# Architectural Strategies for Federated Learning

**Priyanka Atwani**
priyankaatwani@live.com

July 16, 2017, 49 pages

| | |
|---|---|
| **Academic supervisor:** | Ana Oprescu, a.m.oprescu@uva.nl |
| **Daily supervisor:** | Leon Gommans, leon.gommans@klm.com |
| **Host organisation/Research group:** | AirFrance-KLM, https://www.airfranceklm.com |

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

# Abstract

To realize the potential benefits of data-driven research in industry, a crucial step is facilitating data exchange between various parties. A powerful illustrating example comes from the aviation industry, where predictive maintenance could benefit from more fleet data, however many challenges hinder effectively exchanging this data across airlines, such as protection of crew's rights. One promising solution to this problem is Federated Learning, where the learning occurs without data actually being exchanged.

This paper considers scalable and modular design approaches to implement Federated Learning (FL). We showcase how such approaches can be applied when considering predictive aircraft engine maintenance based on machine learning (ML). We address data quality, privacy concerns, and collaborative data sharing among airlines through FL workflows. Our system, built with Docker containerization and leveraging the FABRIC testbed infrastructure, effectively handles large-scale dynamic datasets while ensuring scalability, modularity, and maintainability. Key design strategies include data partitioning, update strategies, adaptive learning, and disagreement handling. Incorporating microservices architecture based patterns enhances scalability and efficiency, allowing independent service deployment, development, testing, and scaling. This integrated approach ensures secure data exchange and seamless integration with existing aviation data management systems.

# Contents

# Chapter 1

# Introduction

In an era where technological advancements are reshaping industries, the aviation sector stands at the forefront of innovation [1]. According to the European Union Aviation Safety Agency (EASA), AI-based predictive maintenance, fueled by an enormous amount of fleet data [2], allows for anticipating failures and providing preventive remedies [3]. However, effectively utilizing this data, particularly for predictive maintenance, poses significant challenges. Machine learning algorithms trained on historical data can detect anomalies and predict maintenance needs, improving flight safety and operational efficiency. The availability of quality data, describing anomalies that can point to the need for maintenance apart from normal maintenance needs, is rare. The absence of rare event data affects the accuracy of these algorithms. This absence has also led to interest in collaborative data sharing among airline operators to improve predictive maintenance models [4].
The Independent Data Consortium for Aviation (IDCA) [5], among others, recognizes the potential of collaborative data sharing but faces implementation hurdles, including privacy concerns, intellectual property issues, and regulatory compliance. Additionally, the dynamic and sensitive nature of aviation data necessitates careful consideration of scalability and security. Federated Learning (FL)—a decentralized approach to machine learning—offers a promising solution by allowing model training on distributed data without centralized aggregation, thereby ensuring data sovereignty and protecting privacy [6].

## 1.1 Problem statement

Despite the promising potential of federated learning (FL) for predictive maintenance in the aviation sector, there is a significant research gap concerning the architectural design and implementation strategies required to transition from centralized machine learning workflows to decentralized FL systems. Existing literature primarily focuses on the theoretical and algorithmic aspects of FL, often overlooking the practical challenges associated with data partitioning, communication overhead[7][8], model convergence, and client heterogeneity. The careful consideration of architecture and design decisions in federated learning is essential to address the technical complexities and operational challenges inherent in decentralized data processing. These decisions directly impact the system's modularity, scalability, reliability, security, and overall efficiency, ensuring the development of robust, high-performance federated learning solutions.

### 1.1.1 Research questions

To address the challenges posed by federated learning in predictive maintenance for the aviation sector, we investigate the adaptation of software architecture patterns and critical design decisions using the N-CMAPSS simulated flight dataset. Our research is structured around the following research questions:

- RQ1: How can software architecture patterns be tailored to accommodate federated learning systems?
  - SQ: What criteria or metrics should guide the selection of software architecture patterns to accommodate the nature of federated learning systems?
- RQ2:What key design decisions need to be made in order to facilitate federated learning models?

### 1.1.2 Research method

This research employs a systematic approach to explore and implement software architecture patterns tailored for federated learning in the context of predictive maintenance using the N-CMAPSS dataset. Initially, exploratory data analysis (EDA) is conducted on the N-CMAPSS dataset to understand its characteristics and prepare for modeling tasks. Following this, potential federated learning architectures are investigated through a comprehensive review of existing literature and the identification of suitable patterns such as client-server, monolithic and microservices architecture. Criteria for selecting appropriate software architecture patterns are then established, focusing on metrics like scalability, modularity and maintainability, which are validated through testing and analysis. Subsequently, key design decisions critical for facilitating effective federated learning models are identified, with a particular emphasis on their impact on metrics previously defined. Finally, prototypes are developed and validated against the identified criteria and design decisions, providing a robust framework for evaluating the metrics and suitability of federated learning.

## 1.2 Contributions

Our research makes the following contributions:

1. Identification of Optimal Software Architecture Patterns: Investigated and identified software architecture patterns most suitable for supporting federated learning systems, addressing their unique decentralized nature.
2. Critical Design Decisions for Federated Learning: Examined and defined key design decisions crucial for enabling effective implementation and operation of federated learning models in practical scenarios.
3. Architectural Design Patterns for Federated Learning: Studied architectural design patterns that can accommodate federated learning in a reliable and scalable microservices-based architecture.
4. Demonstration in the Aviation Industry: Demonstrated the viability of federated learning in the critical scenario of the aviation industry with the help of the N-CMAPSS dataset, potentially saving thousands of lives by enabling secure inter-airline data sharing for predictive maintenance of turbofan jet engines.
5. Integration of Fabric Framework: Leveraged the Fabric framework to facilitate the implementation of federated learning, providing essential tools and infrastructure to support secure and efficient data sharing and model training
6. Performance Evaluation Using JMeter: Conducted extensive testing using JMeter to evaluate the performance and scalability of the implemented architectures, providing insights into their effectiveness.
7. Maintainability and modularity testing: Performed maintainability and modularity testing using Radon to assess the quality of the software codebase. Radon was used to analyze metrics such as cyclomatic complexity, maintainability index, and code metrics, ensuring that the system architecture not only meets functional requirements but is also easy to maintain and extend.

## 1.3 Outline

In Chapter 2 we provide the background and context for this thesis. Chapter 3 details the exploratory data analysis (EDA) performed on the N-CMAPSS dataset. Chapter 4 we compare different software architectures suitable for federated learning. Chapter 5 elaborates on the key design strategies employed in this research. The experimental setup is discussed in Chapter 6.1. The results for testing based on metrics is presented in Chapter 7. Results and their implications are discussed in Chapter 8. Chapter 9 reviews the existing work related to this thesis. Finally, we present our concluding remarks in Chapter 10 outlines directions for future work.

# Chapter 2

# Background

This chapter provides the essential context for understanding the research presented in this thesis.

### 2.0.1 Federated Learning

Federated Learning was a solution first introduced by Google [9]. It is a way to train models by preserving privacy. In contrast to traditional centralized training approaches where data is aggregated into a central repository for training, posing significant privacy risks, federated learning allows numerous clients, such as edge devices and organizations, to collaboratively train a model while preserving the privacy of individual data sources. It is a decentralized approach that minimizes the need for data transfer, reduces privacy concerns by keeping sensitive information local, and enhances model correctness by learning from diverse, heterogeneous datasets distributed across different environments.
The figure 2.1 shows the federated learning process which involves [10]:

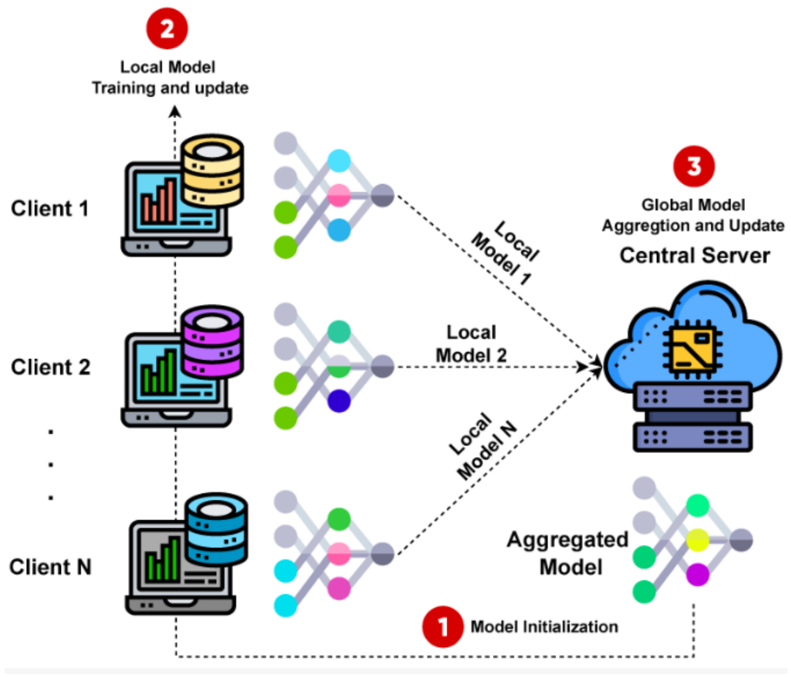


**Figure 2.1: Federated learning process**[10]

- Initialization: The client downloads the latest global model with some initial parameters from a central server.
- Local training: Each client device or edge server trains the model on the local dataset. This training process typically involves multiple iterations of forward and backward passes through the data, updating the model parameters to minimize the loss function.
- Model aggregation: After completing local training, each client sends its updated model parameters

(e.g., gradients) back to the central server.The central server aggregates the model updates received from all participating clients, typically using techniques like Federated Averaging. The aggregated updates are then used to update the global model.

This process "brings the machine learning model to the data" rather than "bringing the data to the model," ensuring that data privacy is maintained throughout the training process.

### Comparison of Centralized and Federated Learning

In order to understand how federated learning works, comparing the difference with traditional centralized learning is important [11].

**Data Distribution and Storage:**

Traditional Centralized: Data is stored in a central repository or data center. All data is aggregated in one location, which facilitates easier access and management for training models.

Federated Learning: Data remains distributed across multiple devices(or edge nodes) or locations. Each device(or edge node) holds its own data, and data typically does not leave the device due to privacy concerns.

**Model Training:**

Traditional Centralized: Models are trained using all available data in a centralized location or server. This allows for comprehensive training on a large dataset in one place.

Federated Learning: Models are trained locally on each device or edge node using the data available on that device. Only model updates (not raw data) are typically sent to a central server. To prevent raw data from being uploaded, Federated Learning employs secure techniques such as differential privacy, secure multi-party computation, and homomorphic encryption. These methods ensure that only model parameters or gradients are shared, keeping the raw data local and confidential.

**Communication, Data Governance and Privacy:**

Traditional Centralized: Communication is straightforward between the central server and clients. Data acess and usage are managed centrally, often through strict data handling protocols.

Federated Learning: Communication involves sending only model updates or gradients to a central server or coordinator. This preserves data sovereignty and privacy as raw data is decentralized and not shared centrally.

**Resource Requirements:**

Traditional Centralized: Requires significant computational resources centrally to process and train models on large datasets.

Federated Learning: Relies on computational resources distributed across many devices. The burden on each device is typically lighter as it only needs to train on its local data and send updates however due to frequent communication between devices and the server to send model updates, it can be costly in terms of bandwidth and time, especially with large models or slow networks.

**Scalability and Flexibility:**

Traditional Centralized: Scalability can be a challenge due to the need for large-scale infrastructure and potential bottlenecks in data access. Limited flexibility as it relies on central data storage and homogeneous infrastructure.

Federated Learning: Scales by adding more distributed resources. As more devices join the network, the overall computational capacity increases. Each device contributes to the training process by handling local data and periodically sending model updates to the central server for aggregation. This distributed approach helps manage the computational load and mitigates central bottlenecks. More flexible in accommodating different types of devices and data sources. It can adapt to various device capabilities and data distributions, enhancing its applicability across diverse environments.

| **Aspect** | **Centralized Learning** | **Federated Learning** |
|---|---|---|
| **Data Distribution and Storage** | Central repository. | Individual devices (or edge nodes). |
| **Model Training** | Trained at one place. | Trained locally on each device (or edge node). |
| **Communication, Data Governance, and Privacy** | Simple communication. Data access and usage are governed centrally. | Only model updates are sent, preserving data sovereignty and privacy. |
| **Resource Requirements** | High central computational resources. | Distributed computational resources. |
| **Scalability** | Scalability challenges due to central infrastructure. | Scales by adding distributed resources. |

**Table 2.1: Summary of Comparison of Centralized and Federated Learning**

### Features of federated learning

The key features of federated learning are [12]:

- **Decentralized Data:** Federated Learning enables machine learning models to be trained on data distributed across multiple devices or servers without centralizing the data.
- **Privacy-Preserving:** Data remains on the user's device, ensuring that sensitive data is not exposed to others. This privacy-preserving nature makes Federated Learning suitable for applications requiring high privacy and security. To ensure privacy is preserved, techniques such as differential privacy, secure multi-party computation, and homomorphic encryption are often used [13]. These methods protect the data and model updates from being reverse-engineered to access the original data.
- **Collaborative Learning:** Multiple devices or servers can collaborate and contribute to the training of a machine learning model, even if they have different datasets.
- **Efficiency:** By reducing the amount of data that needs to be transmitted to a central server, Federated Learning minimizes computational and communication overheads. However, the efficiency of Federated Learning can depend on the number of participating nodes and the parameters of the system. Effective communication protocols and optimization strategies are crucial to managing these overheads.
- **Iterative Updates:** Federated Learning allows for continuous updates to the machine learning model over time without the need for full retraining. This iterative process involves periodically aggregating model updates from multiple devices and updating the global model. This approach ensures that the model continually improves while leveraging the latest data available locally on the devices.
- **Flexibility:** Federated Learning is a versatile approach that can be used in various settings, including edge devices, mobile devices, and data centers.

### Types of federated learning

Based on the distribution of data, federated learning can be classified into [12]:

- **Horizontal federated learning**: In horizontal federated learning the datasets held by different clients have the same feature space but differ in the samples. This type is suitable when different organizations possess data of similar kinds but pertaining to different individuals.
- **Vertical federated learning**: This type of federated learning involves datasets that share the same sample space but have different feature spaces. This type is useful when different organizations possess different types of information about the same set of entities.
- **Federated transfer learning**: Federated transfer learning is used when the datasets held by clients differ in both samples and features. This type leverages transfer learning techniques to train a model in scenarios where the data distribution and feature spaces vary significantly across clients.

Based on the type of participating clients and their characteristics, federated learning can be classified into [14, 15]:

- Cross silo: Cross-Silo Federated Learning involves a relatively small number of clients, such as organizations or institutions, that collaborate to train a global model. These clients, or silos, typically have more significant computational resources and stable network connections compared to individual devices in cross-device FL.

- Cross device: Cross-Device Federated Learning involves a large number of clients, such as mobile phones, tablets, or IoT devices, that collaboratively train a global model. This type of FL focuses on leveraging the distributed nature of many user devices.

#### Federated learning algorithms

Federated Learning employs a variety of algorithms to handle the distributed nature of data and computation. The most commonly used algorithms include [16, 17]:

- Federated Average (FedAvg): It works by averaging the model weights from multiple clients to update the global model. The steps include initializing the global model, distributing it to clients, locally updating the model on each client, and then averaging the updates to form a new global model. This process iterates until convergence. FedAvg also incorporates versioning to track different iterations of the global model, ensuring that each client works with the correct model version during training and update cycles.
- Federated Stochastic Gradient Descent(FedSGD): FedSGD is a variant where each client computes the gradient of the loss function with respect to its local data and sends this gradient to the central server. The server then aggregates these gradients to update the global model. This method requires more frequent communication than FedAvg but can be beneficial for certain applications.
- Federated Optimization(FedOpt): FedOpt encompasses a set of optimization techniques adapted for federated settings. These techniques include adaptive learning rates and momentum-based methods that aim to improve convergence rates and handle the variability in client data distributions.
- Secure aggregation: Secure Aggregation ensures that model updates from clients are encrypted, preventing the central server from accessing individual updates. Techniques like homomorphic encryption and secure multiparty computation are used to aggregate updates in a privacy-preserving manner. Secure aggregation typically takes place on the server-side, where the server aggregates the encrypted updates without decrypting them, ensuring that individual client data remains confidential throughout the process.

#### Challenges and Solutions

Despite its advantages, federated learning faces several key challenges [16]:

- **Data Heterogeneity:** Clients often have non-identically distributed data, which can lead to issues in model convergence and accuracy. Solutions such as personalized federated learning and meta-learning have been proposed to address these challenges.
- **Scalability:** The scalability of federated learning systems is limited by the number of clients and the communication overhead. Hierarchical federated learning and asynchronous update mechanisms are among the solutions to enhance scalability.
- **Communication Efficiency:** Frequent communication between clients and the central server can be a bottleneck. Techniques like compression of model updates and periodic aggregation help reduce communication costs.
- **Privacy and Security:** Ensuring the privacy and security of data and model updates is paramount. Advanced cryptographic methods and privacy-preserving techniques are continuously being developed to mitigate these concerns.

### 2.0.2 Microservices

Software architecture refers to the high-level structure of a software system, including the arrangement of its components and the ways they interact [18]. It's a critical aspect of the development process, since it determines the system's robustness, scalability, and maintainability. Traditionally, many systems were built using a monolithic architecture, where all functionalities are intertwined and deployed as a single unit. Although monolithic architecture based systems are simpler to develop, they pose several significant challenges as systems grow. Scalability becomes problematic because any change, even a small one, requires redeploying the entire application. This can lead to longer development cycles and difficulties in implementing continuous deployment. Furthermore, the large, complex codebases typical of monolithic

systems are hard to manage, making it difficult to isolate and fix bugs or add new features.
The issues with monolithic architecture led to the rise of distributed computing such as Service oriented architecture. Therefore, microservice architecture has emerged as an architectural style by promoting the use of fine-grained services and the cooperation of the nodes in the cloud [19]. Netflix, Amazon, Spotify and other companies are moving their applications from monolithic architecture towards microservices architecture [20]. Microservices based architecture design is an approach where a software system or application is composed of small, independent services that communicate through well-defined APIs.[21].
Technological advancement has furthered the adoption of microservices architecture. Cloud computing provides the necessary infrastructure to deploy and scale services efficiently. Containerization technologies such as Docker allow services to run in isolated environments, ensuring consistency across different deployment stages. Orchestration tools such as Kubernetes automate the deployment, scaling, and management of containerized applications, making it easier to handle complex microservices architectures. DevOps practices such as continuous integration/continuous deployment (CI/CD) further support the adoption of microservices.
There are many advantages of a microservices architecture [22], the main advantages are :

- Scalability: Each service can be scaled independently based on demand, optimizing resource usage.
- Fault tolerance: The breakage of one service can be easily repaired and will not break the entire system.
- Maintainability: Smaller, modular codebases are easier to manage, test, and debug.
- Quicker releases: It enables faster releases because each service evolves and deploys independently.

Despite the advantages, of a microservices architecture, they come with their own set of challenges [19]. The increased number of services introduces more complexity in terms of infrastructure due to the distributed nature of this architecture, managing service communication can be challenging. Developers may have to write extra code to ensure smooth communication between modules. When working with smaller teams a monolithic architecture is easier to manage as there is a single codebase to maintain and test. Organizations need to weigh out the architecture that suits them best by taking into consideration factors like team structure, project complexity and resource availability.

### 2.0.3 NASA Turbofan Jet Engine Dataset (N-CMAPSS)

The N-CMAPSS dataset represents a significant enhancement and extension of the original C-MAPSS dataset [23], addressing key limitations and introducing new features to facilitate more comprehensive and realistic engine health monitoring and prognostics research. In this work, improvements are introduced in two primary aspects [24]:

- First, the dataset now includes simulations of complete flights as recorded on board a commercial jet, encompassing climb, cruise, and descent flight conditions representative of various commercial flight routes . This enhancement provides researchers with a more holistic view of engine operation and degradation patterns under diverse operational scenarios, enhancing the dataset's realism and applicability to real-world settings.
- Second, the fidelity of degradation modeling is significantly increased by establishing a relationship between the onset of the degradation process and the engine's operational history. By capturing the influence of operational factors on degradation progression, such as flight duration, load, and environmental conditions, the N-CMAPSS dataset offers more nuanced insights into engine health dynamics and enables the development of more accurate prognostics models.

The N-CMAPSS dataset is a valuable resource in the field of prognostics and condition-based maintenance (CBM) for engineering assets, particularly aircraft engines. This dataset, released by the NASA Ames Prognostics Center of Excellence (PCoE) in collaboration with ETH Zurich and Palo Alto Research Center (PARC), provides simulated run-to-failure trajectories for a small fleet of large turbofan engines under realistic flight condition [23]. The N-CMAPSS dataset contains synthetic run-to-failure degradation trajectories for a fleet of turbofan engines under real flight conditions, with eight sets of data from 128 units and seven different failure modes. Each data file includes both development and test datasets, containing six types of variables such as operative conditions, measured signals, virtual sensors, engine health parameters, RUL labels, and auxiliary data [24]. This comprehensive dataset is designed to reflect accurate engine degradation patterns, supporting research in predictive maintenance and prognostics [25]. Figure 2.2 provides a schematic representation of the simulated turbofan engine and its sensors as used in the N-CMAPSS dataset. It illustrates the various components of the engine and the placement of sensors that capture the critical operational parameters, essential for modeling engine degradation and predicting failures.

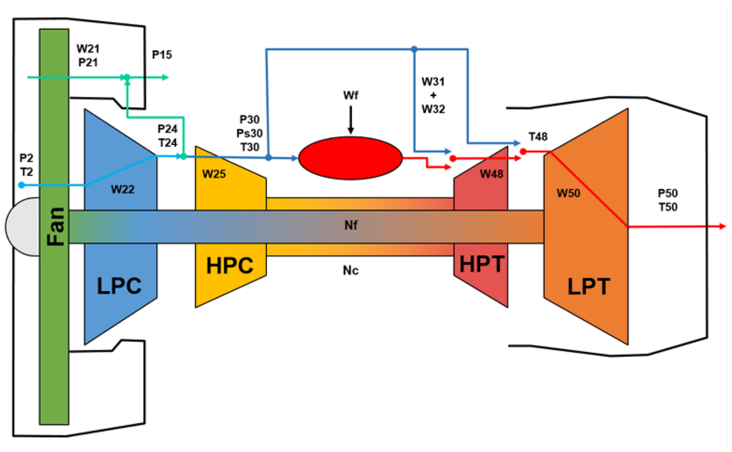


**Figure 2.2: Schematic representation of the CMAPSS model[24]**

### 2.0.4 FABRIC

The FABRIC (Framework for Accelerated Built-In Resilience Infrastructure for Computing) testbed is an advanced research infrastructure designed to support the exploration and development of next-generation computing systems and networks. The FABRIC testbed is developed by a consortium of institutions and organizations aiming to provide a versatile and scalable platform for research and experimentation in various domains, including networking, cybersecurity, distributed computing, and federated learning [26].

#### Key features

- Scalable infrastructure: FABRIC offers a highly scalable and flexible infrastructure that enables researchers to design, deploy, and manage complex experimental setups. It supports a wide range of hardware and software configurations, allowing for customized and optimized environments tailored to specific research needs.
- Distributed environment: The testbed comprises geographically distributed nodes interconnected by high-speed networks. This distribution allows for realistic experimentation with latency, bandwidth, and other network parameters, closely mimicking real-world conditions.
- High- performance computing: FABRIC provides access to high-performance computing resources, including powerful processors, accelerators, and storage systems. These resources facilitate the execution of compute-intensive tasks and large-scale simulations essential for advanced research.
- Integration with Docker and Kubernetes: The testbed supports containerization technologies like Docker and orchestration tools like Kubernetes, making it easier to manage and scale applications. This integration streamlines the deployment of microservices architectures and enhances the efficiency of development and testing workflows.
- Advanced networking: FABRIC includes advanced networking features, such as software-defined networking (SDN) and network function virtualization (NFV). These capabilities enable researchers to experiment with innovative network designs, protocols, and security mechanisms.

The FABRIC testbed is a state-of-the-art research infrastructure designed to support a wide array of scientific and engineering endeavors. Its scalable, distributed, and high-performance nature makes it an ideal platform for investigating and advancing federated learning systems, among other cutting-edge technologies. By leveraging FABRIC, researchers can conduct realistic and impactful experiments that contribute to the development of resilient, efficient, and secure computing and networking systems. Additionally, the FABRIC Across Borders (FAB) is an extension of the testbed connecting the core North American network with global institutions, enabling international collaboration and accelerating scientific discovery.

# Chapter 3

# Exploratory Data Analysis (EDA)

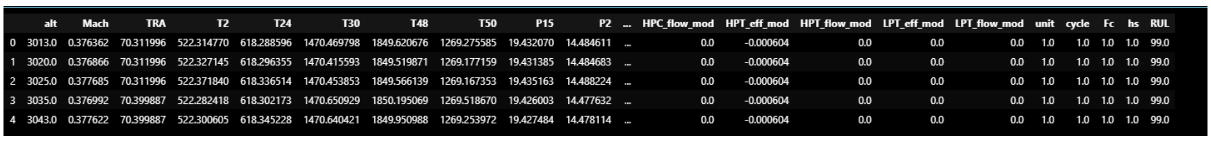
Exploratory Data Analysis (EDA) is a crucial step in understanding the characteristics and underlying patterns within a dataset. In this section, we perform EDA on the N-CMAPSS dataset to gain insights into the data distribution, identify potential anomalies, and understand the relationships between different variables.

### 3.0.1 Dataset overview

The N-CMAPSS dataset consists of eight sets of data from 128 units, encompassing seven different failure modes affecting the flow (F) and/or efficiency (E) of all rotating sub-components. Each dataset is stored in a Hierarchical Data Format (HDF5) file and includes variables such as operative conditions, measured signals, virtual sensors, engine health parameters, Remaining Useful Life (RUL) labels, and auxiliary data like unit numbers and flight cycle numbers. This is explained in depth in Chapter 2. For this exploratory data analysis (EDA), we focus on the DS01 dataset within N-CMAPSS since it is the least complex. The DS01 dataset is considered the least complex because it has fewer failure modes and simpler operational scenarios compared to other datasets in the N-CMAPSS collection. This reduced complexity makes it more manageable for initial analysis and model development, allowing us to establish a foundational understanding before tackling more intricate datasets.

### 3.0.2 Data Loading and Initial Inspection

To begin the EDA, we load the dataset and perform an initial inspection to understand its structure and content as illustrated in Figure 3.1. The data files provide two sets of data: the development dataset and the test dataset. Each dataset contains six types of variables: operative conditions, measured signals, virtual sensors, engine health parameters, RUL labels, and auxiliary data.

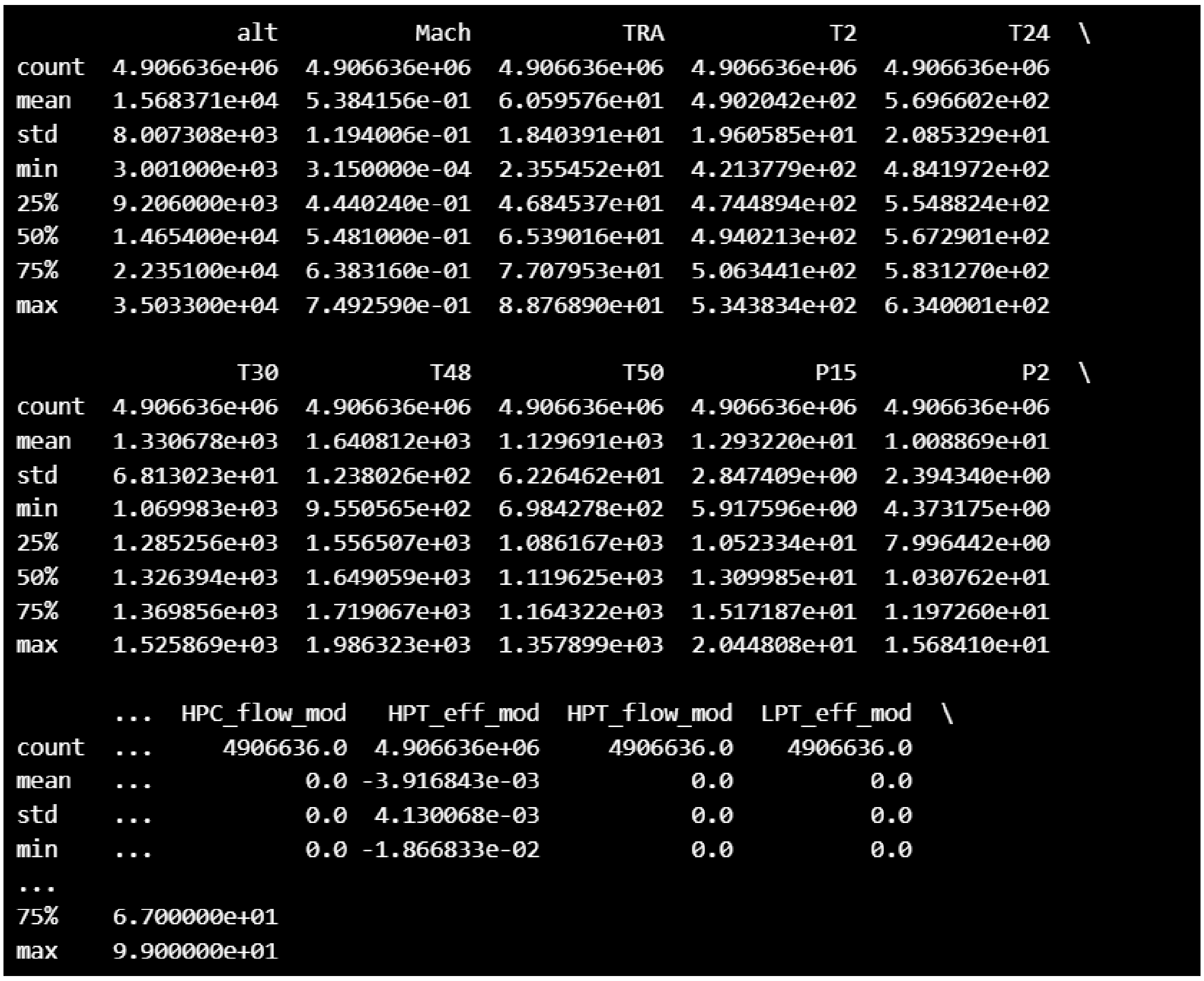


| | alt | Mach | TRA | T2 | T24 | T30 | T48 | T50 | P15 | P2 | ... | HPC_flow_mod | HPT_eff_mod | HPT_flow_mod | LPT_eff_mod | LPT_flow_mod | unit | cycle | Fc | hs | RUL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 3013.0 | 0.376362 | 70.311996 | 522.314770 | 618.288596 | 1470.469798 | 1849.620676 | 1269.275585 | 19.432070 | 14.484611 | ... | 0.0 | -0.000604 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 99.0 |
| 1 | 3020.0 | 0.376866 | 70.311996 | 522.327145 | 618.296355 | 1470.415593 | 1849.519871 | 1269.177159 | 19.431385 | 14.484683 | ... | 0.0 | -0.000604 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 99.0 |
| 2 | 3025.0 | 0.377685 | 70.311996 | 522.371840 | 618.336514 | 1470.453853 | 1849.566139 | 1269.167353 | 19.435163 | 14.488224 | ... | 0.0 | -0.000604 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 99.0 |
| 3 | 3035.0 | 0.376992 | 70.399887 | 522.282418 | 618.302173 | 1470.650929 | 1850.195069 | 1269.518670 | 19.426003 | 14.477632 | ... | 0.0 | -0.000604 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 99.0 |
| 4 | 3043.0 | 0.377622 | 70.399887 | 522.300605 | 618.345228 | 1470.640421 | 1849.950988 | 1269.253972 | 19.427484 | 14.478114 | ... | 0.0 | -0.000604 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 1.0 | 99.0 |

**Figure 3.1: First few rows of the training dataset**

### 3.0.3 Summary statistics

We begin by generating summary statistics for the numerical variables in the dataset to understand their distributions, central tendencies, and variability. Figure 3.2 presents the summary statistics of the training dataset. The presence of zero or low values in the summary statistics is attributed to the varying measurement intervals and types of features. Specifically, some features are calculated on a per-second basis, while others are assessed per unit. Additionally, some features are not measured at all, resulting in zero values for those instances.



```
                alt          Mach           TRA            T2           T24  \
count  4.906636e+06  4.906636e+06  4.906636e+06  4.906636e+06  4.906636e+06
mean   1.568371e+04  5.384156e-01  6.059576e+01  4.902042e+02  5.696602e+02
std    8.007308e+03  1.194006e-01  1.840391e+01  1.960585e+01  2.085329e+01
min    3.001000e+03  3.150000e-04  2.355452e+01  4.213779e+02  4.841972e+02
25%    9.206000e+03  4.440240e-01  4.684537e+01  4.744894e+02  5.548824e+02
50%    1.465400e+04  5.481000e-01  6.539016e+01  4.940213e+02  5.672901e+02
75%    2.235100e+04  6.383160e-01  7.707953e+01  5.063441e+02  5.831270e+02
max    3.503300e+04  7.492590e-01  8.876890e+01  5.343834e+02  6.340001e+02

                T30           T48           T50           P15            P2  \
count  4.906636e+06  4.906636e+06  4.906636e+06  4.906636e+06  4.906636e+06
mean   1.330678e+03  1.640812e+03  1.129691e+03  1.293220e+01  1.008869e+01
std    6.813023e+01  1.238026e+02  6.226462e+01  2.847409e+00  2.394340e+00
min    1.069983e+03  9.550565e+02  6.984278e+02  5.917596e+00  4.373175e+00
25%    1.285256e+03  1.556507e+03  1.086167e+03  1.052334e+01  7.996442e+00
50%    1.326394e+03  1.649059e+03  1.119625e+03  1.309985e+01  1.030762e+01
75%    1.369856e+03  1.719067e+03  1.164322e+03  1.517187e+01  1.197260e+01
max    1.525869e+03  1.986323e+03  1.357899e+03  2.044808e+01  1.568410e+01

       ...  HPC_flow_mod   HPT_eff_mod  HPT_flow_mod  LPT_eff_mod  \
count  ...     4906636.0  4.906636e+06     4906636.0    4906636.0
mean   ...           0.0 -3.916843e-03           0.0          0.0
std    ...           0.0  4.130068e-03           0.0          0.0
min    ...           0.0 -1.866833e-02           0.0          0.0
...
75%    6.700000e+01
max    9.900000e+01
```

**Figure 3.2: Summary statistics of the training dataset**

### 3.0.4 Correlation analysis

To identify potential relationships between variables, we compute the correlation matrix and visualize it using a heatmap. This analysis helps in understanding how different operative conditions and measured signals relate to each other and to the engine health parameters. Figure 3.3 displays the correlation matrix for the dataset.From this correlation matrix, we can re-confirm that 'fan_eff_mod', 'fan_flow_mod', 'LPC_eff_mod', 'LPC_flow_mod', 'HPC_eff_mod','HPC_flow_mod','HPT_flow_mod', 'LPT_eff_mod' and 'LPT_flow_mod' provide little to no usability for training the model.
Furthermore, due to the dataset's design reported in [24], where it was noted that these dataset columns are included despite being deemed irrelevant, we did not proceed with rolling window correlations. This insight guided the decision to stop further exploratory data analysis (EDA) for these columns. Consequently, these sensor columns have been excluded from the dataset.

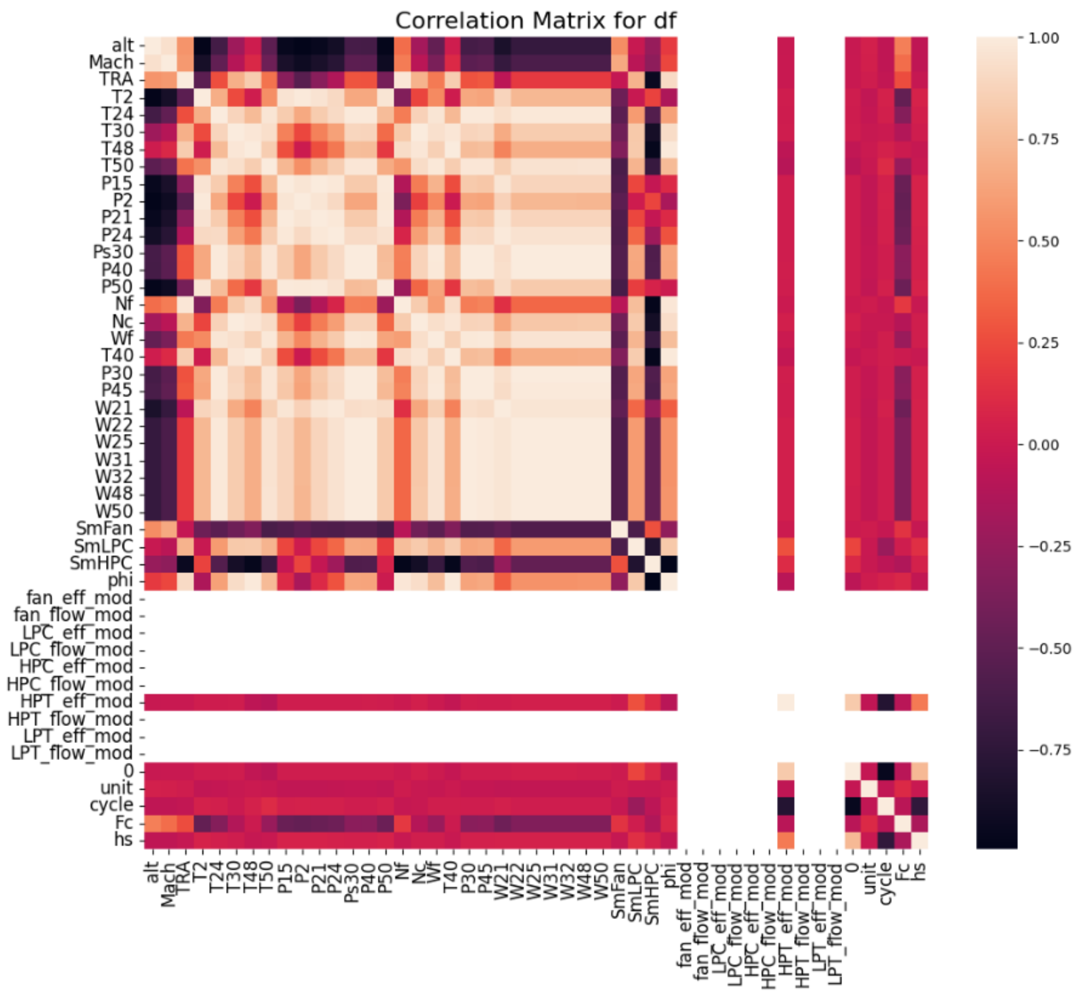


**Figure 3.3: Correlation matrix for Dataset**

### 3.0.5 Feature Importance

Feature importance is a critical aspect of model interpretability in machine learning, allowing us to understand which variables have the most influence on target predictions. In this section, we analyze the importance of different features in predicting the Remaining Useful Life (RUL) of aircraft engines using a Random Forest classifier.
To assess feature importance, we employ the Random Forest algorithm, which is well-suited for this task due to its ensemble nature and inherent ability to rank features based on their contribution to prediction accuracy.
The dataset is first prepared by separating the target variable (RUL) from the feature variables, which include operational settings, sensor measurements, and other relevant parameters. The Random Forest classifier is then trained on the dataset, and the importance of each feature is determined by its impact on the model's performance. Specifically, feature importance is calculated based on how much each feature reduces the impurity of the nodes in the decision trees.
The feature importances are visualized using a horizontal bar chart in figure 3.4, which provides a clear representation of which features are most significant in predicting RUL. This visualization helps identify the key factors affecting engine degradation and informs maintenance strategies. From this feature importance plot the following conclusions can be drawn:

- HPT_eff_mod (High-Pressure Turbine efficiency modifier) is the most critical feature for predicting the Remaining Useful Life (RUL) of the system. Following this, cycle (number of cycles) and Unit (specific equipment instance) are also highly influential.
- Moderately significant features include Mach (Mach number), SmLPC (Smoothed Low-Pressure Compressor), and TRA (Throttle Resolver Angle).

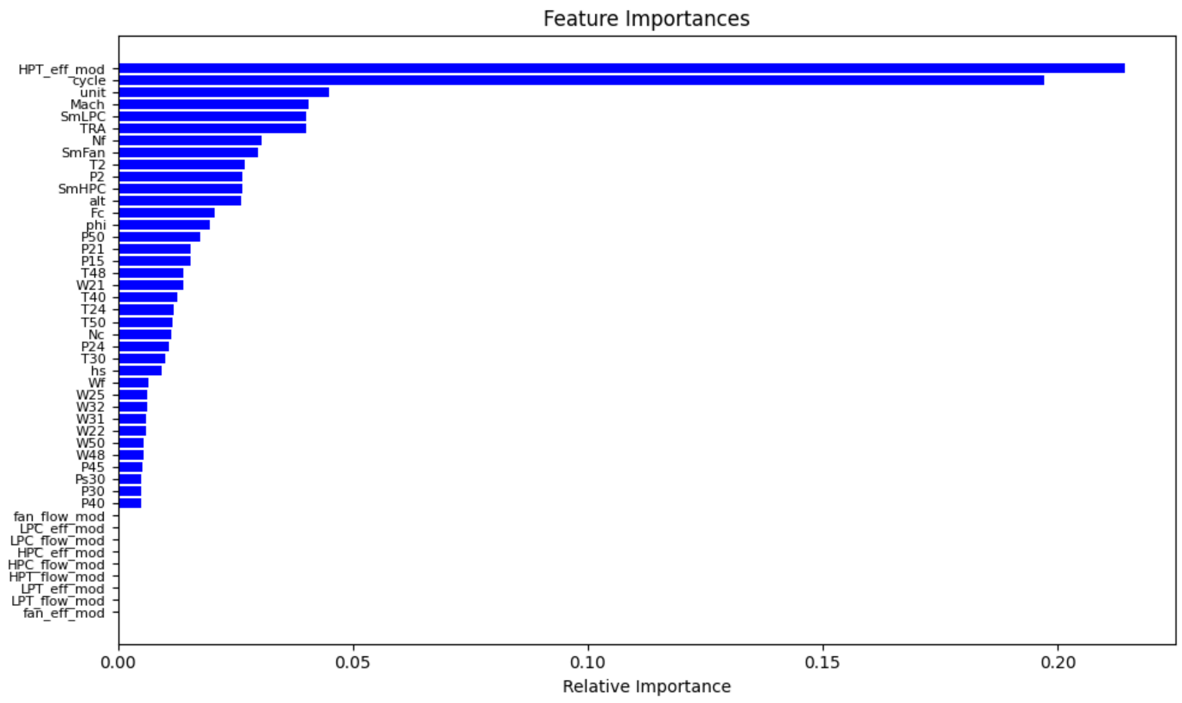


**Figure 3.4: Visualization of Feature importance**

- Modifications and efficiency measures in components like the fan, Low-Pressure Compressor (LPC), High-Pressure Compressor (HPC), and Low-Pressure Turbine (LPT) show low relative importance, indicating they contribute less to the prediction model.

The analysis indicates that the Mach number, which relates to aircraft speed, is notably influential in predicting RUL. This result might suggest a counterintuitive relationship where higher aircraft speeds could potentially correlate with longer engine life. However, this observation warrants further investigation to understand the underlying causes and verify if there are any additional factors influencing this correlation. Thus, further analysis is needed to determine the root causes of the feature importance and to validate these findings. Understanding feature importance enables us to focus on the most influential features, potentially reducing the dataset's dimensionality and enhancing the model's performance and interpretability. This step is essential for building robust prognostic models that can reliably predict the RUL of aircraft engines and guide maintenance decisions.

Chapter 4

# System Design & Architecture in Federated Learning

In the design and implementation of FL systems, several architectural components and patterns must be carefully considered to ensure efficiency, scalability, and data privacy. This section explores the essential components of FL systems and evaluates different software architecture patterns based on key criteria chosen.

## 4.1 Architectural Components of Federated Learning

Federated Learning (FL) systems consist of several essential components that collaborate to enable distributed model training across a network of devices while preserving data privacy and security. Understanding these components is crucial for adapting software architecture patterns to effectively support FL systems. The key components, as illustrated in Figure 4.1, are [27]:

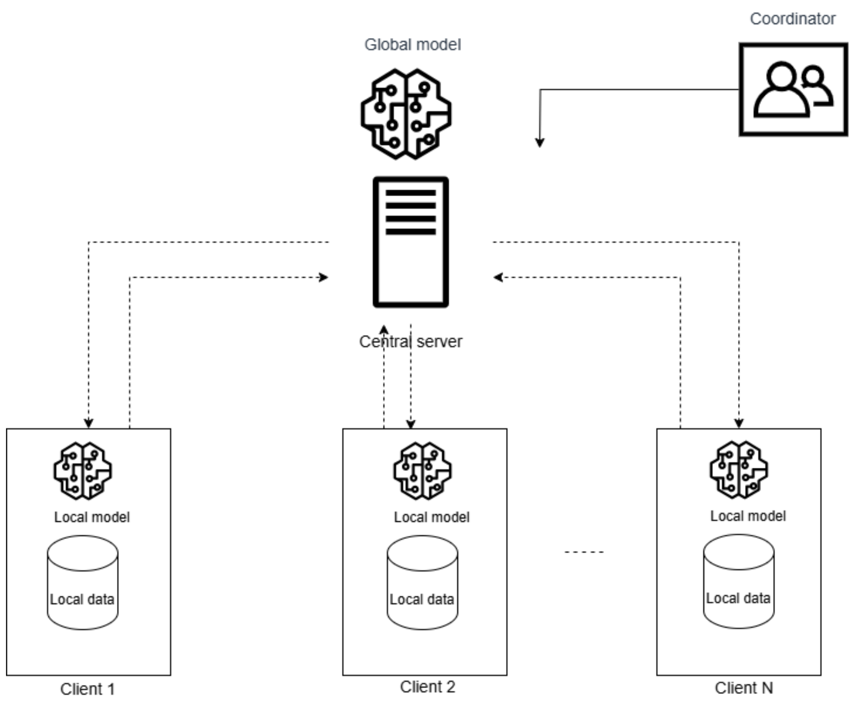


**Figure 4.1: Components in federated learning**

- Client Devices: Client devices in FL systems typically include mobile devices, Internet of Things(IoT) endpoints, or edge nodes. These devices possess local data sets and computational resources necessary for training machine learning models. The main function of client devices include:

  - Data Providers: Typically mobile devices, edge devices, or IoT sensors that generate or hold data.
  - Local Training: Execute model training on local data without transmitting raw data to the server.
  - Model Updates: Send locally trained model updates (parameters) to the server or aggregator.

- Central server: The server, also known as the aggregator, plays a pivotal role in the federated learning architecture by acting as the central entity that manages and integrates the collective intelligence gathered from distributed client devices. The main function of central server includes:
  - Model updates: Receives model updates from client devices.
  - Aggregation: Aggregates model updates from multiple clients to create a global model.
  - Distribution: Sends updated global model parameters back to clients for further training.
- Coordinator: The coordinator is the management entity that oversees the entire federated learning process, ensuring that all components work harmoniously to achieve the desired outcomes. The main functions of the coordinator are:
  - Management Entity: Coordinates the federated learning process.
  - Client Selection: Determines which clients participate in each round of training.
  - Model Initialization: Distributes initial model parameters to client devices.
  - Convergence Monitoring: Monitors convergence and quality of the global model.

In summary, the process works as follows: The Coordinator starts the federated learning cycle by selecting a subset of Client Devices and distributing the initial global model. Client Devices then perform local training using their own data and send the updated model parameters to the Central Server. The Central Server aggregates these updates to form a new global model, which is then sent back to the Client Devices for further training. This iterative process continues until the global model achieves convergence and is then stopped by the Coordinator.

## 4.2 Software Architecture Pattern Criteria

The criteria for selecting software architecture patterns are:

- Scalability: Federated learning involves training models across a large number of distributed clients especially in this use case where multiple airlines are meant to collaborate for better engine health predictions. Therefore using scalability as a metric ensures that the architecture can handle increasing numbers of clients and data volumes efficiently, without compromising performance or introducing bottlenecks.
- Maintainability: In federated learning, where models are trained across distributed client devices, the need for updates and bug fixes is constant. Maintainability ensures that changes can be implemented smoothly across all clients without disrupting the learning process or compromising data integrity. This is essential for keeping the federated learning system up-to-date with the latest improvements and fixes.
- Modularity: Modular design allows the federated learning architecture to be divided into smaller, independent modules or components. Each module can be developed, tested, and maintained separately, simplifying the overall development process. Developers can focus on specific functionalities or components without needing to understand the entire system at once, leading to faster development cycles and easier debugging. Modular architectures in federated learning can also promote component reuse and scalability. New features or improvements can be implemented by adding or modifying modules without affecting the entire system, which supports flexibility and adaptability as the number of participating clients and complexity of models evolve.

Although there are other criteria that can be used to select suitable software architecture patterns such as:

- Security: Security is essential in federated learning, where sensitive data is distributed across multiple clients. The architecture must incorporate robust security measures to protect data privacy during model training, aggregation, and communication among participating airlines. Security ensures compliance with regulations and builds trust among stakeholders.
- Interoperability: Federated learning architectures often need to integrate with existing IT infrastructures and data sources across different airlines. Interoperability ensures seamless data exchange,

compatibility with diverse hardware and software environments, and effective collaboration among multiple stakeholders.

- Flexibility: The architecture should be flexible enough to accommodate different machine learning algorithms, model architectures, and optimization techniques. This flexibility supports experimentation with new methodologies and adaptation to evolving business requirements and technological advancements.

In this study, we focus on the criteria of scalability, modularity, and maintainability because they intersect to form a strong foundation for designing and evaluating software architecture patterns in federated learning scenarios. These criteria collectively address critical challenges that are pivotal for the successful deployment and operation of federated learning systems, particularly in dynamic and collaborative environments involving multiple airlines.

## 4.3 Architecture Selection based on Key Criteria

Selecting the appropriate architecture involves evaluating traditional and modern software architecture patterns against the criteria identified. The primary architecture patterns considered for Federated learning are Client-Server, Monolithic, and Microservices. Each of these architectures has its strengths and weaknesses when evaluated against criteria defined above, scalability, modularity, and maintainability.

### 4.3.1 Client-server architecture

The Client-Server architecture is a traditional model in which clients (devices or nodes) perform local model training on their respective datasets. Periodically, these clients send their model updates to a central server. The server then aggregates these updates to form a global model, which is redistributed to the clients. The evaluation of this architecture against the chosen criteria is as follows [28, 29]:

- **Scalability:**
  - **Pros:** This architecture supports a large number of clients, as each client operates independently during the local training phase.
  - **Cons:** The central server can become a bottleneck if the number of clients or the volume of data to be processed increases significantly. The server's capacity to handle and aggregate updates limits scalability.
- **Modularity:**
  - **Pros:** There is a clear separation of concerns where each client is responsible for local training and the server is responsible for aggregation, enhancing modularity.
  - **Cons:** While the client-server separation is modular, increasing complexity within each component can lead to integration challenges.
- **Maintainability:**
  - **Pros:** Managing interactions between clients and the server is relatively straightforward. Updating server logic typically does not disrupt client operations.
  - **Cons:** Centralized control requires meticulous coordination to ensure consistency and smooth operations across all clients, particularly during updates.

### 4.3.2 Monolithic Architecture

In a Monolithic architecture, all functionalities are contained within a single codebase. This includes client-side logic, server-side aggregation, and updation of the global model. The evaluation of this architecture against the chosen criteria is as follows [30]:

- **Scalability:**
  - **Pros:** Initial deployment is simple, as everything is contained in one codebase.
  - **Cons:** As the system grows, scalability becomes a significant challenge. The tightly coupled components make it difficult to scale specific parts of the system independently.
- **Modularity:**
  - **Pros:** Simple to develop and deploy initially, with all logic in a single codebase.

  - **Cons:** Limited modularity which consequently leads to difficulties in managing and extending the system since any changes or updates require modifying and testing the entire codebase.

- **Maintainability:**
  - **Pros:** Easier to develop initially with fewer moving parts.
  - **Cons:** Over time, the monolithic nature makes maintenance tedious. Debugging and updating the system become more complex as the codebase grows.

### 4.3.3 Microservices architecture

The Microservices architecture decomposes the system into small, independent services that communicate with each other over a network. Each service handles a specific function [31]. In this scenario, this means that there is a service for each of the following local training, model aggregation, or model distribution.

- **Scalability:**
  - **Pros:** High scalability, as each service can be scaled independently based on demand. This flexibility allows for efficient resource utilization and better performance under varying loads.
  - **Cons:** Network overhead and potential latency issues due to the distributed nature of services.
- **Modularity:**
  - **Pros:** High modularity with clear separation of responsibilities. Each service can be developed, tested, and deployed independently.
  - **Cons:** Increased complexity in managing inter-service communication and ensuring system-wide consistency.
- **Maintainability:**
  - **Pros:** Easier to maintain, as updates to a specific service do not affect the entire system. This allows for continuous integration and deployment practices.
  - **Cons:** Requires robust monitoring and orchestration to manage multiple services effectively

The following table summarizes the evaluations of different architectural patterns based on key criteria—scalability, modularity, and maintainability [28–31]:

| **Criterion** | **Client-Server** | **Monolithic** | **Microservices** |
|---|---|---|---|
| **Scalability** | High client scalability, server bottlenecks | Limited scalability due to tight coupling | High scalability, each service scales independently |
| **Modularity** | Moderate, clear client-server separation | Low, single codebase | High, services are independent and focused |
| **Maintainability** | Moderate, easy client-server interactions | Low, complexity grows with the codebase | High, independent services simplify updates |

**Table 4.1: Summary of Architecture Evaluations**

Therefore, for federated learning systems, particularly in dynamic and collaborative environments like multiple airlines working together, the Microservices architecture stands out as the most suitable choice. It offers the highest scalability, modularity, and maintainability, which are critical for managing distributed clients, ensuring robust and efficient model training, and maintaining the system over time. The Client-Server architecture, while straightforward and moderately modular, may face challenges with server scalability. The Monolithic architecture, although easy to start with, becomes difficult to maintain and scale as the system grows in complexity. Thus, the Microservices approach provides a balanced solution for federated learning scenarios.

## 4.4 Prototype Implementation

The prototype implementation were done for both client-server and microservices architectures for federated learning. These architectures are designed to enable multiple distributed clients to train local machine learning models on their datasets and collaboratively aggregate these models to build a global model. Monolithic architecture was not considered for the prototype implementation due to its inherent limitations in scalability and modularity as can be seen in Table 4.1.

### 4.4.1 Client-server architecture

In this section, we present the implementation details of a client-server architecture tailored for federated learning. The architecture allows multiple distributed clients to train local machine learning models on their respective datasets and collaboratively aggregate these models to build a global model.

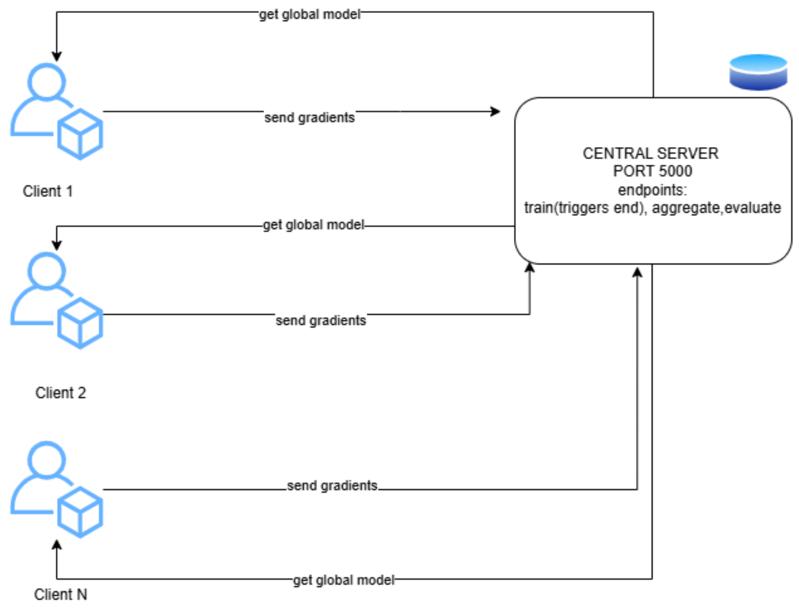


**Figure 4.2: Client-server implementation**

- **Server-Side Implementation**
  The server-side implementation utilizes the Flask framework to create a RESTful API, facilitating communication with multiple client devices. Key components and functionalities of the server-side code (server.py) are as follows:
  - **Aggregation Endpoint (/aggregate)**: Upon receiving model updates from clients, the server aggregates these updates to update the global model parameters (aggregate_models). This endpoint supports both sequential aggregation and final aggregation based on the specified mode.
  - **Final Aggregation Endpoint (/final_aggregate)**: This endpoint performs the final aggregation of client model parameters stored in client_parameters. It calculates the mean of model coefficients and intercepts across all participating clients and updates the global model accordingly.
  - **Model updation(/update_global_model)**: The server initializes a global machine learning model. This model serves as the starting point for aggregation, providing initial parameters that are updated as client models are aggregated.
  - **Model Retrieval (/get_global_model)**: This endpoint allows clients to retrieve the current global model parameters. It provides access to the model's coefficients and intercepts.
  - **Model Update (/set_global_model)**: This endpoint allows for updating the global model with new parameters. It receives updated model data from clients and sets the global model accordingly.
- **Client-Side Implementation**
  The client-side implementation (client.py) simulates multiple distributed clients participating in federated learning. Key functionalities and interactions include:

- **Data Preparation**: Clients prepare their local datasets (dev_data and test_data) stored in HDF5 format, which are then converted into Pandas DataFrames (df_train and df_test).
- **Model Training**: Clients train their local machine learning models using their respective training data (train_client). The locally trained models are then sent to the server for aggregation.
- **Final Model Evaluation**: Clients evaluate the global model (evaluate_global_model) on their test data to assess its predictive performance using evaluation metrics provided by the server.

### 4.4.2 Microservices Implementation

The microservices architecture in this federated learning setup is designed to separate concerns and allow scalable, independent deployment of different components. This section details the implementation of various services.

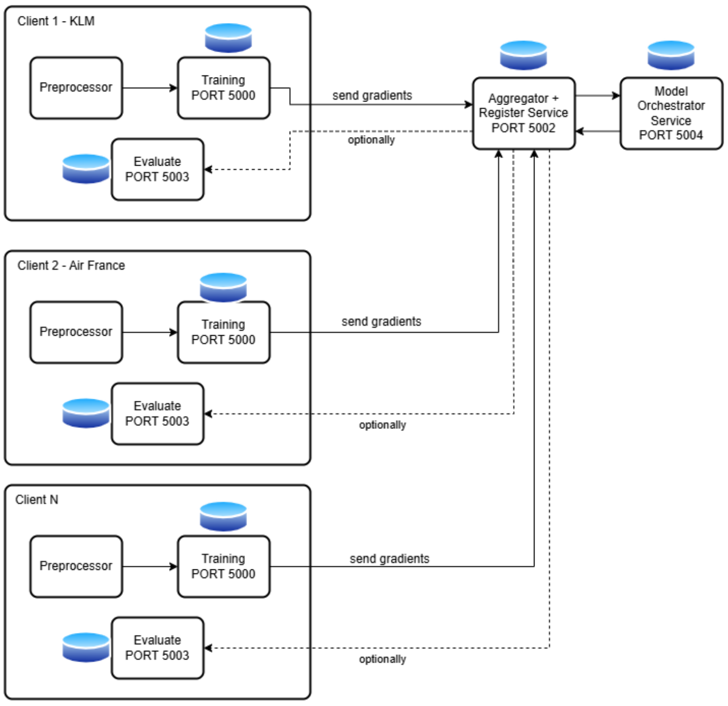


**Figure 4.3: Microservices implementation**

- **Model Service**
  The Model Service handles the storage and retrieval of the global model parameters. It provides two main endpoints:
  - An endpoint to get the current global model parameters.
  - An endpoint to set (update) the global model parameters.

  This service acts as a central repository for the global model, allowing other services to fetch and update the model as needed.
- **Training Service**
  The Training Service allows clients to train local models using their data and updates the model parameters. It performs the following key functions:
  - Fetches the current global model parameters from the Model Service.
  - Trains a local model using the client's data.
  - Returns the updated model parameters to the client.

  This service enables clients to contribute to the federated learning process by training on their local datasets and sharing the learned parameters for aggregation.

- **Aggregation Service** The Aggregation Service is responsible for collecting model updates from multiple clients and aggregating them to form the global model. It performs the following key functions:
    - Fetches the current global model parameters from the Model Service.
    - Aggregates client parameters by averaging them with the current global model parameters.
    - Updates the global model on the Model Service.

  This service ensures that the global model reflects the combined knowledge from all clients, improving its accuracy with each aggregation.
- **Registration Service**
  The Registration Service allows clients to register with the federated learning system. This is crucial for managing client participation and ensuring that updates are only accepted from registered clients. It performs the following key functions:
    - Registers new clients by storing their details.
    - Manages client status and participation in the federated learning process.

  This service ensures that the system maintains a registry of active clients, facilitating organized and controlled updates to the global model.
- **Evaluation Service**
  The Evaluation Service evaluates the global model using test data provided by the clients. It calculates and returns metrics such as Root Mean Squared Error (RMSE) and R-squared (R2) score. The evaluation process involves:
    - Fetching the global model parameters from the Model Service.
    - Using the global model to make predictions on the test data.
    - Calculating and returning evaluation metrics to assess the model's performance.

  This service provides critical feedback on the effectiveness of the global model, guiding further training and aggregation efforts.
- **Client Requests**
  The Client Requests script simulates client interactions with the federated learning system. It handles the following tasks:
    - Prepares the local dataset by converting it into the required format.
    - Sends training requests to the Training Service and receives updated model parameters.
    - Sends aggregated parameters to the Aggregation Service.
    - Evaluates the global model using the Evaluation Service and test data.

  This script orchestrates the end-to-end workflow of a federated learning cycle, from data preparation and local training to model aggregation and evaluation.

### Using Flask-RESTful

To implement these services efficiently, we utilize 'flask_restful' for creating and managing API endpoints. Flask-RESTful enhances the modularity of the codebase by allowing us to structure our endpoints as resources, making the code easier to maintain and extend.

### Benefits of using Flask-RESTful

The benefits of using Flask-RESTful include [32]:

- Modularity: Flask-RESTful encourages a modular design by allowing each API endpoint to be defined as a separate resource class. This modularity helps in managing large codebases by separating concerns and enabling easier maintenance. For instance, different functionalities like training, evaluating, and aggregating can each be encapsulated within their own resource classes.
- Clear routing: With Flask-RESTful, routing is handled cleanly and intuitively. Each resource class can be associated with one or more URL endpoints, and HTTP methods (GET, POST, etc.) can be mapped to class methods. This clear separation between routing logic and business logic simplifies the development process and reduces the likelihood of routing errors.

- Reusability and extensibility: Resources in Flask-RESTful can inherit from one another, promoting code reuse and extensibility. Common functionalities can be implemented in a base resource class and extended by more specific resource classes. This hierarchy allows for shared behavior and consistency across different API endpoints.

**Working of workflow**

In the context of our federated learning system, the workflow of interactions between the main API and the microservices is streamlined using Flask-RESTful. Here's a detailed look at how the training process works:

**Training Workflow**
**Client Request:**
A client application sends a POST request to the /train endpoint of the main API. The request includes the training data, formatted as JSON, with keys such as "X_train" for feature data and "y_train" for target data.

**Main API Handling:** The TrainingResource class within the main API processes this request. It uses request parsers to extract and validate the training data from the request. If the data is invalid, the API responds with an appropriate error message and status code.

**Forwarding to Training Service:** Upon successful validation, the main API forwards the training data to the training service by making a POST request to the /train endpoint of the training service. This communication occurs over the internal Docker network, using the URL http://training-service:5000/train.

**Training Service Processing:** The training service receives the training data, trains the machine learning model, and returns a response indicating the success of the training process or any errors encountered.

**Main API Response:** The main API receives the response from the training service and relays it back to the client. If the training was successful, the client receives a success message; otherwise, an error message is returned.
This systematic approach ensures that each step—from client request to microservice execution and subsequent client feedback—is handled seamlessly and efficiently, leveraging Flask-RESTful's robust framework for API development.
Similar workflows govern the other services within our federated learning system. Clients can interact with various endpoints (/evaluate, /register, /aggregate) to perform specific tasks, with each resource class managing the request flow and delivering relevant responses back to the client.

**Conclusion**

By using Flask-RESTful, our federated learning system achieves a structured and modular design. Each service operates as an independent microservice, and the main API coordinates interactions between them. This modularity enhances maintainability, scalability, and robustness, allowing for efficient handling of complex workflows such as training and evaluation in a federated learning environment.

### 4.4.3 Comparison of Client-Server and Microservices Architectures

The primary difference between the client-server and microservices based architectures lies in their design and deployment strategies, particularly in how they manage scalability, modularity, and service interactions.
**Client-Server based Architecture:** In a client-server based architecture, a single server handles all requests from multiple clients. The server performs all central functions, including model aggregation, retrieval, and updates. This architecture simplifies implementation but can become a bottleneck as the number of clients increases or the complexity of interactions grows. The server manages all client interactions through a RESTful API, providing endpoints for model aggregation, retrieval, and updates. This approach is straightforward but may face scalability and maintainability issues and can become a single point of failure.

**Microservices- based Architecture:** In contrast, the microservices-based architecture divides the application into multiple independent services, each responsible for a specific function. This setup includes separate services for model storage, training, aggregation, registration, and evaluation. Each service communicates via APIs and can be deployed independently, allowing for greater scalability and flexibility. For example, the Model Service handles model storage and retrieval, the Training Service manages local model training, and the Aggregation Service coordinates the combination of client updates. This architecture allows for scalability, enhances modularity and fault tolerance but introduces additional complexity in managing service interactions and deployments.

# Chapter 5

# Design Strategies

Enhancing federated learning models to efficiently handle large-scale and dynamic datasets requires careful consideration of several critical design decisions. The following design decisions were made in this federated learning set up :

### 5.0.1 Partitioning and Sampling Strategy

1. **Data Partitioning:** The original dataset from the HDF5 file is split into three parts. Each part contains a portion of the `dev_data`, maintaining the original order and preserving data integrity. This partitioning allows each client or node to train independently on its designated subset of the data.
2. **Data Preparation:** Clients prepare local datasets (`dev_data` and `test_data`) stored in HDF5 format, which are then converted into Pandas DataFrames (`df_train` and `df_test`).

### 5.0.2 Update Strategy

In federated learning, the update strategy is crucial to address the challenges posed by client drift. Client drift [33] occurs when the data distribution of a client's local dataset changes over time. This can lead to a divergence between the local models and the global model, impacting the performance and robustness of the federated learning system. To mitigate the effects of client drift and ensure effective model convergence, a well-designed update strategy is essential.

- **Local Training and Update:** Each participant of the FL process independently trains its local model using its dataset and immediately sends updates (parameters) to the server after training.
- **Server Aggregation:** Upon receiving updates from a client, the server aggregates these updates into the global model.
- **Model Deployment:** The updated global model is then deployed for evaluation or further training rounds.

### 5.0.3 Adaptive Learning and Transfer Learning

Adaptive learning and transfer learning are key strategies in our federated learning approach, aimed at optimizing model performance and enhancing knowledge transfer across distributed clients. These methods facilitate efficient model adaptation and convergence by leveraging pre-trained global parameters and incorporating local data insights. Clients begin their training process with models initialized using global parameters provided by the central server. This initialization process, known as transfer learning, enables clients to quickly adapt to their local datasets while retaining valuable knowledge from previous training cycles. As clients train on their local data, they continuously refine their models and contribute unique insights through iterative updates, which are then aggregated into the global model. The key components are:

- **Client-Side Training (`client.py`):** Simulates distributed clients participating in federated learning.
- **Local Model Training:** Each client trains a local model on its dataset (`train_client`), reflecting client-specific data characteristics.

The process details include:

1. **Transfer Learning Initialization:** Clients initialize their models with global parameters (`coef` and `intercept`) received from the server.
2. **Local Training:** Clients perform training on their local datasets (`X_train` and `y_train`), adapting the global model to their specific data distributions.
3. **Model Update:** After local training, clients send updated model parameters back to the server (`final_aggregate` endpoint) for aggregation into the global model.

### 5.0.4 Handling Disagreement between clients

Disagreement scenarios in federated learning occur when clients have non-cooperative relationship between stakeholders participating in the FL process. The design strategy incorporates mechanisms to detect, report, and resolve these disagreements effectively:
**Client Registration with Disagreement Information:** During registration, clients can specify other clients they disagree with. This information is stored and used during the aggregation phase to avoid conflicts.
**Disagreement Detection and Reporting:** Before aggregating model updates, the system checks for reported disagreements among clients. If a client has disagreements with others, its updates are excluded from the global model of the conflicted group. Instead, these updates are added to a separate global model specifically for clients with no disagreements among themselves.
**Client-Side Handling:** Each client records their data-sharing preferences during registration. This data is used to ensure that clients who have mutual agreements on sharing are grouped together. Clients who have mutual disagreements are placed in separate groups to avoid conflicts.
**Server-Side Handling:** The server creates a separate global model for each group of clients. Since clients in the same group agree on sharing data, their model updates are aggregated to form a global model specific to that group. This approach ensures that each global model reflects consistent and cooperative data from within its respective group, thus maintaining the integrity and effectiveness of the federated learning process across different client clusters.
**Dynamic group re-assignment:** To manage disagreements that arise between communication rounds, the system dynamically reassesses and reassigns client groups. If new disagreements emerge or existing agreements change, clients are reorganized into new groups accordingly. This dynamic reassignment helps to adapt to evolving client relationships and ensures that each group's global model remains representative of its members' consensus.

#### Illustration

The figure 5.1 considers a federated learning process involving three clients—Client 1, Client 2, and Client 3—where Client 1 and Client 3 have a mutual disagreement and do not wish to share data with each other, the system handles this by organizing clients into distinct groups based on their data-sharing preferences. Client 1 and Client 3 are placed in separate groups to avoid conflicts. Consequently, Client 1 and Client 2, who are willing to share data, form one group. Meanwhile, Client 3, who also has no issues with Client 2, forms a group with Client 2. Each group then independently aggregates the model updates from its members, resulting in distinct global models that accurately represent the consensus within each group.
During communication rounds, if conflicts arise or client preferences change, the system dynamically reassigns groups. Each group independently aggregates model updates, leading to distinct global models that accurately reflect the agreement within each group.
By incorporating dynamic group reassignment, we ensure that the federated learning system remains flexible and responsive to changing client relationships, thereby improving the robustness and effectiveness of the model training process.

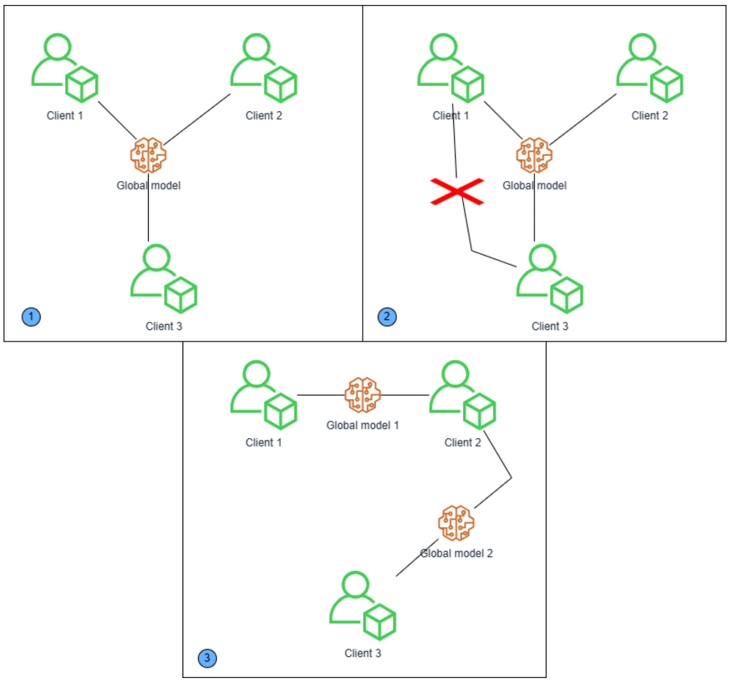
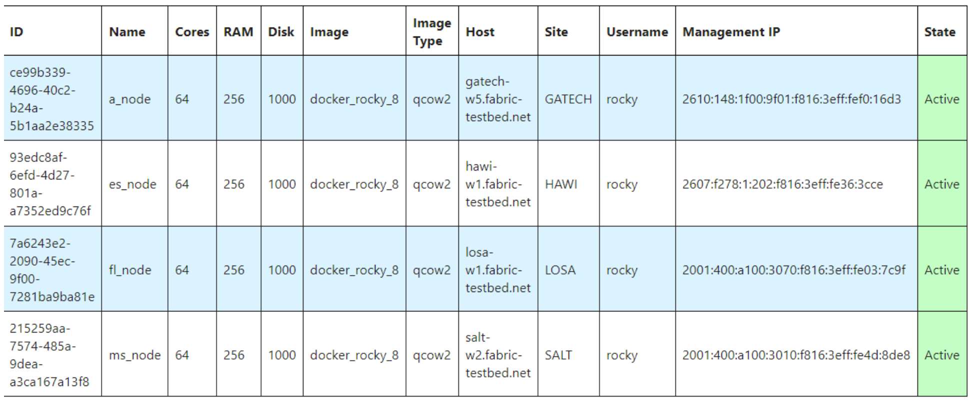


**Figure 5.1: Disagreement scenario**

### 5.0.5 Containerizing the application

The following steps outline the containerization process:

1. **Creating Docker Containers:** Each service within the federated learning system, such as the client training service, model aggregation service, evaluation service and main API server, was encapsulated within a Docker container. This encapsulation ensures that each service runs in its isolated environment with all necessary dependencies.
2. **Using Docker Compose:** Docker Compose was utilized to orchestrate the multiple containers required for the federated learning system. Docker Compose allows for defining and running multi-container Docker applications. This setup includes the configuration for how the services interact, network settings, and dependencies between services.
3. **Deployment with FABRIC Infrastructure:** Leveraging the FABRIC infrastructure the Docker containers were deployed across distributed environments. FABRIC provides robust infrastructure capabilities, enabling effective resource allocation and deployment of Dockerized services to enhance performance and scalability.
   (a) **Creating a Slice:** We began by creating a slice within the FABRIC infrastructure. A slice in FABRIC is a logical partition of resources that can be customized to meet specific needs. Using the FABRIC Portal, we defined a slice with sufficient compute, storage, and network resources to accommodate our Docker containers.
   (b) **Configuring the Slice:** The slice configuration involved specifying the required nodes and network settings. We selected nodes located in different geographical regions to enhance the redundancy and availability of our federated learning system. Each node was allocated resources based on the needs of the Docker containers it would host. This can be seen in the 5.2 below.
   (c) **Deploying Docker Containers:** With the slice configured, we deployed the Docker containers across the allocated nodes. Each container, encapsulating a service from our federated learning system, was assigned to a specific node within the slice. This deployment leveraged FABRIC's capabilities for resource allocation and network management, ensuring that the services ran efficiently and could communicate with each other as required.The deployment of the slice can be understood from the figure 5.3 representing the slice.



| ID | Name | Cores | RAM | Disk | Image | Image Type | Host | Site | Username | Management IP | State |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ce99b339-4696-40c2-b24a-5b1aa2e38335 | a_node | 64 | 256 | 1000 | docker_rocky_8 | qcow2 | gatech-w5.fabric-testbed.net | GATECH | rocky | 2610:148:1f00:9f01:f816:3eff:fef0:16d3 | Active |
| 93edc8af-6efd-4d27-801a-a7352ed9c76f | es_node | 64 | 256 | 1000 | docker_rocky_8 | qcow2 | hawi-w1.fabric-testbed.net | HAWI | rocky | 2607:f278:1:202:f816:3eff:fe36:3cce | Active |
| 7a6243e2-2090-45ec-9f00-7281ba9ba81e | fl_node | 64 | 256 | 1000 | docker_rocky_8 | qcow2 | losa-w1.fabric-testbed.net | LOSA | rocky | 2001:400:a100:3070:f816:3eff:fe03:7c9f | Active |
| 215259aa-7574-485a-9dea-a3ca167a13f8 | ms_node | 64 | 256 | 1000 | docker_rocky_8 | qcow2 | salt-w2.fabric-testbed.net | SALT | rocky | 2001:400:a100:3010:f816:3eff:fe4d:8de8 | Active |

**Figure 5.2: Slice configuration**

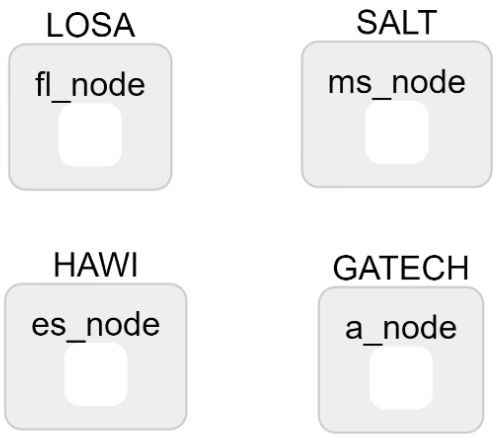


**Figure 5.3: Servers running docker containers**

(d) **Monitoring and Management:** Once the containers were deployed, we used FABRIC's monitoring tools to oversee the performance and health of the services. FABRIC provided insights into resource usage, network traffic, and overall system performance, allowing us to make adjustments as necessary to optimize the federated learning system's operation.

By utilizing FABRIC infrastructure to create and manage a dedicated slice, we ensured that our Dockerized services were deployed across a distributed and robust environment. This approach enhanced the performance, scalability, and reliability of our federated learning system, supporting its complex and distributed nature effectively.

### Why FABRIC?

The choice of FABRIC over other cloud service providers like GCP or Azure was influenced by several factors:

- **Consortium-Based Approach:** FABRIC is a consortium-based research infrastructure, which fosters collaboration among academic and research institutions. This aligns well with the goals of federated learning, where collaboration and data privacy are paramount.
- **Specialized Infrastructure:** FABRIC offers a highly customizable and flexible infrastructure tailored for experimental and cutting-edge research projects. This makes it an ideal platform for developing and testing novel federated learning techniques.
- **Cost Efficiency:** Using FABRIC can be more cost-effective for research projects compared to commercial cloud services, especially for long-term experiments and development cycles.
- **Performance and Scalability:** FABRIC provides robust resources and infrastructure capabilities, enabling efficient deployment and management of Dockerized services to enhance performance

and scalability as described in Section 2.0.4.

These factors made FABRIC a compelling choice for deploying and managing the federated learning system within this project.

## Evaluation based on Metrics

Based on the metrics defined for choosing architecture in Section 4.2 namely scalability, modularity and maintainability, the design strategies can be justified on how they support them:

### Partitioning and Sampling Strategy

- **Scalability:** The partitioning strategy enables horizontal scalability, as each client can handle a subset of the data independently. This scalability allows for efficient utilization of resources across distributed nodes or clients.
- **Modularity:** By separating the dataset into independent parts, the strategy promotes modularity in training and maintenance. Each client operates autonomously on its data subset, facilitating easier updates and modifications to individual components without affecting the entire system.
- **Maintainability:** The use of HDF5 format for storing and converting datasets into Pandas DataFrames enhances maintainability. It provides a standardized and efficient method for data handling, improving the readability and maintainability of the codebase.

### Update Strategy

- **Scalability:** By allowing independent client updates, the system can scale efficiently by adding or removing clients without impacting others, thereby enhancing overall scalability.
- **Modularity:** The modular design facilitates easier management and updates of individual client contributions and the global model, promoting a flexible and adaptable system architecture.
- **Maintainability:** With updates processed iteratively and independently, the system is easier to maintain as each component operates autonomously without complex dependencies on other clients.

### Adaptive Learning and Transfer Learning

- **Scalability:** The strategy supports scalability by allowing distributed training across multiple clients. Each client independently trains its model, reducing the load on the central server and enabling parallel processing.
- **Modularity:** The architecture is modular as it separates concerns between clients and the central server. Clients operate autonomously during local training, enhancing modularity and allowing for easy addition or removal of clients.
- **Maintainability:** The use of transfer learning facilitates maintainability by leveraging pre-trained global models. This approach reduces the need for retraining from scratch and ensures consistency across client updates.

### Disagreement Handling Strategy

- **Scalability:** The disagreement handling strategy supports scalability by allowing the system to continue functioning effectively even when some clients disagree with others. By detecting and managing disagreements at the client and server levels, the system can handle a larger number of clients with diverse and potentially conflicting data distributions without compromising the overall model performance.
- **Modularity:** The modular approach to handling disagreements ensures that each client can operate independently while still contributing to the global model. Disagreements are managed in a way that allows for modular updates and maintenance of client-specific logic without affecting the entire system. This separation of concerns enhances the flexibility and adaptability of the system architecture.

- **Maintainability:** The disagreement handling strategy improves maintainability by clearly defining mechanisms for detecting, reporting, and resolving conflicts. By storing and utilizing disagreement information during the aggregation phase, the system reduces the complexity of managing conflicting updates. This structured approach simplifies the maintenance of the codebase and ensures consistent handling of disagreements across different training rounds and client updates.

### 5.0.6 Containerizing the application

- **Scalability:** Docker containers support horizontal scalability by enabling flexible deployment of services across distributed nodes or clients. Each Docker container encapsulates specific functionality, allowing for efficient resource utilization and scalability as the system load varies.
- **Modularity:** Containerizing the application promotes modularity by isolating components into independent containers. This separation of concerns simplifies updates and maintenance, as each container operates autonomously without dependencies on other components.
- **Maintainability:** Docker containers improve maintainability by standardizing deployment and management processes. The use of Docker Compose for local orchestration and integration with FABRIC infrastructure ensures consistent deployment practices, enhancing the overall stability and maintainability of the federated learning system.

# Chapter 6

# Experimental Setup

## 6.1 Scalability Measurement

In order to evaluate the performance of client-server and microservices architectures with regards to scalability Apache JMeter, a widely-used tool for performance testing was used on FL process between two clients using the N-CMAPSS dataset. This setup aimed to provide insights into how each architecture handles simulated user traffic and to compare their performance across various metrics. It facilitated the creation of realistic testing scenarios and provided insights into how each architecture handles varying levels of simulated user traffic through HTTP requests [34]. This experimental setup is directly related to scalability because it tests how each architecture handles increasing loads and evaluates performance under simulated high-traffic conditions. The insights gained help determine the ability of each architecture to scale effectively and maintain performance as demands grow.

### 6.1.1 Test Environment

#### Infrastructure

The performance tests were conducted using a single laptop, which served as both the testing and simulation environment. The laptop was configured to run Apache JMeter and execute the performance tests for the client-server and microservices architectures. This setup simulated user interactions and system behavior under controlled conditions, with the laptop's hardware and network settings providing a consistent testing environment.

#### Software

Apache JMeter was utilized for load generation and performance measurement. The JMeter test plans were designed to simulate a range of user interactions with the client-server and microservices architectures through HTTP requests. Key metrics such as response times, throughput, and maximum latencies were collected and analyzed for comparative analysis.

### 6.1.2 Testing architecture

- Client server architecture: Implemented as a monolithic application where a single server instance handled all client requests directly.
- Microservices architecture: Configured as a set of independent services communicating through a central server, with tests conducted in two scenarios:
  - Non-disagreement scenario: Clients are willing to share data with one another and there are no conflicts.
  - Disagreement scenario: Clients have conflicts and are not willing to share their data with one or more clients.

### 6.1.3 Test configuration

#### JMeter Configuration

JMeter test plan was designed to simulate user interactions with the client-server and microservices architectures. The test plans included HTTP requests to the server or services, configured to replicate real-world usage patterns. JMeter listeners were used to collect performance metrics such as response times, throughput, and latency.

#### Test execution

The tests involved two clients performing a single round of federated learning and then finally the use of an evaluation service to evaluate the global model. This was done to ensure consistency and comparability across different test scenarios.

- Client-Server Tests: The tests involved simulating a controlled number of concurrent requests to a single server instance.
- Microservices Tests: These tests simulated interactions with multiple independent microservices. The performance was measured in two distinct scenarios:
  - Non-Disagreements Scenario: Both clients provided consistent updates to the global model, reflecting no conflicts. This scenario assessed how well the microservices architecture performed when client updates were harmonious.
  - Disagreements Scenario: The two clients provided conflicting updates to the global model, simulating conflicts. This scenario evaluated the impact of conflicting updates on performance metrics such as response time, throughput, and latency.

## 6.2 Maintainability and Modularity measurement

The experimental setup for maintainability and modularity measurement focused on evaluating the design and organization of the client-server and microservices architectures. This assessment aimed to provide insights into how modular and maintainable each architecture is by analyzing code metrics and structure.
To achieve this, the codebase was divided into two distinct project directories: one for the client-server architecture and one for the microservices architecture. Each directory contained the respective implementation, and various metrics were collected and compared using the Radon tool, which specializes in code metrics analysis [35]. Radon provides a comprehensive view of code quality, including maintainability index, cyclomatic complexity, and modularity metrics.

### 6.2.1 Codebase Organization

The codebase was organized into directories as follows:

- **Client-Server Architecture Directory:** This directory included the implementation of the client-server architecture of the FL process.
- **Microservices Architecture Directory:** This directory contained the implementation of the microservices architecture, where the code was organized into a set of independent services communicating through a central server.

### 6.2.2 Metrics Collection with Radon

Radon was employed to measure and compare various code metrics, including:

- **Cyclomatic Complexity:** This metric evaluates the complexity of the code by measuring the number of linearly independent paths through the code. Lower complexity values suggest more modular and easier-to-understand code.
- **Code Metrics:** Radon provides insights into the modularity of the code, helping to evaluate how well the code is implemented and documented.
- **Maintainability Index:** This metric assesses the ease with which code can be understood, modified, and extended. A higher maintainability index indicates more maintainable code.

# Chapter 7

# Results

In this chapter, we present the findings from our experiments. The first section presents the results related to scalability metrics, while the second section presents the outcomes concerning maintainability and modularity metrics.

## 7.1 Scalability/Performance Results

This testing was done with simulating two clients in a federated learning process. In evaluating the performance of both architectures, Apache JMeter was instrumental. It served as a robust tool for performance measurement, providing insights into how each architecture handles varying levels of simulated user traffic. Apache JMeter facilitated the creation of realistic testing scenarios, simulating multiple concurrent users interacting with the applications through HTTP requests [34]. During testing, Apache JMeter collected essential metrics such as response times, throughput and maximum latencies for both client-server and microservices architectures.
Response Time measures how quickly the system responds to a request; lower values are desirable as they indicate faster performance. Throughput reflects the number of requests processed per second; higher throughput signifies the system's ability to handle more requests efficiently. Latency is the delay before a response is received; lower latency is preferable, as high latency can indicate delays and inefficiencies in processing requests. Overall, good performance is characterized by low response times and latency combined with high throughput, indicating that the system is both fast and capable of handling a large volume of requests efficiently.
Based on the metrics gathered, Apache JMeter enabled a comparative analysis of the architectures. The results are visualized in Figure 7.1, which shows the mean response time, throughput, and latency with error bars for each architecture.

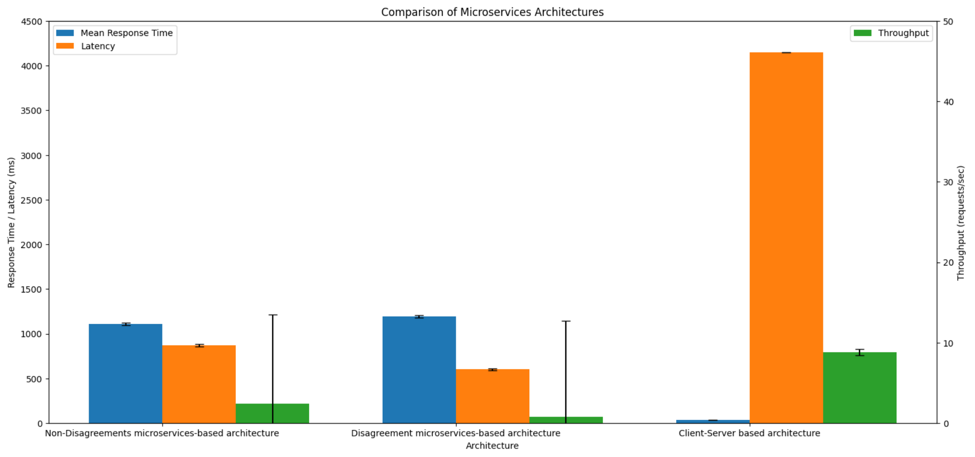


**Figure 7.1: Mean Response Time, Throughput, and Latency with Error Bars for Different Architectures.**

### 7.1.1 Client-Server Architecture

- Average Response Time: 36.9 ms
- Throughput: 8.8 req/sec
- Max Latency: 4152.3 ms
- Runtime: 73 ms

The client-server architecture showcases impressive performance, with a notably low average response time and high throughput. However, it is important to address the relatively high maximum latency observed. This increased latency can be attributed to the direct client-server communication model, which, while effective for predictable workloads, may struggle under peak loads or in large-scale operations.
A contributing factor to this issue could be the single-threaded nature of the Flask framework used for server implementation. By default, Flask processes requests sequentially, which can lead to bottlenecks and increased latency when managing high volumes of concurrent requests. This limitation becomes evident in scenarios with substantial load, where the server's ability to efficiently handle multiple simultaneous requests is compromised.

### 7.1.2 Microservices Architecture

**Non-Disagreements Scenario**

- Average Response Time: 1108.9 ms
- Throughput: 2.4 req/sec
- Max Latency: 869.9 ms
- Runtime: 1569.2 ms

**Disagreement Scenario**

- Average Response Time: 1192.0 ms
- Throughput: 0.8 req/sec
- Max Latency: 601.5 ms
- Runtime: 1839.4 ms

The microservices architecture shows distinct performance characteristics between scenarios with and without client disagreements. In scenarios without disagreements, the architecture maintains moderate response times and throughput, but it still faces a higher average response time compared to the client-server model. When disagreements occur, both response time and throughput degrade further, reflecting the added complexity of inter-service communication and model aggregation processes.

## 7.2 Maintainability and modularity results

To assess the maintainability and modularity of both architectures, we employed Radon, a Python library for code analysis. Radon provides valuable metrics such as cyclomatic complexity and maintainability index, which are critical for understanding the code's complexity and modularity.

### 7.2.1 Cyclomatic Complexity

Cyclomatic complexity is a measure of the number of linearly independent paths through a program's source code. It provides an indication of the code's complexity and potential maintainability issues [36]. Cyclomatic complexity helps to assess modularity indirectly as well. While it doesn't measure modularity directly, it can indicate whether the code is broken down into manageable, single-purpose modules. By analyzing cyclomatic complexity, you can infer the degree of modularity in a codebase and identify areas where modularity can be improved. Lower complexity generally aligns with better modularity, reflecting well-structured and maintainable code [37]. The cyclomatic complexity for the both architectures are as follow:

**Client-Server Architecture**

The cyclomatic complexity for the client-server architecture is as follows:

- **Clients (clients.py)**: Complexity ranges from A (2) to A (2).

- **Server (server-fl.py)**: Complexity ranges from A (1) to A (3).

**Microservices-based Architecture**

- **Aggregation Service (aggregation-service.py)**: Complexity ranges from B (6) to B (10).
- **Client Requests (client-requests.py)**: Complexity ranges from A (2) to A (5).
- **Evaluate Service (evaluate-service.py)**: Complexity A (2).
- **Main API (main-api.py)**: Complexity ranges from A (1) to A (3).
- **Model Service (model-service.py)**: Complexity A (1).
- **Training Service (training-service.py)**: Complexity A (2).

### 7.2.2 Code Metrics

Code metrics, including Lines of Code (LOC), Source Lines of Code (SLOC), and comment density, offer insights into the size and documentation of the codebase. These metrics help us gauge the extent of code implementation and its documentation quality. Effective modularity is often associated with a manageable LOC and a well-documented codebase, which supports easier understanding and modification of the code [37]. Below there is the results of the code metrics for both architectures:

**Client-Server Architecture**

| File | LOC | SLOC | Comments | Comment Density (%) |
|---|---|---|---|---|
| Clients | 106 | 83 | 11 | 10% |
| Server | 144 | 102 | 13 | 9% |

**Table 7.1: Code Metrics for Client-Server Architecture**

**Microservices-based Architecture**

| File | LOC | SLOC | Comments | Comment Density (%) |
|---|---|---|---|---|
| Aggregation Service | 130 | 95 | 10 | 8% |
| Client Requests | 107 | 78 | 18 | 17% |
| Evaluate Service | 42 | 30 | 5 | 12% |
| Main API | 98 | 66 | 6 | 6% |
| Model Service | 23 | 19 | 0 | 0% |
| Training Service | 66 | 43 | 12 | 18% |

**Table 7.2: Code Metrics for Microservices-based Architecture**

### 7.2.3 Maintainability Index

The Maintainability Index (MI) is a composite metric derived from cyclomatic complexity, Halstead volume, and lines of code. It provides a score that reflects the ease with which the code can be maintained. A higher maintainability index indicates more maintainable code, suggesting that the code is easier to understand, modify, and extend. This index supports evaluating the modularity of the codebase, with a higher score generally indicating better modularization and code quality. The formula used to calculate the maintainability index is [35]:

$$\text{MI} = \max\left[0, 100\left(171 - 5.2\ln(V) - 0.23G - 16.2\ln(L) + 50\sin(\sqrt{2.4C})\right)/171\right]$$

Where:
V is the Halstead Volume (see below);
G is the total Cyclomatic Complexity;

L is the number of Source Lines of Code (SLOC);
C is the percent of comment lines (important: converted to radians).

The Maintainability Index for both architectures is presented in the following sections:

### Client-Server Architecture

| **File** | **Maintainability Index** |
|---|---|
| Clients (clients.py) | A |
| Server (server-fl.py) | A |

**Table 7.3: Maintainability Index for Client-Server Architecture**

### Microservices architecture

| **File** | **Maintainability Index** |
|---|---|
| Aggregation Service (aggregation-service.py) | A |
| Client Requests (client-requests.py) | A |
| Evaluate Service (evaluate-service.py) | A |
| Model Service (model-service.py) | A |
| Training Service (training-service.py) | A |

**Table 7.4: Maintainability Index for Microservices-based Architecture**

# Chapter 8

# Discussion

In this chapter, we begin by discussing the various observations made from the results listed in Chapter 7. Afterwards we answer the research questions listed in Chapter 1.

## 8.1 Reasoning behind using N-CMAPSS Dataset

The research for this thesis was conducted in collaboration with data scientists and engineers at KLM Royal Dutch Airlines. Initially, the aim was to use an in-house dataset from KLM, the cooling dataset. However, the complexity of this dataset posed a significant challenge given the time constraints of the project. Both the KLM dataset and the N-CMAPSS dataset are time-series datasets, generating sensor data over the duration of a flight, making them suitable for developing predictive maintenance models. Due to the high industry standards and the similarity to the KLM dataset, the scientists and engineers at KLM recommended using the N-CMAPSS dataset for this research.

## 8.2 RQ1: How can software architecture patterns be tailored to accommodate federated learning systems?

In comparing the client-server and microservices architectures within the context of federated learning, several key observations and insights emerge from the results presented in Chapter 7.

#### Response Time and Throughput

**Client-Server Architecture:**The client-server architecture demonstrates remarkable performance with an average response time of 36.9 ms, high throughput of 8.8 req/sec, and a maximum latency of 4152.3 ms. These metrics highlight the efficiency of the client-server model, which benefits from direct communication between clients and servers. This architecture based design minimizes overhead and effectively manages concurrent requests, making it well-suited for applications with predictable workloads where fast response times and high throughput are critical.
**Microservices Architecture:** In contrast, the microservices architecture exhibits significantly higher average response times in both scenarios compared to the client-server model. Specifically, the non-disagreements scenario shows an average response time of 1108.9 ms, and the disagreement scenario shows an even higher average response time of 1192.0 ms. Throughput is also lower in the microservices model, with 2.4 req/sec in the non-disagreements scenario and 0.8 req/sec in the disagreement scenario. Maximum latency values are moderate in the non-disagreements scenario at 869.9 ms and lower in the disagreement scenario at 601.5 ms.
These differences are primarily attributed to the added complexity of inter-service communication in the microservices architecture. The need for multiple service calls to fulfill a single request can increase response times and reduce throughput. Despite these performance challenges, microservices offer advantages in terms of modular development and deployment flexibility. The ability to independently scale services helps optimize resource utilization and adapt to varying operational needs, even though the initial performance metrics are less favorable compared to the client-server model.
In the non-disagreements scenario, clients have aligned objectives and similar data distributions, leading to more consistent and harmonious local model updates. This consistency facilitates several advantages:

lower average response time, higher throughput, and moderate maximum latency. With all clients working towards a single goal, one global model, the aggregation of their updates results in a more accurate and stable global model. This accuracy translates to faster response times when the model is deployed and queried, as the predictions are more reliable and require less computational overhead. The efficient aggregation of consistent updates means the system can handle more requests per second, reflecting increased throughput. The relatively lower maximum latency suggests that the system experiences fewer extreme delays, likely due to the smoother integration of client updates.
Conversely, in the microservices based disagreement scenario, clients have divergent objectives and the FL process is done to train more than one global model (one model for each group), leading to more variability in their local model updates. This disagreement introduces complexities such as higher average response time, lower throughput, and lower maximum latency. The central server must perform more aggregation processes from various clients. This often requires additional computation and more sophisticated algorithms, which can increase the overall response time of the system. The need to handle and mitigate conflicts between clients slows down the system's ability to process requests efficiently, indicating reduced throughput.

**Finding 1:** Client-server architecture shows lower response time with high throughput, whereas microservices architecture shows higher response time and lower throughput.

### Error Handling and Resilience

In the context of performance evaluation for client-server and microservices architectures, error rate is a critical metric that measures the frequency of errors occurring during the execution of requests or operations. Specifically, it calculates the proportion of failed requests or transactions relative to the total number of requests made to the system. A lower error rate indicates a more reliable system with fewer failures, which is crucial for maintaining consistent service quality and user satisfaction. In contrast, a higher error rate suggests more frequent issues, which could impact the system's overall performance and user experience.
Both architectures exhibit low error rates as can be seen in Appendix A, with the microservices architecture marginally higher due to its distributed nature. Microservices provide better fault isolation, where failures in one service do not necessarily affect others, enhancing overall system resilience compared to monolithic client-server applications.

**Finding 2:** They both show low error rates but microservices architecture offers better fault isolation and resilience compared to client-server architectures.

### Scalability and Flexibility

**Client-Server architecture:** Limited scalability due to its monolithic structure. Scaling often involves replicating the entire application, which may be less flexible and more resource-intensive compared to microservices.
**Microservices architecture:** While initial performance metrics may appear poorer, microservices excel in scalability. Individual services can be scaled independently based on demand, optimizing resource utilization and adapting to changing operational needs.

**Finding 3:** Client-server architecture has limited scalability and flexibility, whereas microservices excel in scalability by allowing independent scaling of services based on demand.

### Cyclomatic Complexity

**Client-Server Architecture:** The client-server architecture demonstrates consistently lower cyclomatic complexity across its functions, which are primarily categorized as A levels. This indicates simpler function logic with fewer conditional paths, making the codebase easier to understand and maintain. The lower complexity reflects the more straightforward nature of the monolithic design, where interdependencies are more localized.

**Microservices Architecture:** The microservices architecture shows a broader range of cyclomatic complexity, with some functions exhibiting higher complexity levels, particularly in areas involving service coordination and inter-service communication. This complexity is indicative of the more intricate logic required to manage multiple interacting services and handle various distributed system concerns. Functions with higher complexity (B and above levels) often involve more conditional paths and error handling, making them more challenging to maintain.

**Finding 4:** The microservices architecture has higher average cyclomatic complexity compared to the client-server architecture.

#### Code metrics

The client-server architecture has fewer lines of code overall, with a moderate comment density. The simpler structure and lower number of lines might contribute to easier maintenance.
The microservices-based architecture generally has more lines of code and source lines of code, with varying comment densities. Files like ‘client-requests.py‘ and ‘training-service.py‘ have higher comment percentages, which can aid in understanding and maintaining the code. However, files such as ‘model-service.py‘ have no comments, which could impact maintainability negatively.

**Finding 5:** The microservices architecture has a larger and more varied codebase, with some files lacking comments, which may affect maintainability. whereas the client-server architecture, with its smaller codebase and moderate comment density, generally presents a more straightforward maintenance profile.

#### Maintainability index

The Maintainability Index analysis reveals that both the Microservices-based and Client-Server architectures have high maintainability ratings. Each file in both architectures received an A, suggesting excellent maintainability across the board. This high rating reflects well on the quality and organization of the codebases in both architectural styles.

**Finding 6:** Both architectures exhibit high Maintainability Index scores, indicating excellent maintainability.

Based on these findings, the choice between client-server and microservices architectures depends upon specific project requirements and business objectives. Client-server architecture is favored for applications with predictable workloads and latency requirements, benefiting from its efficient handling of concurrent requests. On the other hand, microservices architecture offers scalability, fault isolation, and flexibility, making it suitable for dynamic environments requiring adaptable and resilient systems. Organizations should weigh these factors carefully to align architectural decisions with long-term operational goals and technical requirements.

**Finding 7:** In this study since we define the metrics for validation to be scalabililty, modularity and maintainability, microservices architecture is the better option.

## 8.3 RQ2: What key design decisions need to be made in order to facilitate federated learning models

The design strategies employed in our federated learning setup effectively address critical challenges such as:
**Data Partitioning and Sampling Strategy:**
Impact: Efficient data partitioning ensures that each client trains on relevant subsets of the data while maintaining data integrity and order. This strategy enables parallel processing and scalable model

training across distributed clients. By leveraging local datasets (dev_data and test_data), it optimizes computational resources and supports diverse data characteristics, enhancing overall model accuracy and performance in predictive maintenance tasks. However, challenges arise in the practical implementation of data partitioning:

- Data Skewness: In real-world scenarios, data may be skewed across clients, meaning some clients may have much more or less data than others. This imbalance can lead to biased model training, where models may not generalize well across the entire dataset. Addressing data skewness often requires advanced techniques to balance the data or adapt the model training process to handle skewed distributions effectively.
- Model Poisoning: The risk of model poisoning is another significant challenge. In federated learning, malicious clients might intentionally introduce biased or incorrect data to degrade the performance of the global model. Ensuring data integrity and robust aggregation methods is essential to mitigate the effects of model poisoning and maintain the reliability of the federated learning system.

**Update Strategy:**
Impact: The update strategy facilitates seamless collaboration among distributed clients by aggregating model updates while preserving consistency and accuracy of the global model. Secure communication channels ensure data integrity during transmission, fostering trust and compliance with aviation data privacy regulations. Periodic aggregation of client models with updated global parameters mitigates client drift, ensuring continuous model improvement and adaptability over successive training rounds.
**Adaptive Learning and Transfer Learning:**
Impact: Adaptive and transfer learning mechanisms accelerate model convergence and performance optimization across diverse client environments. By initializing models with pre-trained global parameters, clients reduce the computational burden and expedite adaptation to local datasets. This approach enhances knowledge transferability and leverages accumulated insights from previous training cycles, promoting efficient learning and contributing to robust predictive maintenance models in aviation.
**Handling Disagreements:**
Impact: Effective management of client disagreements ensures collaborative model aggregation without compromising model integrity. By recording and addressing client-specific disagreements, the system promotes fairness and transparency in model updates. Predefined rules for resolving conflicts during aggregation maintain consensus among participants, fostering cooperative learning and reliable predictive maintenance outcomes across heterogeneous client networks in aviation.
It is important to note that we did not implement a client-server architecture for the disagreement scenario. In a client-server setup, clients with divergent objectives could potentially disconnect or disengage if they were required to participate in a single global model training. By opting for a microservices-based approach in this scenario, we ensure that clients can work towards separate global models, accommodating their individual objectives without compromising data integrity.
**Containerization:**
Impact: Docker containerization enhances system agility, scalability, and resource efficiency in deploying federated learning services. By encapsulating client training, model aggregation, and API servers within isolated containers, the system ensures consistent execution environments across distributed deployments. Docker Compose orchestrates interactions between containers, facilitating seamless integration with infrastructure frameworks like the FABRIC testbed. This approach simplifies deployment, maintenance, and scaling of federated learning solutions in aviation, supporting dynamic workload management and operational flexibility.

This research underscores the significance of strategic design decisions in developing scalable, modular, and maintainable federated learning systems. By addressing challenges related to data partitioning, update strategies, adaptive learning, disagreement handling, and containerized deployment, the study provides practical insights for implementing federated learning in real-world applications.

## 8.4 Threats to validity

The following could affect the validity of the research in this thesis:

### 8.4.1 Use of evaluation metrics

Using evaluation metrics to choose an architecture poses a threat to validity by potentially introducing bias towards specific performance criteria while overlooking broader architectural considerations. This approach may lead to architectures optimized for narrow metrics in controlled environments that fail to address critical real-world challenges like scalability, adaptability, and regulatory compliance in aviation predictive maintenance. Moreover, focusing solely on quantitative metrics risks neglecting qualitative factors, stakeholder priorities, and complex trade-offs necessary for robust architectural design. To mitigate these risks, a balanced evaluation approach integrating both quantitative metrics and qualitative assessments is essential to ensure that architectural decisions align with operational realities and stakeholder needs across diverse aviation environments.

### 8.4.2 Performance metrics limitations

The performance metrics collected, such as response time, throughput, and latency, provide valuable insights but may not encompass all aspects of federated learning performance. Metrics related to model accuracy, convergence rates, and communication efficiency might need further exploration to provide a more comprehensive evaluation. For instance, the simulation was conducted with only two clients with a single dataset partitioned into parts, which is a simplified representation of real-world scenarios where federated learning typically involves a larger number of clients. In practice, federated learning systems must scale to handle hundreds or thousands of clients, each contributing diverse and potentially heterogeneous data.

### 8.4.3 Testing environment

The performance metrics were obtained in a controlled testing environment using Apache JMeter. Real-world conditions, such as varying network latencies, server loads, and hardware configurations, may differ significantly and impact the observed performance metrics. The results may not fully reflect the performance of the architectures in diverse production environments.

### 8.4.4 Design decision impacts

The design decisions, including data partitioning and handling disagreements, play a crucial role in the system's performance. While these strategies address key challenges, they may introduce specific limitations or trade-offs that affect overall effectiveness. For instance, while the aggregation strategy helps reduce wait time, it may also lead to additional computational overhead. Balancing these trade-offs is essential for optimizing federated learning systems.

# Chapter 9

# Related work

Federated learning has emerged as a compelling paradigm in the machine learning domain, enabling collaborative model training across decentralized devices while maintaining data privacy. This chapter reviews key works related to federated learning, focusing on its foundational concepts and applications across various fields.

## 9.1 Foundational Concepts and Algorithms

The concept of federated learning was first introduced by McMahan et al. (2017) with the development of the Federated Averaging (FedAvg) algorithm[38]. This work laid the groundwork for training deep neural networks across decentralized devices with minimal communication costs. FedAvg achieves this by averaging model updates from multiple clients, which significantly reduces the need for frequent data exchanges between clients and the central server.
Building on this foundation, several variations and improvements to the FedAvg algorithm have been proposed. For instance, [39] addressed the issue of non-IID (non-independent and identically distributed) data across clients by introducing data-sharing strategies to mitigate the impact of skewed data distributions on model performance. Similarly [6] proposed the FedProx algorithm, which adds a proximal term to the local objective function to stabilize training when dealing with heterogeneous client data and computational resources.

## 9.2 Applications and Case Studies

Federated learning has found applications in a wide range of domains, from healthcare and finance to mobile edge computing and the Internet of Things (IoT). The table below summarizes the concepts explored in the related works.

| Citations | Cloud | FL | industry | scalability | privacy | data sharing |
|---|---|---|---|---|---|---|
| [40] | ✓ | ✓ | Healthcare | - | ✓ | - |
| [41] | ✓ | ✓ | - | - | ✓ | - |
| [42] | - | ✓ | Aviation | - | ✓ | - |
| [43] | ✓ | ✓ | - | ✓ | ✓ | - |
| [44] | - | ✓ | - | - | ✓ | ✓ |
| [45] | - | ✓ | Healthcare | - | ✓ | - |
| [9] | - | ✓ | Finance | - | ✓ | - |
| [46] | - | ✓ | Mobile Edge Computing | - | ✓ | - |
| [47] | - | ✓ | Healthcare | - | ✓ | - |
| [48] | - | - | PdM | - | ✓ | ✓ |

**Table 9.1: Related works categorization**

# Chapter 10

# Conclusion

In this work, we have explored federated learning, focusing on its application in scenarios requiring robust, decentralized, and privacy-preserving machine learning models. Our research looked into the design and implementation of a federated learning system using a microservices-based architecture, leveraging Flask for API management and Docker for containerization. We addressed the challenges of model training, evaluation, and aggregation in a distributed environment, emphasizing the importance of efficient communication protocols and scalable deployment strategies.
We began by discussing the fundamental concepts of federated learning, highlighting its potential to transform data-intensive industries by enabling collaborative model training without compromising data privacy. Our work on federated learning architectures, particularly in the context of the aviation maintainance industry, demonstrated the viability of this approach in real-world applications. The system we designed showed promise in handling large-scale deployments, managing decentralized model updates, and ensuring efficient data exchanges between clients and servers.
The experimental setup and results provided valuable insights into the scalability, modularity and maintainability of federated learning systems. However, we also identified several limitations and areas for improvement which are highlighted in 10.1.

## 10.1 Future work

### 10.1.1 Validation of Data and Aggregation Impurity

One critical area for future research is the validation of data and the mitigation of aggregation impurity in federated learning systems. Ensuring the quality and consistency of data across distributed clients is paramount. In future studies, methods for verifying the integrity of data before and after aggregation will be explored. Techniques to detect and correct data inconsistencies or impurities during the aggregation process will be developed, aiming to improve the reliability and performance of the global model. Additionally, addressing challenges such as data skewness and model poisoning will be crucial. Data skewness, where the distribution of data across clients is imbalanced, can affect the accuracy of the global model. Model poisoning, where malicious clients intentionally introduce erroneous data, can compromise the integrity of the federated learning process. Developing robust strategies to handle these issues will be important for maintaining a high-quality global model.

### 10.1.2 Reverse Engineering of Global Model Updates

Another promising direction for future work is investigating the security implications of reverse engineering global model updates from local training. Understanding the potential vulnerabilities in the federated learning process can help in developing more robust security measures. Future research will focus on analyzing the feasibility of extracting sensitive information from global model updates and devising strategies to prevent such security breaches. This will enhance the privacy-preserving aspects of federated learning and protect client data from potential adversaries.

### 10.1.3 Networking and Deployment in Different Data Centers

Future research will also explore the networking and deployment of federated learning systems across different data centers, particularly leveraging the FABRIC testbed infrastructure. Establishing efficient communication protocols and optimizing the networking setup will be crucial for scaling federated learning applications. This involves testing various networking configurations and evaluating their impact on system performance and reliability. Deploying federated learning systems in multiple data centers will also provide insights into handling latency, fault tolerance, and load balancing in a distributed environment, ultimately leading to more robust and scalable federated learning architectures.

### 10.1.4 Use of Kubernetes for Fault Tolerance

Future work will include the exploration of Kubernetes (K8s) for enhancing fault tolerance in federated learning systems [10]. Kubernetes offers robust features for managing containerized applications, such as automated deployment, scaling, and self-healing mechanisms. Research will focus on leveraging Kubernetes to ensure high availability and reliability of federated learning processes by:

- **Automated Scaling:** Dynamically scaling the number of client instances based on workload to maintain optimal performance.
- **Self-Healing:** Automatically detecting and replacing failed nodes or pods to minimize downtime and maintain continuous operation.
- **Load Balancing:** Efficiently distributing workloads across multiple nodes to prevent bottlenecks and ensure balanced resource utilization.
- **Deployment Strategies:** Implementing rolling updates and canary deployments to ensure seamless updates and minimal disruption during new model deployments.

Implementing Kubernetes for fault tolerance will enhance the robustness of federated learning systems, ensuring they can handle failures gracefully and maintain consistent performance in diverse environments.

### 10.1.5 Model Versioning for Federated Learning

It is crucial to consider model versioning in federated learning systems. Model versioning will allow for tracking and managing different iterations of the global model throughout its lifecycle. By implementing a versioning system in the future, we can ensure that previous models are accessible for comparison, rollback, or further analysis by the clients.

# Acknowledgements

I would like to express my heartfelt gratitude to my thesis supervisor, Dr. Ana Oprescu, for her invaluable support and guidance throughout the course of my thesis. I am also deeply appreciative of Leon Gommans and Jeroen Mulder from KLM for their insightful feedback and the opportunity to collaborate with KLM during my research.
Furthermore, I would like to thank all data scientists and engineers at KLM who provided me with their assistance in understanding the various concepts essential to complete this thesis. Additionally, I am grateful to my colleagues at the University of Amsterdam for their constructive feedback and support throughout my master's program.
Lastly, I would like to thank my parents for their unwavering support throughout my academic journey.

# Bibliography

[1] D. Negi and P. Jaiswal, "Technological advancement shaping the future of aviation," *International Journal of Research Publication and Reviews*, vol. 5, no. 4, pp. 5348–5351, Apr. 2024. DOI: `https://doi.org/10.55248/gengpi.5.0424.1064`.

[2] J. B. Simon, D. Karkada, N. Ghosh, and M. Belkin, "More is better in modern machine learning: When infinite overparameterization is optimal and overfitting is obligatory," *arXiv (Cornell University)*, Jan. 2023. DOI: `https://doi.org/10.48550/arxiv.2311.14646`.

[3] EASA, *Easa artificial intelligence roadmap 2.0 - a human-centric approach to ai in aviation*, May 2023. [Online]. Available: `https://www.easa.europa.eu/en/document-library/general-publications/easa-artificial-intelligence-roadmap-20`.

[4] P. Korvesis, "Machine learning for predictive maintenance in aviation," Ph.D. dissertation, Nov. 2017.

[5] "Data for aviation." (), [Online]. Available: `https://dataforaviation.org`.

[6] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020. DOI: `10.1109/MSP.2020.2975749`.

[7] J. Konečný, H. B. Mcmahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv (Cornell University)*, Oct. 2016. DOI: `https://doi.org/10.48550/arxiv.1610.05492`.

[8] D. Mahlool and M. Abed, *A comprehensive survey on federated learning: Concept and applications*, Jan. 2022.

[9] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[10] M. Sabuhi, P. Musilek, and C.-P. Bezemer, "Micro-fl: A fault-tolerant scalable microservice-based platform for federated learning," *Future Internet*, vol. 16, no. 3, pp. 70–70, Feb. 2024. DOI: `https://doi.org/10.3390/fi16030070`.

[11] S. Abdulrahman, H. Tout, H. Ould-Slimane, A. Mourad, C. Talhi, and M. Guizani, "A survey on federated learning: The journey from centralized to distributed on-site learning and beyond," *IEEE Internet of Things Journal*, vol. PP, Oct. 2020. DOI: `10.1109/JIOT.2020.3030072`.

[12] S. Tyagi, I. S. Rajput, and R. Pandey, "Federated learning: Applications, security hazards and defense measures," Mar. 2023. DOI: `https://doi.org/10.1109/dicct56244.2023.10110075`.

[13] B. Nagy *et al.*, "Privacy-preserving federated learning and its application to natural language processing," *Knowledge-Based Systems*, vol. 268, p. 110 475, 2023, ISSN: 0950-7051. DOI: `https://doi.org/10.1016/j.knosys.2023.110475`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0950705123002253`.

[14] W. Yang, N. Wang, Z. Guan, L. Wu, X. Du, and M. Guizani, "A practical cross device federated learning frame-work over 5g n etworks," *IEEE Wireless Communications*, pp. 1–8, 2022. DOI: `https://doi.org/10.1109/mwc.005.2100435`.

[15] C. Huang, J. Huang, and X. Liu, "Cross-silo federated learning: Challenges and opportunities," *arXiv preprint arXiv:2206.12949*, 2022.

[16] A. Nilsson, S. Smith, G. Ulm, E. Gustavsson, and M. Jirstrand, "A performance evaluation of federated learning algorithms," *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*, Dec. 2018. DOI: `https://doi.org/10.1145/3286490.3286559`.

[17] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao, “A survey on federated learning,” *Knowledge-Based Systems*, vol. 216, p. 106 775, 2021, ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2021.106775. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705121000381.

[18] [Online]. Available: https://www.computer.org/resources/software-architecture.

[19] M. Söylemez, B. Tekinerdogan, and A. Kolukısa Tarhan, “Challenges and solution directions of microservice architectures: A systematic literature review,” *Applied Sciences*, vol. 12, no. 11, 2022, ISSN: 2076-3417. DOI: 10.3390/app12115507. [Online]. Available: https://www.mdpi.com/2076-3417/12/11/5507.

[20] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, Sep. 2017. DOI: https://doi.org/10.1109/mcc.2017.4250931.

[21] [Online]. Available: https://cloud.google.com/learn/what-is-microservices-architecture#section-2.

[22] P. Irudayaraj and S. P., “Adoption advantages of micro-service architecture in software industries,” *International Journal of Scientific Technology Research*, vol. 8, pp. 183–186, Sep. 2019.

[23] S. Chatterjee and A. Keprate, *Exploratory data analysis of the n-cmapss dataset for prognostics*, Dec. 2021. DOI: https://doi.org/10.1109/IEEM50564.2021.9673064. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9673064.

[24] M. Arias Chao, C. Kulkarni, K. Goebel, and O. Fink, “Aircraft engine run-to-failure dataset under real flight conditions for prognostics and diagnostics,” *Data*, vol. 6, no. 1, p. 5, Jan. 2021. DOI: https://doi.org/10.3390/data6010005.

[25] K. Goebel, J. Celaya, S. Sankararaman, I. Roychoudhury, M. Daigle, and A. Saxena, *Prognostics: The Science of Making Predictions.* Apr. 2017, ISBN: ISBN-10: 1539074838 ISBN-13: 978-1539074830.

[26] [Online]. Available: https://portal.fabric-testbed.net/about/about-fabric.

[27] K. Bonawitz *et al.*, *Towards federated learning at scale: System design*, Mar. 2019. DOI: https://doi.org/10.48550/arXiv.1902.01046. [Online]. Available: http://arxiv.org/abs/1902.01046.

[28] L. Rushani and F. Halili, “Differences between service-oriented architecture and microservices architecture,” *International Journal of Natural Sciences: Current and Future Research Trends*, vol. 13, pp. 30–48, Apr. 2022. [Online]. Available: https://ijnscfrtjournal.isrra.org/index.php/Natural_Sciences_Journal/article/view/1089.

[29] T. Salah, M. Jamal Zemerly, C. Y. Yeun, M. Al-Qutayri, and Y. Al-Hammadi, “The evolution of distributed systems towards microservices architecture,” in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016, pp. 318–325. DOI: 10.1109/ICITST.2016.7856721.

[30] K. Gos and W. Zabierowski, “The comparison of microservice and monolithic architecture,” Apr. 2020, pp. 150–153. DOI: 10.1109/MEMSTECH49584.2020.9109514.

[31] N. Salaheddin and N. Ahmed, “Microservices vs. monolithic architectures [the differential structure between two architectures],” *MINAR International Journal of Applied Sciences and Technology*, vol. 4, pp. 484–490, Oct. 2022. DOI: 10.47832/2717-8234.12.47.

[32] 2018. [Online]. Available: https://flask-restful.readthedocs.io/en/latest/.

[33] Y. Shi, Y. Zhang, Y. Xiao, and L. Niu, “Optimization strategies for client drift in federated learning: A review,” *Procedia Computer Science*, vol. 214, pp. 1168–1173, 2022, 9th International Conference on Information Technology and Quantitative Management, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2022.11.292. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705092202004X.

[34] A. S. Foundation, *Apache jmeter-apache jmetertm*, 2019. [Online]. Available: https://jmeter.apache.org/.

[35] Radon Development Team, *Radon documentation.* [Online]. Available: https://radon.readthedocs.io/en/latest/intro.html.

[36] R. Banker, S. Datar, and D. Zweig, "Software complexity and maintainability," Jan. 1989, pp. 247–255. DOI: 10.1145/75034.75056.

[37] A. W. R. Emanuel, R. Wardoyo, J. E. Istiyanto, and K. Mustofa, *Modularity index metrics for java-based open source software projects*, 2013. arXiv: 1309.5689 [cs.SE]. [Online]. Available: https://arxiv.org/abs/1309.5689.

[38] M. H. Brendan, E. Moore, D. Ramage, S. Hampson, and Blaise, "Communication-efficient learning of deep networks from decentralized data," *arXiv (Cornell University)*, Feb. 2016. DOI: https://doi.org/10.48550/arxiv.1602.05629.

[39] Y. Zhao, L. Liu, L. Lai, N. Suda, D. J. Civin, and V. Chandra, "Federated learning with non-iid data," *arXiv (Cornell University)*, Jun. 2018. DOI: https://doi.org/10.48550/arxiv.1806.00582.

[40] S. Rajendran *et al.*, "Cloud-based federated learning implementation across medical centers," *JCO Clinical Cancer Informatics*, no. 5, pp. 1–11, Dec. 2021. DOI: https://doi.org/10.1200/cci.20.00060.

[41] *Federated learning on google cloud — cloud architecture center*. [Online]. Available: https://cloud.google.com/architecture/federated-learning-google-cloud.

[42] S. Stefanov, "Automating the centralized-to-federated transition for the nasa c-mapss dataset," Ph.D. dissertation, Jun. 2023.

[43] M. Staňo, L. Hluchý, M. Bobak, P. Krammer, and V. Tran, "Federated learning methods for analytics of big and sensitive distributed data and survey," May 2023, pp. 000 705–000 710. DOI: 10.1109/SACI58269.2023.10158622.

[44] Q. Li *et al.*, "A survey on federated learning systems: Vision, hype and reality for data privacy and protection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[45] M. J. Sheller, G. A. Reina, B. Edwards, J. Martin, and S. Bakas, "Federated learning in medicine: Facilitating multi-institutional collaborations without sharing patient data," *Scientific reports*, vol. 10, no. 1, pp. 1–12, 2020.

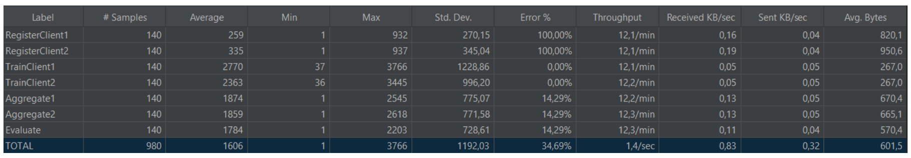
[46] A. Hard *et al.*, "Federated learning for mobile keyboard prediction," in *Proceedings of the 1st Workshop on Privacy-Aware Mobile Computing*, 2018, pp. 1–7.

[47] W. Lai and Q. Yan, "Federated learning for detecting covid-19 in chest ct images: A lightweight federated learning approach," in *2022 4th International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, 2022.

[48] Y. Ao and Y. Jiang, "Manufacturing data privacy protection system for secure predictive maintenance," in *2022 5th International Conference on Data Science and Information Technology (DSIT)*, IEEE, 2022, pp. 1–5.

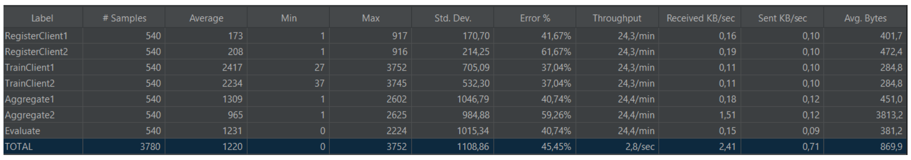
# Appendix A

# Detailed JMeter Tests output



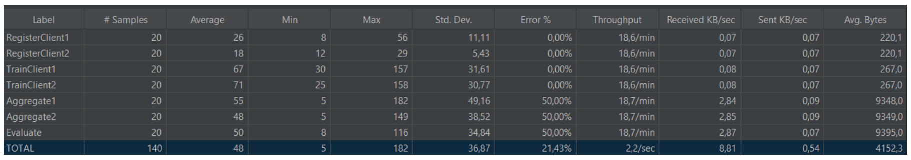
| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| RegisterClient1 | 140 | 259 | 1 | 932 | 270,15 | 100,00% | 12,1/min | 0,16 | 0,04 | 820,1 |
| RegisterClient2 | 140 | 335 | 1 | 937 | 345,04 | 100,00% | 12,1/min | 0,19 | 0,04 | 950,6 |
| TrainClient1 | 140 | 2770 | 37 | 3766 | 1228,86 | 0,00% | 12,1/min | 0,05 | 0,05 | 267,0 |
| TrainClient2 | 140 | 2363 | 36 | 3445 | 996,20 | 0,00% | 12,2/min | 0,05 | 0,05 | 267,0 |
| Aggregate1 | 140 | 1874 | 1 | 2545 | 775,07 | 14,29% | 12,2/min | 0,13 | 0,05 | 670,4 |
| Aggregate2 | 140 | 1859 | 1 | 2618 | 771,58 | 14,29% | 12,3/min | 0,13 | 0,05 | 665,1 |
| Evaluate | 140 | 1784 | 1 | 2203 | 728,61 | 14,29% | 12,3/min | 0,11 | 0,04 | 570,4 |
| TOTAL | 980 | 1606 | 1 | 3766 | 1192,03 | 34,69% | 1,4/sec | 0,83 | 0,32 | 601,5 |

**Figure A.1: Disagreement scenario results microservices-based architecture results**



| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| RegisterClient1 | 540 | 173 | 1 | 917 | 170,70 | 41,67% | 24,3/min | 0,16 | 0,10 | 401,7 |
| RegisterClient2 | 540 | 208 | 1 | 916 | 214,25 | 61,67% | 24,3/min | 0,19 | 0,10 | 472,4 |
| TrainClient1 | 540 | 2417 | 27 | 3752 | 705,09 | 37,04% | 24,3/min | 0,11 | 0,10 | 284,8 |
| TrainClient2 | 540 | 2234 | 37 | 3745 | 532,30 | 37,04% | 24,3/min | 0,11 | 0,10 | 284,8 |
| Aggregate1 | 540 | 1309 | 1 | 2602 | 1046,79 | 40,74% | 24,4/min | 0,18 | 0,12 | 451,0 |
| Aggregate2 | 540 | 965 | 1 | 2625 | 984,88 | 59,26% | 24,4/min | 1,51 | 0,12 | 3813,2 |
| Evaluate | 540 | 1231 | 0 | 2224 | 1015,34 | 40,74% | 24,4/min | 0,15 | 0,09 | 381,2 |
| TOTAL | 3780 | 1220 | 0 | 3752 | 1108,86 | 45,45% | 2,8/sec | 2,41 | 0,71 | 869,9 |

**Figure A.2: Non-disagreement scenario microservices-based architecture results**



| Label | # Samples | Average | Min | Max | Std. Dev. | Error % | Throughput | Received KB/sec | Sent KB/sec | Avg. Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| RegisterClient1 | 20 | 26 | 8 | 56 | 11,11 | 0,00% | 18,6/min | 0,07 | 0,07 | 220,1 |
| RegisterClient2 | 20 | 18 | 12 | 29 | 5,43 | 0,00% | 18,6/min | 0,07 | 0,07 | 220,1 |
| TrainClient1 | 20 | 67 | 30 | 157 | 31,61 | 0,00% | 18,6/min | 0,08 | 0,07 | 267,0 |
| TrainClient2 | 20 | 71 | 25 | 158 | 30,77 | 0,00% | 18,6/min | 0,08 | 0,07 | 267,0 |
| Aggregate1 | 20 | 55 | 5 | 182 | 49,16 | 50,00% | 18,7/min | 2,84 | 0,09 | 9348,0 |
| Aggregate2 | 20 | 48 | 5 | 149 | 38,52 | 50,00% | 18,7/min | 2,85 | 0,09 | 9349,0 |
| Evaluate | 20 | 50 | 8 | 116 | 34,84 | 50,00% | 18,7/min | 2,87 | 0,07 | 9395,0 |
| TOTAL | 140 | 48 | 5 | 182 | 36,87 | 21,43% | 2,2/sec | 8,81 | 0,54 | 4152,3 |

**Figure A.3: Client-server based architecture results**