for each pattern, allowing observed results to be reliably linked to specific architectural decisions. However, determining accurate numerical values for input parameters is an ever existing challenge in *system* performance [12] and other settings could be further investigated. Accordingly, we provide publicly available instructions for setting up experiments and testing different configurations [19]. Additionally, AP4Fed empowers software architects by offering a user-friendly interface to modify parameters and tailor experiments to their specific requirements, thus enabling customization in running the system performance analysis.

Construct validity, i.e., the statistical validity of experimental results is monitored by repeating all the experimental configurations 10 times. We show output metrics average and their 99% confidence intervals, thus to assess the accuracy of the presented numerical outcomes.

6. Related work

Architectural Patterns. Our work builds on extensive research emphasizing the significance of adopting architectural patterns in software system design [43, 6, 63, 46]. In recent years, the software architecture community has increasingly enforce the adoption of patterns in ML systems [66, 11, 65, 48]. Washizaki et al. [66] conducted a study that classifies architectural patterns specifically designed for ML systems, offering developers a structured framework to identify and implement appropriate solutions. Their findings highlight the importance of systematically evaluating the system performance and predictive performance of these patterns, which represents the core focus of our research. Ntentos et al. [48] present a qualitative study aimed at supporting architectural decision-making for training strategies in reinforcement learning systems. Practitioners' knowledge is exploited to derive established patterns and best practices, and ultimately formalize these insights into a reusable architectural design decision model. Warnett et al. [64] investigate the impact of architectural design decisions within ML operations (MLOps), and a set of metrics is introduced to quantify the automation in such a context. Interestingly, the authors point out that the metrics are valuable to quantify the quality of MLOps systems whilst ensuring compliance with non-functional requirements, e.g., system performance as proposed in this paper. Leest et al. [34] and Takeuchi et al. [62] propose well-defined approaches to tackle practical design challenges in ML systems, further emphasizing the need for methodologies that bridge architectural design with real-world application requirements.

Federated Learning. FL presents notable benefits over centralized ML, particularly in terms of system efficiency and privacy [70, 35]. However, this distributed approach also introduces new architectural challenges, especially in managing the complex interactions between servers and clients. To address these issues, recent studies by Lo et al. [40] and Rajasekaran et al. [53] have proposed a suite of architectural patterns that provide structured solutions to these challenges. Recent studies have begun integrating these architectural patterns highlighting the importance of structured approaches to enhance system robustness. Ma et al. [42] introduce the Client Selector pattern by implementing an algorithm that prioritizes clients based on computational capacity and network conditions, aiming to mitigate delays caused by slower clients while preserving model accuracy. Pavlidis et al. [51] 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. Chahoud et al. [15] propose a mechanism that facilitates the establishment of a trust relationship between the server and the pool of eligible clients, while concurrently strengthening defenses against unauthorized data access or manipulation. Fan et al. [23] explore Heterogeneous Data handling techniques by implementing GANs within a FL system, thus addressing challenges like privacy and data heterogeneity. Unlike standard FL, this method synchronizes both weights and the interaction between generators and discriminators, enforcing confidentiality and preserving data quality.

To the best of our knowledge, the closest related methodologies are discussed hereafter. Lai et al. [33] introduce a FL benchmark engine. Unlike our research, their work focuses on creating a standardized benchmarking suite for FL systems without exploring the impact of architectural alternatives on system performance through empirical evaluation. Casalicchio et al. [14] introduce a workbench platform designed to evaluate the system performance of Federated Learning systems, incorporating patterns described in [41]. While this work was initially considered for comparison, the absence of reproducible experimental instructions and the lack of accessibility to the proposed tool prevent any meaningful correlation with our approach.

In summary, existing studies primarily focus on introducing architectural patterns for FL systems, emphasizing only their potential advantages. Our work advances the state-of-the-art by proposing a methodology that not only evaluates the impact of these architectural alternatives on system performance and predictive performance, but also provides insights about their combined usage. To this end, we introduce AP4Fed, a benchmark framework that enable software architects to design and analyze FL systems adopting one or multiple architectural patterns. Its strengths include a modular design that simplifies the integration of architectural patterns and a GUI that makes building configurations straightforward, even for complex setups.

7. Conclusion

This paper presents AP4Fed, a publicly available benchmark framework to quantitatively evaluate FL systems through the integration of six architectural patterns. AP4Fed provides a versatile platform that enables software architects to emulate realistic FL systems and assess the impact of architectural alternatives derived from the literature [41] and considered relevant

for system performance and predictive performance. Our findings suggest that architectural patterns are promising solutions for addressing FL design challenges. However, their adoption involves trade-offs, as the predictive performance improvements come with an additional overhead that varies depending on the specific pattern. For instance, the *Heterogeneous Data Handler* reduces training time and increases model accuracy by utilizing GAN-based data augmentation to balance non-IID client datasets, thereby minimizing gradient variance and enhancing global model convergence. The *Multi-Task Model Trainer* pattern improves global model accuracy by leveraging shared data characteristics across separated but related tasks. In addition, the combination of multiple architectural patterns requires thorough analysis, as they may interact in unexpected ways, potentially introducing system performance drawbacks in the FL system. Analyzing these interactions helps the identification of trade-offs and optimization of overall FL system performance and predictive performance.

In future work, a primary activity will be devoted to evaluate the usefulness of the proposed framework, thus understanding at which extent it supports software architects. To this end, we plan to define a user study by which software architects can express their opinions. This is of key relevance, since we can use the feedback to further improve our framework. Besides, we plan to extend AP4Fed by considering further evaluation metrics, e.g., software/hardware failures for system reliability. As another example, security flaws may be detected in data since clients may be exposed to attacks, and it may be relevant to detect data poisoning. Furthermore, we plan to test additional datasets, mainly exploring tabular formats, since they may contribute to different findings.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback, which helped us improve the quality of the manuscript. 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] Alawadi, S., Ait-Mlouk, A., Toor, S., Hellander, A., 2024. Toward efficient resource utilization at edge nodes in federated learning. Progress in Artificial Intelligence 13, 101–117.

[2] Antoniou, A., Storkey, A.J., Edwards, H., 2018. Augmenting Image Classifiers Using Data Augmentation Generative Adversarial Networks, in: International Conference on Artificial Neural Networks (ICANN), Springer. pp. 594–603.

[3] Augenstein, S., McMahan, H.B., Ramage, D., Ramaswamy, S., Kairouz, P., Chen, M., Mathews, R., y Arcas, B.A., 2020. Generative Models for Effective ML on Private, Decentralized Datasets, in: International Conference on Learning Representations (ICLR).

[4] Banabilah, S., Aloqaily, M., Alsayed, E., Malik, N., Jararweh, Y., 2022. Federated Learning Review: Fundamentals, Enabling Technologies, and Future Applications. Information Processing & Management 59, 103061.

[5] Baresi, L., Quattrocchi, G., Rasi, N., 2023. Open Challenges in Federated Machine Learning. Internet Computing 27, 20–27.

[6] Bass, L., Clements, P., Kazman, R., 2012. Software Architecture in Practice. 3rd ed., Addison-Wesley Professional.

[7] Beck, K., 1987. Using pattern languages for object-oriented programs, in: International workshop on the Specification and Design for Object-Oriented Programming (OOPSLA).

[8] Berberi, L., Kozlov, V., Nguyen, G., Sáinz-Pardo Díaz, J., Calatrava, A., Moltó, G., Tran, V., López García, Á., 2025. Machine learning operations landscape: platforms and tools. Artificial Intelligence Review 58, 167.

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

[10] Bisong, E., 2019. Kubeflow and kubeflow pipelines, in: Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners. Springer, pp. 671–685.

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

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

[13] Brock, A., Donahue, J., Simonyan, K., 2019. Large Scale GAN Training for High Fidelity Natural Image Synthesis, in: International Conference on Learning Representations, (ICLR).

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

[15] Chahoud, M., Mourad, A., Otrok, H., Bentahar, J., Guizani, M., 2025. Trust Driven On-Demand Scheme for Client Deployment in Federated Learning. Information Processing & Management 62, 103991.

[16] Cohen, M.C., Keller, P.W., Mirrokni, V.S., Zadimoghaddam, M., 2019. Overcommitment in Cloud Services: Bin Packing with Chance Constraints. Management Science 65, 3255–3271.

[17] Compagnucci, I., Corradini, F., Fornari, F., Re, B., 2024. A Study on the Usage of the BPMN Notation for Designing Process Collaboration, Choreography, and Conversation Models. Business & Information Systems Engineering 66, 43–66.

[18] Compagnucci, I., Pinciroli, R., Trubiani, C., 2025. Performance Analysis of Architectural Patterns for Federated Learning Systems, in: 2025 IEEE 22nd International Conference on Software Architecture (ICSA), pp. 289–300.

[19] Compagnucci, I. and Pinciroli, R. and Trubiani, C., 2025. Open Science Artifact: Experimenting Architectural Patterns in Federated Learning Systems. doi:`https://doi.org/10.5281/zenodo.16638879`.

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

[21] Eken, B., Pallewatta, S., Tran, N., Tosun, A., Babar, M.A., 2024. A multivocal review of mlops practices, challenges and open issues. ACM Computing Surveys .

[22] European Commission, . European General Data Protection Regulation (GDPR). `https://gdpr-info.eu/`.

[23] Fan, C., Liu, P., 2020. Federated Generative Adversarial Learning, in: Pattern Recognition and Computer Vision, Springer. pp. 3–15.

[24] Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A.C., Bengio, Y., 2014. Generative adversarial nets, in: Conference on Neural Information Processing Systems, pp. 2672–2680.

[25] Hastie, T., Tibshirani, R., Friedman, J.H., 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics, Springer.

[26] Jeong, E., Oh, S., Kim, H., Park, J., Bennis, M., Kim, S., 2018. Communication-Efficient On-Device Machine Learning: Federated Distillation and Augmentation under Non-IID Private Data. CoRR abs/1811.11479.

[27] Jiang, Y., Wang, S., Valls, V., Ko, B.J., Lee, W.H., Leung, K.K., Tassiulas, L., 2022. Model Pruning Enables Efficient Federated Learning on Edge Devices. Transactions on Neural Networks and Learning Systems 34, 10374–10386.

[28] Jobin, A., Ienca, M., Vayena, E., 2019. The global landscape of AI ethics guidelines. Nat. Mach. Intell. 1, 389–399.

[29] Kairouz, P., et al., 2021. Advances and Open Problems in Federated Learning. Foundations and Trends in Machine Learning 14, 1–210.

[30] Khan, F., Salahuddin, S., Javidnia, H., 2020. Deep Learning-Based Monocular Depth Estimation Methods: A State-of-the-Art Review. Sensors 20.

[31] Kreuzberger, D., Kühl, N., Hirschl, S., 2023. Machine learning operations (mlops): Overview, definition, and architecture. IEEE Access 11, 31866–31879.

[32] Krizhevsky, A., Hinton, G., et al., 2009. Learning Multiple Layers of Features from Tiny Images. Technical Report.

[33] Lai, F., Dai, Y., Singapuram, S., Liu, J., Zhu, X., Madhyastha, H., Chowdhury, M., 2022. FedScale: Benchmarking Model and System Performance of Federated Learning at Scale, in: Proceedings of Machine Learning Research (PMLR), pp. 11814–11827.

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

[35] Li, L., Fan, Y., Tse, M., Lin, K.Y., 2020a. A Review of Applications in Federated Learning. Computers & Industrial Engineering 149, 106854.

[36] Li, M., et al., 2024. Deep Learning and Machine Learning with GPGPU and CUDA: Unlocking the Power of Parallel Computing. CoRR abs/2410.05686.

[37] Li, T., Sahu, A.K., Zaheer, M., Sanjabi, M., Talwalkar, A., Smith, V., 2020b. Federated Optimization in Heterogeneous Networks, in: Conference on Machine Learning and Systems, mlsys.org.

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

[39] Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., et al., 2020. Elastic machine learning algorithms in amazon sagemaker, in: International Conference on Management of Data (SIGMOD), pp. 731–737.

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

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

[42] Ma, C., Li, J., Ding, M., Wei, K., Chen, W., Poor, H.V., 2021. Federated Learning With Unreliable Clients: Performance Analysis and Mechanism Design. Internet Things Journal 8, 17308–17319.

[43] Martin, R.C., 2000. Design Principles and Design Patterns. Object Mentor 1, 597.

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

[45] McMahan, H.B., Moore, E., Ramage, D., y Arcas, B.A., 2016. Federated Learning of Deep Networks using Model Averaging. CoRR abs/1602.05629. URL: `http://arxiv.org/abs/1602.05629`.

[46] Meijer, W., Trubiani, C., Aleti, A., 2024. Experimental evaluation of architectural software performance design patterns in microservices. J. Syst. Softw. 218, 112183.

[47] Nathan Silberman, Derek Hoiem, P.K., Fergus, R., 2012. Indoor Segmentation and Support Inference from RGBD Images, in: European Conference on Computer Vision.

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

[49] Pal, A., Das, A., 2021. TorchGAN: A Flexible Framework for GAN Training and Evaluation. Journal of Open Source Software 6, 2606.

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

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

[52] Qi, P., Chiaro, D., Guzzo, A., Ianni, M., Fortino, G., Piccialli, F., 2024. Model aggregation techniques in federated learning: A comprehensive survey. Future Gener. Comput. Syst. 150, 272–293.

[53] Rajasekaran, V., Periyasamy, J., Brahmam, M., Baluswamy, B., 2024. Architectural Patterns for the Design of Federated Learning Systems. Model Optimization Methods for Efficient and Edge AI. pp. 223–239.

[54] Recupito, G., Pecorelli, F., Catolino, G., Moreschini, S., Di Nucci, D., Palomba, F., Tamburri, D.A., 2022. A multivocal literature review of mlops tools and features, in: Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 84–91.

[55] Richards, M., 2015. Software Architecture Patterns. volume 4.

[56] Ronneberger, O., Fischer, P., Brox, T., 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. CoRR abs/1505.04597.

[57] Sánchez, P.M.S., Celdrán, A.H., Xie, N., Bovet, G., Pérez, G.M., Stiller, B., 2024. FederatedTrust: A solution for trustworthy federated learning. Future Gener. Comput. Syst. 152, 83–98.

[58] Shah, S.M., Lau, V.K.N., 2023. Model Compression for Communication Efficient Federated Learning. Transactions on Neural Networks and Learning Systems 34, 5937–5951.

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

[60] Smith, V., Chiang, C., Sanjabi, M., Talwalkar, A., 2017. Federated Multi-Task Learning, in: Conference on Neural Information Processing Systems, pp. 4424–4434.

[61] Soudan, B., Abbas, S., Kubba, A., Abu Waraga, O., Abu Talib, M., Nasir, Q., 2025. Scalability and performance evaluation of federated

learning frameworks: a comparative analysis. International Journal of Machine Learning and Cybernetics 16, 3329–3343.

[62] Takeuchi, H., Doi, T., Washizaki, H., Okuda, S., Yoshioka, N., 2021. Enterprise Architecture based Representation of Architecture and Design Patterns for Machine Learning Systems, in: Enterprise Distributed Object Computing Workshop, (EDOC), pp. 245–250.

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

[64] Warnett, S.J., Ntentos, E., Zdun, U., 2025. A model-driven, metrics-based approach to assessing support for quality aspects in mlops system architectures. Journal of Systems and Software 220, 112257.

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

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

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

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

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

[70] Zhang, C., Xie, Y., Bai, H., Yu, B., Li, W., Gao, Y., 2021. A Survey on Federated Learning. Knowledge-Based System 216, 106775.

[71] Zhang, Y., Gan, Z., Fan, K., Chen, Z., Henao, R., Shen, D., Carin, L., 2017. Adversarial Feature Matching for Text Generation, in: International Conference on Machine Learning, (ICML), pp. 4006–4015.

[72] Zhang, Y., Yang, Q., 2022. A Survey on Multi-Task Learning. Transactions on Knowledge and Data Engineering 34, 5586–5609.

[73] Zhang, Y., Zeng, D., Luo, J., Fu, X., Chen, G., Xu, Z., King, I., 2024. A Survey of Trustworthy Federated Learning: Issues, Solutions, and Challenges. ACM Trans. Intell. Syst. Technol. 15, 112:1–112:47.

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

[75] Zhu, J., Park, T., Isola, P., Efros, A.A., 2017. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks, in: International Conference on Computer Vision, (ICCV), Computer Society. pp. 2242–2251.

[76] Zhu, X., Wang, J., Chen, W., Sato, K., 2023. Model Compression and Privacy Preserving Framework for Federated Learning. Future Generation Computer Systems 140, 376–389.