Within the domain of Data Analytics (DA), there is an important field known as Machine Learning (ML), which is embedded in everyday people's lives. A significant topic within this field is Federated Learning (FL). FL occurs when different devices collaborate to build a common model without exchanging their data; instead, the data remains on the devices, and only model updates are sent to a central server where aggregation occurs.

A good example for FL in everyday people’s lives is when phone users are typing a message, and the keyboard predicts words to complete the sentence, a model developed by Google engineers.

A different example where FL is helping society move forward is in the healthcare sector. Patients are using wearable devices to track their movements and help doctors monitor their health conditions.

FL is growing rapidly and is helping the technological and medical sectors build robust machine learning models. This significant development motivates the focus of this thesis.

The primary purpose of this project is to experiment with FL frameworks to evaluate their implementability and develop a functional FL server. Therefore, the research objectives (ROs) are:

* **To evaluate the implementability of existing FL frameworks.** This section experiments with popular FL frameworks, such as *PySyft, FATE, Flower, FedML* and *TensorFlow Federated* (TFF)*.* by examining their architecture and their applicability to real-world FL scenarios. This evaluation serves as the starting point for the primary research.
* **To develop a FL server**. A practical example of a web Flask FL server will be built, featuring two scenarios: technological and pharmaceutical. Each scenario will run separately, connecting five clients. Each scenario will have its own data, synthetic data for the technological scenario and images for the pharmaceutical scenario. These configurations will be trained, validated, and tested using Neural Networks (NN) that will classify binary outputs. This experiment aims to bridge the gap between popular FL frameworks and real-world FL applications.
* **To compare FL frameworks and the FL server.** This includes discussing the pros and cons each method brings to key industries such as the technological and pharmaceutical sectors. The comparison will address aspects like ease of implementation, scalability, data privacy, and model performance.

This concept was introduced in 2016 by Google engineers (McMahan et al., 2016). FL is a shared model that is trained across multiple devices, often referred to as clients. Each client trains its own local model and sends the updates to a central server, where the updates are aggregated to improve the global model. It is important to note that clients do not exchange their data; the data remains private for each client. This principle drove the design of FL, following the concepts of focused collection or data minimization, which were introduced by the White House in 2013. The intent is to prevent personal data from being sent over the network and potentially being stolen or manipulated by malicious third parties.

Depending on the nature of client FL can be classified in two types cross-device and cross-silo

The clients for cross-device can be mobile devices, edge devices, Internet of Things (IoT) devices, smartphones, tablets, wearables, etc. Figure 2.2.1. illustrates this scenario.  
The characteristics are, the high number of participants it can be thousands to millions of devices, it may have limited processing power and battery life, datasets tend to be small and network bandwidth may be limited. Devices may also connect and disconnect intermittently.

In this scenario, clients can be organizations or institutions such as hospitals, banks, and companies using large data centers. Figure 2.2.2 illustrates a cross-silo ecosystem. Some differences compared to the cross-device scenario include, clients are no longer small devices, there are fewer clients, clients have high computational power and large datasets, and the network is reliable with stable communication.

FL can be categorized according to the distribution of the data held by the clients participating in the modelling. These categories help to understand the different methodologies and use cases for FL.

Horizontal Federated Learning (HFL) or sample-based FL, occurs when different clients have datasets that share the same feature space but differ in the samples they hold (see Figure 2.3.1). A practical example of HFL is when two hospitals in different regions each have patient records with the same features (e.g., age, height, weight, diagnosis) but for different patients. These hospitals can collaborate to train a model to predict disease outcomes without sharing patient data.

Vertical Federated Learning (VFL), or feature-based FL, occurs when different clients have datasets that share the same sample IDs but differ in the feature space. An example of this is a bank and an e-commerce company that have data on the same set of customers. The bank has financial information such as credit scores and loan histories, while the e-commerce company has purchase behaviours and browsing records. By combining their data, they can build a model to predict customer credit without sharing raw data.

Federated Transfer Learning (FTL) is applicable when the datasets of different clients differ in both samples and features, and the overlap between both is minimal. To clarify, a good example is a European pharmaceutical company and a healthcare research institution in China collaborating using FTL. The pharmaceutical company has drug efficacy data, while the healthcare institution has patient health records. Even though they have different types of data, it is possible to train a global model to predict drug effectiveness on certain health conditions.

Terms like FL and Distributed Machine Learning (DML) can create confusion due to their similarities. The main difference lies in the training process: in FL, there is a central server that aggregates updates sent by the clients, whereas in DML, there is no central server; instead, data is spread across different nodes and computations are shared among these nodes. Table 2.4 will help clarify the intricacies of each concept.

Given the rapid advancements in DA and subsequently in ML, FL is also growing fast. A thorough review of the current literature is essential. By typing *“Federated Learning”* in Google Scholar and selecting *“review articles”,* it returns 7,510 articles in 0.07 seconds. The aim of this chapter is to select valid and relevant articles that align the ROs with the literature review, ensuring a smooth experimentation process that validates these objectives. All sources are organized into five themes: FL frameworks, FL algorithms, real-world FL settings, FL datasets, and FL server implementation.

A requirement for selecting the FL frameworks was that they must be open source. Open-source frameworks are transparent and trustworthy, developed and maintained by a collaborative community, free to use, and constantly evolving. Additionally, they can be customized to meet users' specific needs. After the selection it was necessary to rank the frameworks. To accomplish this GitHub stats were helpful. Figure 3.1. depicts *PySyft* GitHub repository stats.

Contributors, forks and stars were counted for each framework. These stats were then normalised and finally averaged. Figure 3.2. illustrates the formulas and table 3.1. shows the results being *PySyft* the most popular open-source FL framework.

By creating this ranking, the population sample for objective one was defined as *PySyft, FATE, Flower, FedML,* and *TFF*. The population was restricted to the top five FL frameworks due to the limited amount of time. As the sampling method is non-probabilistic and the sampling type is judgmental, this approach to ranking the FL frameworks may help mitigate the inherent bias that experimentation has as a primary research methodology and also focus the selection on samples that can represent the entire population. Based on this selection, section 3.1.1. details the frameworks that will be used in the experimentation, and section 3.1.2. lists are the remaining frameworks. contributed to the idea of evaluating open-source FL frameworks and helped establish criteria for comparison. These criteria include ease of use and deployment, development, analysis capabilities, accuracy, and performance. These criteria will be applied in Section 5, where popular FL frameworks are evaluated.

introduced *PySyft* a multi-language library that facilitates secure and private ML. It was developed by the *OpenMined* community with the objective of making FL data science more accessible through Python bindings and user-friendly interfaces. *PySyft* uses libraries like *PyTorch* and TensorFlow with additional capabilities. Comparing it with other frameworks like *TFF* and *PaddleFL*; *PySyft* offers detailed building blocks, allowing developers to implement FL efficiently. Also compared to Flower that supports heterogeneous client environments and offers tools for mobile and edge devices, claiming and advantage over *PySyft* in these aspects.

According to FATE is provided to aid enterprises and institutions in implementing large-scale and distributed collaborative learning with data protection. A number of secure computation protocols and machine learning algorithms are supported within FATE. Through the out-of-box usability and end-to-end building modules and visualization tools, users are able to get their applications up and running with efficiency and effectiveness. It not only offers a distributed platform that supports both stand-alone and cluster deployment but also privacy-preserving *XGBoost*, federated transfer learning, and multi-variate data. *FATE* interacts with users using *FATE-FLow*, which serves as the scheduling system, *FATE-Board*, a visualization tool, and *FATE-Serving*, which is an inference high-performance serving engine. *KubeFATE* is designed by *VMware* to have *FATE* constructed over *Kubernetes* at the data centre, hence an enterprise-managed solution over organizations' distributed infrastructure. It also supports cross-cloud deployment and management through *FATE-cloud*. Second, *FATE* has a security definition in which all parties are honest-but-curious, ensuring that the server learns only aggregated parameters, but not the data of any individual. It guarantees performance that is lossless, which means the algorithms in *FATE* provide comparable accuracy to a centralized solution. *FATE* supports research into the industry communities working together and has been seen as an increasingly business application of interest. Future work in the field will focus on the integration of blockchain functionalities into *FATE*; building lightweight versions of *FATE* for edge deployment and applications; and building new applications using *FATE* in an industrial scenario, such as computer vision and automatic speech recognition.

*FedML* is an open research library and benchmark built for enabling development support and fair comparison in federated learning algorithms. Compared with previous works, it addresses the current limitation of supporting different configurations and computing paradigms for distributed training, mobile on-device training, and standalone simulation. It makes flexible, generic API designs, standardized algorithm implementations, and a comprehensive benchmark dataset available for non-I.I.D. settings. *FedML* is architected into high-level API interactions through its *FedML-API*, whereas the low-level functionality is realized by *FedML*-core to allow convenient implementation of distributed algorithms by users. This library also includes a real-world module for training on smartphones, called *FedML-Mobile*. Using such cryptographic primitives, standardized benchmarks can enforce privacy, security, and robustness, ensuring fair comparisons. *FedML* is designed to encourage community contributions that push the boundaries of what it can do. In design, the critical requirements are met for federated learning research by which researchers can prototype new algorithms and evaluate them on a common fair platform with consistent datasets and experimental settings. The broad support of computing paradigms by the library will make it applicable in different research scenarios, from huge-scale distributed systems to resource-constrained mobile devices. This flexible design of the API allows researchers to extend and customize the library for their specific needs. Standard benchmarks enable trustworthy comparisons of the performance of different algorithms. Moreover, *FedML* is not only robust in terms of privacy and security in FL but also applies advanced cryptographic techniques that ensure user data is secure to the level of model robustness. FedML follows a community-driven approach and is always changing and extending its features. New improvements are regularly updated based on feedback and contributions from global researchers. Such a collaborative effort helps push the frontiers of FL and ensures that *FedML* retains its leading status in research and development.

presented a user-friendly framework, an open-source framework developed to make the implementation and scalability of FL much easier. *Flower's* goal is to bridge the gap between academic research and practical application in real-world FL settings with large-scale experiments and very varied device settings. The big advantage of *Flower*, compared to most other frameworks for simulations, is that it can be used in real deployments with real devices; thus, it is a very good and flexible tool. It has been designed with an architecture supporting most machine learning frameworks, including *TensorFlow*, *PyTorch*, while offering flexible API designs, standardized algorithm implementations, and benchmark datasets for non-Independent and Identically Distributed (non-IID) settings. This way, it has proven to be an excellent tool for experimenting with FL in different configurations and computational paradigms. The important abstractions and functionalities inside Flower are the high-level API interactions in the part represented by *FedML-API*, and low-level functionality in the part represented by *FedML-Core*. This makes it easier for the users to program distributed algorithms. It also has an on-device training capability for smartphones with cryptographic techniques to guarantee privacy, security, and robustness called *FedML-Mobile*. This is a framework motivating the community's contribution continuously to increase the power of it. The architecture of Flower allows a transparent, seamless transition for researchers from simulation to deployment on real devices. With heterogeneous client support and scalable infrastructure, Flower becomes a tool absolutely necessary in the hands of the researcher when FL investigation is performed so that the gap between theory and practice may be addressed.

delve into how *TFF*, an open-source framework, is utilized for machine learning on decentralized data. It has been designed for research and experimentation. Some of the key features are TFF enables FL through low-latency models with less power consumption. The framework uses two layers, the FL learning Application Programming Interface (API) and the federated core (FC) API. The FL API allows developers to implement training and evaluation on existing TensorFlow models through a high-level interface. The FC API integrates TensorFlow with distributed communication operators focusing on computations across distributed systems like mobile phones, tablets, and sensors. Comparing *TFF* to other frameworks, it offers a unique well integrated structure others do not provide this level of integration. *TFF* allows experimenting with new algorithms is not tied to predefined algorithms.

Another innovative framework OpenFL created by Intel Labs and the University of Pennsylvania, OpenFL supports decentralized machine learning models. It allows organizations to train models using data locally without any transfer, and that operates by distributing a global model across various nodes while each organization trains its model locally. Model updates are sent to an aggregator to enhance the global model. This framework is compatible with popular ML frameworks like TensorFlow and PyTorch. In comparison to other frameworks, it stands out due to its open-source nature, TensorFlow Federated or PySyft focus more on academic research applications while OpenFL is focused on real-world applications.

describe NVIDIA FLARE (NF) as an open-source Software Development Kit (SDK) purposefully developed to make it easier for data scientists and researchers to train federated learning models. NF, in support of many collaborators, is applied to create powerful and generalizable AI models by sharing the weights of the models rather than the private data. It is very lightweight and flexible, supporting the scaling of different machine learning frameworks, among which are *PyTorch*, *TensorFlow*, and *XGBoost*. In this way, NF allows researchers to adapt their ML workflow under a federated paradigm and finally achieve secure and privacy-preserving multiparty collaboration through techniques like homomorphic encryption and differential privacy. Some of the key aspects found in NF are high-level APIs of programmable FL workflows, prototyping simulators, and a project management dashboard. It is constructed to support productivity features in the built-in SDK research to deployment simulation to the real-world architecture of NF: multitasking, high availability, server failover, and secure provisioning. In addition, a good application for NF has been found in practice, particularly within the health sector, with regard to predicting clinical outcome for COVID-19 patients and segmenting brain lesions in medical imaging. This paper also presented some of the numerous benefits that a component-based design of NF accrues to make it extensible and customizable, thereby inviting the research community to further develop it.

propose *PaddleFL* is an open-source federated learning framework developed by *PaddlePaddle*, with the purpose of safe collaboration on training machine learning models over a massive amount of devices or organizations without sharing raw data. It provides implementations for different federated learning algorithms and flexible, extensible architecture that will easily plug into different machine learning frameworks. In the aspect of implementations, PaddleFL is an attempt for distributed model training, while it solves the privacy issues in data. In this way, this project will have applications in multiple fields: healthcare and finance, where data security becomes very important. The framework itself has built-in tools for data preprocessing and model training and evaluation under horizontal and vertical federated learning scenarios. It aims to be user-friendly by having complete documentation and examples that can assist the user in quickly getting a foothold. PaddleFL adopts advanced techniques, including homomorphic encryption and secure multiparty computation, into data safety and privacy during the training process. It is also updated and improved all the time by the open-source community, making it a strong candidate for the implementation of federated learning projects.

introduced *Substra* as a framework designed to make machine learning both collaborative and secure. They developed *Substra* to handle the tricky issue of working with sensitive data without compromising privacy. Instead of moving data around, *Substra* keeps it decentralized data stays where it is, and only the necessary algorithms and non-sensitive information are shared. *Substra* uses Distributed Ledger Technology (DLT) to ensure that all operations are secure and traceable. This means there is no need to rely on a central authority to verify the integrity of the data and operations. Originally designed for healthcare applications, *Substra* is flexible enough to work with various data types, algorithms, and programming languages. It supports multiple computation methods, especially those used in Federated Learning. The framework is built on three core principles: collaboration, privacy, and traceability. It brings together data providers and algorithm designers to work on shared goals while keeping data private and secure. Substra manages four key assets: objectives, datasets, algorithms, and models. Each of these assets has specific permissions to control who can access and process them. Computations in *Substra* are coordinated across different nodes, ensuring that data never leaves its original location. The decentralized architecture uses smart contracts to enforce permissions and maintain a tamper-proof ledger of all activities. This makes *Substra* a versatile tool for various collaborative machine learning projects, such as data and algorithm collaborations, data consortiums, and combined training and evaluation efforts.

developed *FLGo* a platform designed to streamline the process of cross-application FL research and enhance shareability among developers. It is a lightweight FL framework aiming to be a customizable solution to suit different applications and data heterogeneity. *FLGo* addresses the gap that exists in current FL frameworks which often make the FL deployment very complex. Some of the key Features, are benchmarks and algorithms, customization, experimental tools, and high degree of shareability. Compared to other frameworks it stands out in, system heterogeneity, high-level API, multi-architecture support, asynchronous operations and customization and flexibility. As a conclusion *FLGo* has been developed with the intention of making FL more accessible to a broader range of developers by simplifying customization and enhancing its shareability. It also aims to bridge the existing gap with conventional machine learning and FL.

After reviewing the frameworks, it was clear that they all utilize algorithms. Some of them implement custom FL algorithms; however, there is one algorithm common to all of them: *FedAvg*. *SecureBoost* and *FedProx* are also used by multiple frameworks. These algorithms, along with *FedMA*, will be explained in the sections below.

The key algorithm that was developed for FL was Federated Averaging *(FedAvg)* that allows to train the models distributed among multiple devices while preserving the centralized control of the process. This approach was proposed in the paper by Google researchers. *FedAvg* improves the basic of federated learning by adopting the Stochastic Gradient Descent (SGD) algorithm, generally in terms of the iteration in the model averaging.

It starts with the central server sending global model parameters to other servers in the nearest proximity. In each round clients are randomly chosen to participate with others in the network. These selected clients get the current global model and then update it for several epochs on their local data using mini-batch SGD. Each of the clients also derives new model parameters These updated parameters are sent back to the server, which aggregates them by averaging, weighted by the number of training samples on each client.

Besides, *FedAvg* works well with non-IID data, which occur frequently in the context of FL where data is dispersed across clients in a likely non-uniform manner. Specifically, *FedAvg* reduces the number of communication rounds by averaging the locally computed model updates and it is efficient even in environments with low bandwidth. The algorithm also scales well to a large number of clients as in each round only a fraction of the clients is involved, and the server only aggregates their updates.

When tested across different model structures and datasets for image classification such as *MNIST*, *CIFAR10* and language modelling such as Shakespeare dataset, it was shown that *FedAvg* succeeds in cutting down the communication costs while still giving a sound performance in conditions of non-IID and imbalance of the data. In sum, *FedAvg* is foundational in FL since it optimizes both computational and communication efficiency.

In short, *FedProx* is a federated optimization algorithm designed with the objectives of handling challenges that come with system and statistical heterogeneity in a federated learning network. It is first motivated as an extension and re-parameterization of the *FedAvg* algorithm. While *FedAvg* has demonstrated empirical success, it's been seen to falter in the face of system capability diversity and the nonidentical distribution of data on the devices. These are the aspects that *FedProx* modifies to handle them and increase the robustness and stability of the optimization process. In this context, the addition of a proximal term to the local objective function within *FedProx* is aimed at compensating for a potentially large influence of local updates so that they remain close to the global model. The proximal term serves two purposes: it naturally introduces statistical heterogeneity in the local updates by properly controlling the distance from the original global model and allows us to safely introduce variable amounts of local work, coming from systems' heterogeneity.

While *FedAvg* requires all the devices to perform an equal amount of work, a fixed number of local epochs, *FedProx* allows for non-uniform amounts of work by devices, up to their system capability, in terms of resources. This can handle stragglers (i.e., devices that do less work) better without dropping them, leading to more stable convergence. The server initializes the global model and samples a subset of devices in each iteration. For each sample device, it executes one local update based on its data for optimization of a modified local objective function with an added proximal term. The proximal term is defined as , where is the current global model and is the local model. After local updates, each device sends the updated model back to the server. Aggregating these updates, one averages the updates and thereby forms a new global model, exactly the same as *FedAvg*.

Interestingly, *FedProx* shows more stability with respect to accuracy in heterogeneous settings than the baseline *FedAvg* does, and even more, it is probably true for such a setting. The algorithm is valuable for federated learning applications, as the proximal term enables it to handle variable amounts of local computation, hence mitigating issues that arise in systems and statistical heterogeneity.

The challenge in solving through the development of *FedMA*, or Federated Matched Averaging, lies in federated learning, especially when one opts for modern neural network architectures such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory Networks (LSTMs). Traditional methods like FedAvg tend to perform poorly due to their weight averaging at the coordinate-wise level, which results in suboptimal global models—more so in cases with very heterogeneous data.

*FedMA* constructs the global model in a shared layer-wise manner by matching and averaging hidden elements, for instance channels in convolutional layers, or hidden states in LSTM layers, in a feature-extraction-signature-wise manner. This matching of feature-extracting signatures ensures that similar functional components are averaged together, thus outperforming conventional strategies while reducing the communication burden.

The algorithm involves several key steps. In the matching process, the server collects first-layer weights from all the clients and conducts a matching process in which it identifies and aligns neurons or channels similar across different models. This ensures that the models maintain the mean of similar functional components. This stage is executed by permutation and averaging, whereby a server takes the match of weights and takes an average as the first layer of the global model, which it then rebroadcasts to clients. Clients would then perform local training on the updated first layer of the global model, holding the matched global layer fixed. These steps go on iteratively, layer by layer, until the full model has been updated.

During this iterative matching and model adaptation process, the Hungarian algorithm is exploited by the server to handle the matching problem efficiently in order to ensure the best permutation of weights. FedMA also caters for the heterogeneity in the data by considering the size of the global model in line with the size of local models and data distribution while ensuring that globally the model is highly efficient and competent even with the changes in data on the clients' side.

More generally, *FedMA* strengthens the federated learning paradigm by aiming for an approach of functional matching of model components to improve overall performance and reduce communication costs. It relies on the utilization of advanced matching techniques by making use of permutation invariance property to ensure the global model integrates the knowledge from all participating clients.

*SecureBoost* is a boosting algorithm by trees developed under the federated learning scheme. It allows collective model training across the parties without disclosing individual data. It is designed for better privacy-destroying low-quality model consolidation present in almost all such processes and to comply with data protection regulation concerns like GDPR.

In *SecureBoost*, data is vertically partitioned: different parties own different features on the same set of users. The first step is privacy-preserving entity alignment, in which data samples from involved parties are matched using privacy-preserving protocols to ensure that nonshared data remains private. Finally, the jointly trained gradient boosting model is used in collaborative model training. Each of the parties computes locally optimal splits for the decision trees using their own data and encrypted gradient statistics sent by the active party. It encrypts gradient and Hessian values and ensures the privacy of data.

In the protocol, SecureBoost defines roles for active parties, who have class labels, and passive parties, who have only feature data. The active party coordinates the training process, including the aggregation of model updates. Model construction follows the philosophy of the widely used and successful *XGBoost*: sequential tree construction by adding splits that optimize a loss function. In such a federated setting, SecureBoost ensures encryption and secure sharing of gradient and Hessian values which are used for splits among the parties. SecureBoost further makes use of additive homomorphic encryption to ensure that each party can calculate the required gradient and Hessian sums for all possible splits locally and send these encrypted values to the active party, who eventually decrypts them to find the globally best split.

*SecureBoost* is a design of loss lessness, which can achieve the same accuracy as tree-boosting algorithms without privacy preservation under centralized data; it might even be appropriate for industrial applications that need strong privacy guarantees. This approach is indeed scalable and highly efficient with very large datasets, keeping the performance on par with non-federated methods, such as *XGBoost* and Gradient Boosting Decision Trees (GBDT), in terms of both convergence and accuracy, even under the influence of privacy constraints. SecureBoost introduces a practical and secure framework of federated learning, in which different organizations can jointly build machine learning models without sharing data. Leveraging advanced cryptographic techniques, SecureBoost protects private information from being exposed throughout the whole process of learning, making it very valuable in privacy-preserving machine-learning applications.

In summary, these federated learning algorithms try to solve a set of challenges, most of which exist for any distributed training system. *FedAvg* started it off by allowing efficient training across devices with centralized control. *FedProx* generalizes that approach to heterogeneity in system capabilities and data distribution. *FedMA* further improves model performance by leveraging advanced matching for layer-wise averaging and is specifically applied to complex architectures using deep neural networks. *SecureBoost* introduces strong privacy-preserving measures in order to collaboratively train securely among the parties with vertically partitioned data. These algorithms improve federated learning in computational efficiency, stability, communication cost, and privacy.

There are numerous papers and sources about FL, but after conducting thorough research, nothing has been found regarding real-world FL settings, specifically real FL systems where a company makes their global models publicly available. This gap is understandable because one of the reasons for FL existence is privacy, therefore, companies prioritise keeping their FL systems private. What companies do make available are FL system architectures, algorithms, prototypes, and experiments. They explain how to implement FL but do not show a FL system that is in production. Consequently, they will never share with the public how Google engineers use FL to predict the next word on people's phones (Hard *et al.*, 2019). In other words, a company will not disclose an FL method that generates revenue for its business. Recognizing this limitation, the following three papers are not company production FL models but are scenarios close to a real-world setting.

bridged the gap between traditional experiments on federated learning and real-world applications, testing with *FS-Real*, a system built to handle challenges related to heterogeneous device environments. Traditional FL research mainly tests with a homogeneous device environment, very different from the diversity and variability of real-world devices, yielding subsequent application problems. A large number of experiments showed that FS-Real is usable, efficient, and scalable: focusing on the effects of FL performance by heterogeneous devices and different scales. The different distributions of devices are homogenous. It evaluated the performance of FL algorithms, such as *FedAvg*, under these different distributions and scales using model accuracy, fairness, convergence time, communication efficiency, and client utilization as metrics. *FS-Real* has been experimented in the case of high scalability and robustness by undergoing stress tests to handle up to 100,000 clients. It shows capability in *FS-Real* to handle large FL tasks effectively. On the other hand, optimized concurrency techniques along with robust mechanisms of client selection make FL processes very efficient in heterogeneous devices with varied responsiveness. More advanced FL techniques were also tested in *FS-Real*, personalization, communication compression, and asynchronous aggregation. Personalization would involve algorithms like *FedBABU* to improve client performance. Communication compression was in the efforts of reducing message size to save on bandwidth and communication costs. Asynchronous aggregation allowed faster devices to move ahead without waiting for slower devices, which facilitated improvement in efficiency and robustness during training. Key results are a significant performance gap between both homogeneous and heterogeneous settings, usually with lower and more varied accuracies for heterogeneous devices due to the varied training dynamics. Real-world application efficiency was better for *FS-Real* than all other tools, similar to how its simulation fidelity was better; the system displayed enhanced capabilities to cope with heterogeneous devices. The system has been proved highly scalable, as heavy loads of clients have been effectively managed by maintaining effective performance through enhanced concurrency and robust mechanisms of client selection. This makes *FS-Real* a robust and scalable approach to solving the real-world condition problem, thereby closing the gap between traditional FL research and practical applications. This work will help develop more effectively and efficiently the deploying FL on a large scale and under heterogeneous conditions.

The platform *AI4EOSC* developed a model using the Pneumonia Chest X-Ray dataset and implemented using a CNN The task involves using the images to implement a NN to predict whether new X-rays are normal or indicate pneumonia. The initial dataset is divided among three clients that simulate hospitals wishing to collaboratively develop a global model without sharing patient data. For each client, the training data is divided using a random split of 75% for training and 25% for testing. Using the *AI4* FL server, the model is deployed. The Flower package is used to build the model. The use of *Jupyter Notebook* (JN) is recommended; once the server is running, hardware is configured by selecting the number of CPUs, disk memory, and RAM. The next step is federated configuration, which includes the number of rounds, evaluation metric, number of clients, and federated aggregation strategy. On the client side, three instances are opened and running in the cloud. It is important to pass the *AI4* FL server key into each client's Python script. After this, by running *$ python3 Client.py*, each client will be initialized and wait for the third one to start and trigger the entire process. On the server side, accuracy is calculated. This is a great example, and the approach differs from how frameworks typically deploy federated learning; in summary, this example is close to a real-world implementation case.

This practical implementation uses Azure Machine Learning (AML), with the same dataset as discussed above. The experiment begins by training the model using a classical approach and then comparing this result with the federated one. Three clients are present, representing hospitals in the US, Europe, and Asia; these are computer instances in Azure. The Nvidia Flare framework is used in this model. A JN is utilized as a controller, sending instructions to the clients and tracking accuracy. In conclusion, this demonstration, along with the one above, is fundamental to understanding how FL is implemented. Unlike section 3.1, where FL frameworks emulate the clients within a JN by encapsulating them into variables, AML and the AI4 FL server use virtual machines that closely emulate actual hospitals. They have different IP addresses, and connection protocols must be used to connect and train the model.

introduced a real-world image dataset specifically designed to address the challenges associated with non-IID and unbalanced data distributions that are common in federated settings. This dataset comprehends images captured by 26 street cameras, across 900 images categorized into 7 object types. Method focus is the study of implementing and benchmarking two major object detection algorithms, YOLO and Faster R-CNN. As a result, a non-IDD and imbalanced dataset was created, ideal to test FL models under real-world conditions. This dataset offers a unique resource compared to other benchmark datasets used in FL research. MNIST or CIFAR are balanced and homogeneous whereas the real-world image dataset provides a real challenge due to its realism, leading to better FL models after training. Also, the use of YOLO and Faster R-CNN differs from simpler models or more synthetic setups often used. In summary realistic approaches can enhance FL modelling. Similarly, used versions of MINST and CIFAR, Fashion-MNIST, CIFAR-10 and CIFAR-100. These datasets are chosen for their varying levels of complexity and the ability to simulate non-IDD conditions effectively. Other frameworks employ the *FedAvg* algorithm, not performing well with non-IID data, and other counterparts do not pay attention to aggregating dissimilar client updates that can lead to poor global modelling. discuss the use of various datasets in the context of FL for medical applications, emphasizing the importance of handling non-IID, unbalanced, and vertically split data. They highlight the challenges of training models across different distributions, which is common in healthcare due to varying patient demographics across hospitals. Key datasets mentioned include MNIST for handwritten digits, CIFAR-10 for object recognition, and keyword spotting datasets, which are used to illustrate the performance of FL algorithms under non-IID conditions. The authors detail how these datasets help in understanding the impacts of data distribution on FL model accuracy and training efficiency. used in his paper a dataset from the Radiological Society of North America (RSNA), includes 5,786 chest X-ray images primarily sourced for a Kaggle competition aimed at advancing medical image analysis for pneumonia detection. This dataset is valuable in federated learning environments, allowing for the development and testing of models across different institutions while maintaining data privacy. Its real-world application, especially in training models to recognize pneumonia from X-rays, highlights its relevance and popularity in healthcare-focused ML tasks.

According to the FL frameworks reviewed in section 3.1., the datasets used in these frameworks can be seen in table 3.4:

Summarising all FL datasets reviewed in this research, table 3.4.1 can be created:

After completing this section, a secondary population was found, FL datasets. For the experimentation part, RSNA Chest X-ray, MNIST, and a synthetically generated dataset will be used. Again, in an attempt to minimize the bias inherent in experimentation, using these datasets, which are widely accepted by the FL community, may mitigate the bias.

This section aligns with the second research objective, developing a FL server. The Papers reviewed here share a common approach, they build a FL server using Flask. This approach will be used in section seven when developing the FL server. Additionally, to consolidate the concept of flask, two books have been reviewed.

implemented federated deep learning to yield improvements in privacy, latency, and bandwidth. The system consists of a group of clients or smart doorbells and a server. Each smart doorbell will capture video, preprocess, and train using TensorFlow Lite towards a local model. The fed local models are sent to a federated server for aggregation. The aggregated global model is then distributed back to the clients for object detection. For the server-side, Flask was used and then containerized with Nginx and Gunicorn, then deployed on AWS EC2. This reduces the cost in communication and ensures data privacy since raw video data is stored in local devices.

developed a FL framework for analysis in mental health, focusing on the prediction of depression. The server-side has been developed using Python and Flask, where the FL process is controlled by a global model that uses *TensorFlow* and *Keras*. It is a multi-class classification model distributed to client devices. Clients train the model locally over their data and hence maintain raw data on the device. It sent model updates back to the server, which aggregated these updates with the method of federated averaging. To ensure greater privacy, security measures included differential privacy and SSL encryption, though the adoption of these was optional. The system further contained an alternate questionnaire and an optional chatbot that assisted in the collection of more information besides supporting the user.

have introduced *Micro-FL* that overcomes scalability and fault tolerance with the help of a microservices architecture. The server implementation uses Docker containers orchestrated by Kubernetes. Several microservices are included in the system: a user interface developed using Flask and Nginx, a communication service using Apache Kafka, a database service using Elasticsearch, and an aggregator for combining model updates. Clients register via a web interface, train local models, and send updates to the server. Server aggregation combines these updates into a global model using the *FedAvg* algorithm. This makes the system more scalable and tolerant to failures because of the modular approach taken towards isolation of components, such that individual scaling and upgrades are independent.

detail the FL server implementation in chapter four by focusing on the aggregator, database, and communication handlers. The process is initialized by importing all the required libraries followed by the definition of the Server class to maintain agent registration, global model synthesis, and message handling. Configuration is initialized through a JSON file that determines IP addresses, port numbers, and aggregation parameters. The *StateManager* class buffers local model data and performs the aggregation criteria, while the Aggregator class integrates local updates to the global model using *FedAvg*. State information is maintained at server levels for smooth operation. A pseudo database on SQLite is used for model data and performance metrics, while the database server is configured to manage model data emanating from the aggregator. It also has guidance on running the server, database, aggregator, and agents with provided configuration files defining the settings. The possible improvements regard redesigning the database, automating the registration of the models, and enhancing the performance metrics for comprehensively guiding on building a very basic but configurable FL server.

explains in chapter eleven how to build RESTful APIs using Flask, from introduction to Representational State Transfer (REST) architecture and its Create, Read, Update and Delete (CRUD) operations. He walks the reader through the creation of APIs by using Flask routing, handling HTTP methods, and returning responses. This chapter has also introduced testing tools such as *cURL* and Postman for API validation and further covered Flask-RESTful, an extension of Flask for structure and modular development of APIs. There are a few practical examples in the course: one in making a simple API for managing a book collection with the implementation of CRUD operations and testing using Postman and *cURL*. On chapter twelve details deploying Flask applications in production using cloud platforms such as Google App Engine and Heroku, from setup to deployment processes. This involves a discussion on configuration files like *requirements.txt* and *Procfile*. The simplicity of hosting Python apps on PythonAnywhere is also discussed. For dedicated servers, this walks through deploying Flask using *uWSGI* and *Nginx* in handling HTTP requests. It discusses the choice of host toward an adequate environment that will best suit the needs of an application and scaling.

In this section, a third population was identified: companies or institutions that use FL, with samples drawn from the technological and medical sectors. In the case of medical companies, hospitals can also be included as they extensively use FL. By combining the second and third populations, the result was a focus on datasets and companies. This will be used in Chapter 7 (Results) to further validate the artifact. Specifically, synthetically generated data for the technological dataset and data created from X-ray and *MNIST* datasets for the medical scenario will be employed. These datasets needed to have two variants, IID and non-IID, in order to validate the FL server similarly to the approaches found in the papers by and. The purpose of utilizing both IID and non-IID variants is to validate the artifact and observe the results produced using this approach.

This literature review gives a complete overview of different FL frameworks that are widely used and accepted by the academic community. These frameworks are very much for research use, and most importantly, they are not described with completion on how a real FL system works—most of them are very abstract and far from reality. The review brings up the fact that the algorithm most extensively used in FL is FedAvg, which merely averages local model updates to build a global model. Yet, in contradiction to the abstract nature of most of the frameworks, the AI4EOSC platform concretely sets a look into more realistic FL scenarios by means of practical implementation on AML. Although these were just simulations, they demonstrated different clients training local models and a central server that updated and aggregated the global model. Each of the clients and servers themselves had their own IP address, making this similar to a cross-device setting. Data sets used in FL popularly include MNIST, CIFAR, SHAKESPEARE, and synthetically generated data. Finally, after reviewing FL frameworks and real-world settings, it became clear that good development of an FL server might be performed in an FL server using Flask. Of course, companies will use more sophisticated methods, but this should be enough to serve as a proof of concept.

Furthermore, this review was instrumental in defining the ROs, as shown in table 3.6.:

To determine the first RO, it was necessary to explore existing FL frameworks. This review initiated the primary research. A primary population, FL frameworks was identified having a sample of five FL frameworks. Allowing experimentation to evaluate the implementability of the selected frameworks. FL algorithms are also tied to this RO, as they are inherent to each FL framework.

Aligned with the second RO, the rest of the sections address various aspects of FL frameworks and their application. In Section 3.3, the gap between FL frameworks and real-world FL settings began to close, as some cases in this section resembled the approaches private companies might use for FL projects. This prompted the development of an FL server that fully demonstrates a central server running on an IP address, with different clients connecting to this IP address, each running on separate ports. In Section 3.4, a secondary population was identified: the datasets. This facilitated sourcing the data for the experimentation, including the RSNA Chest X-ray, MNIST, and a synthetically generated dataset. Finally, Section 3.5 provides insights into implementing an FL server using Flask and identifies a third population, the clients. The sample includes companies or sectors that will be referred to as clients in the experiment. These sectors, such as medical (hospitals and pharmaceutical companies) and technological, extensively use FL as part of their continuous improvement and development.

The third RO involves comparing the findings from the first and second ROs. It is evident that all sections of the literature review are crucial for achieving this objective.

Lastly the sampling strategy carried out in this research is a sampling method non-probabilistic and sampling judgmental, it is necessary to address and remark that bias is embedded in the experimentation method and above populations were selected after carefully considering the following points:

* The frameworks selected are widely used and accepted by the FL community, researchers, and private sector.
* The datasets, RSNA Chest X-ray, MNIST, and a synthetic data are typically used in research and present in many FL experiments as the literature review has revealed.
* The technological and medical sectors, the first created the concept of FL and the second is experimenting heavily with this concept.

Given the reasons above, the chosen samples are representative of their entire population. Refer to Table 3.6.1 to locate each sample within its population, RO, and corresponding literature review section.

This chapter starts with the experimentation phase and evaluates the sample of *PySyft, FATE, Flower, FedML,* and *TFF*, that was discussed in section 3.1. Established the following criteria to evaluate open-source FL frameworks: ease of use and deployment, development, analysis capabilities, accuracy, and performance. However, due to time constraints, this research will focus only on ease of use and real-world applicability. Each aspect can be broken down as follows:

* Ease of use:
* Setup and Configuration: Evaluates the complexity of installing the framework.
* Adaptability to Various Use Cases: Evaluates how the framework can adapt to different business domains and use cases.
* Real-world applicability:
* Examples and Tutorials: Assesses the quality of examples and tutorials available to help new users get started and how close these are to real-world settings.

After evaluating all five FL frameworks the following scores were assigned (see Table 5):

This framework offers interaction with an API via JN. To start the evaluation, the repository was cloned. The documentation is clear, and support is available through Slack. The API itself, when accessed via a browser, does not offer any functionality; actions must be performed via JN. It is designed for programmatic use rather than manual interaction. The repository (OpenMined, 2019) contains twelve JNs that serve as tutorials.

The participants, also known as *PySyft* workers, include the data owner and the data scientists. The data has two variants: mock and private. Data scientists can only access and read the mock dataset. The first four JNs cover the basics of *PySyft*, including how to load and preprocess data securely, how scientists can submit code for remote execution for the owner to review and approve, and how data scientists can download their results.

The fifth JN shows how the data owner trains a multi-party computation model using *PyTorch*. The remaining JNs cover customizing policies for data access, handling multiple code requests for approval by the data owner, managing the data site register control flow, and granting access to new users. They also cover code history, blob storage, submitting Docker files, custom API notebooks, and resetting user passwords.

After reviewing this framework, the conclusion is that *PySyft* is a robust framework for privacy-preserving machine learning but is more suited for academic and research fields. None of the tutorials provided a real-world scenario where different devices train a model locally and a server aggregates the results. Instead, the framework focuses on privacy and user permission management rather than providing real federated learning scenarios.

FATE repository has good structure (FATE, 2021) and clear documentation guiding the user through its directories, examples and tutorials. Active support via issues and discussions in the repository enables users to look for help and contribute to the project.

To evaluate the framework two tutorials were evaluated. The first one was Hetero-NN Tutorial which leveraged the FATE Hetero-NN framework for training a NN model based on vertically partitioned data, where guest and host have different features of the same dataset. Essential FATE libraries were imported, and a context was created to configure the federated environment. The data is loaded from CSV, with labels from guests, and without labels from hosts, into the data frame format of FATE. In this process, based on the type of party, it initializes a model; that is, for a guest, it will initialize both the bottom and top models, whereas for a host, it initializes only the bottom model. By using *HeteroNNTrainerGuest* or *HeteroNNTrainerHost*, it will prepare the training of the model, where the function train trains the model, and the function predict predicts the outcome of the data set by applying a trained model. The run function coordinates the training and prediction, and the script is run with launch, which simulates a FL scenario.

The second tutorial, the *Hetero-SecureBoost* tutorial makes use of FATE's *Hetero-SecureBoost* scheme in that it trains the boosting tree model. Based on party type, initialization of the model is done: a guest initializes the model as *HeteroSecureBoostGuest*, a host as *HeteroSecureBoostHost*. The train function initiates the training loop, while the predict function utilizes a trained model for predicting outcomes over an input dataset. The run function drives both the training and predicting operations. The script is launched with launch for mimicking the federated learning setup. Both tutorials are successful in demonstrating federated learning by enabling the model training process among different parties without exchanging raw data, and hence ensures collaborative learning while keeping private data.

The settings in these scenarios are such that they fit well with real-world federated learning; thus, they ensure data privacy and security. Consequently, FATE can be applied in practical settings for federated learning. Final conclusion same as *PySyft* none of the tutorials provided a real-world scenario where different devices train a model locally and a server aggregates the results.

The Flower GitHub repository provides practical documentation to speed up the process of using this framework. It also has a large community on *Slack.* Thetutorials offer the option of using JNs as well as Python scripts for seamless command line deployment. Among the tutorials offered in Python scripts, two were selected: *vertical-fl* and *pytorch-from-centralized-to-federated.* These were chosen because they are closer to real-world FL cases, and moving away from JNs helps to achieve this.

The *vertical-fl* example uses the *Titanic* dataset to train simple regression models for binary classification. In VFL, each client holds different features of the same dataset, while the server retains the dataset labels. The task was to predict whether the passengers survived or not, with three clients capturing different features. Finally, the server aggregated each client using *FedAvg.*

The *pytorch-from-centralized-to-federated* example demonstrates the transition from a ML centralised setup to a FL setup using *Flower* and *Pytorch*. In the centralised setup, a CNN was trained using the CIFAR-10 dataset achieving 37.8% accuracy. The federated setup distributed the data across two clients training models locally. The server aggregated updates and improved accuracy in 48.9%. This demonstrates how FL can better generalise and improve accuracy.

The *Flower* framework is closer to real FL scenarios because it simulates clients and a server in a pragmatic way, making it easy to understand how each element is laid out. Flower surpasses *PySyft* and *FATE* in this regard. A drawback, however, is that users need to create environments and be familiar with the Linux console to deploy these examples.

*FedML* offers clear documentation and tutorials for deploying its experiments. It also has a broad support community on Slack. Several of its repositories contain code that has been used in real-world settings and academic publications. Two tutorials were evaluated *FedAvg* MNIST LR and the Heart Disease Example.

*FedAvg* MNIST LR demonstrates how to use the federated averaging algorithm for training a logistic regression model on the MNIST dataset under a cross-silo (horizontal) federated learning setup. The experiment has 1,000 clients, with each client training on a partitioned MNIST dataset. Model updates are centrally averaged to form a global model. This experiment is conducted over 100 communication rounds, with two clients participating in a round at a time. The results show an incremental improvement in the accuracy of the model with every round, reaching 99% by the last one, while the loss decreases up to 0.01. This is quite a good example of how federated learning can demonstrate its potential in the case of multi-client and centralized server situations.

The Heart Disease Example uses federated learning on a distributed Heart Disease dataset to illustrate its use in healthcare. The dataset is distributed across four centers: Cleveland, Hungary, Switzerland, and Long Beach V. The dataset holds data specific to each center. Experiments were run using a binary classification model over 10 communication rounds with FedAvg as the optimizer. The performance of the model, with respect to the Area Under the Curve (AUC), stabilizes around 0.7396, demonstrating the capability to handle binary classes.

Unlike PySyft, FATE, and Flower, FedML offers a platform for project management (open.fedml.ai); however, this feature was not evaluated due to time constraints. After evaluation, FedML appears to be the closest to real-world settings, as evidenced by its GitHub repository.

The last FL framework reviewed, TFF, presents a comprehensive GitHub repository, with documentation and tutorials that are easy to follow. The tutorials are JNs that can be run on a *Google Colab* or downloaded to a local computer for exploration, there are a total of twenty tutorials only two were selected for evaluation. TFF offers a robust package divided in two layers, FL and Federated Core (FC). The first layer provides high-level interfaces for integrating *Keras* or *non-Keras* machine learning models into the TFF framework. The second layer consists of lower-level interfaces that allow customization of algorithms by combining *TensorFlow* with distributed communication operators (TensorFlow Federated, 2024). Evaluating the tutorials, the first focused on image classification in a FL setting and the second demonstrated how to build a FL algorithm with TensorFlow.

FL for image classification tutorial demonstrates how to use the TFF high-level *tff.learning* API to perform federated learning on the EMNIST dataset, which is a federated version of the MNIST dataset. The process involves key steps: first, it prepares the non-i.i.d. data across multiple clients for federated learning. Then, a simple neural network is defined using *tf.keras* and is wrapped with TFF *tff.learning.models.VariableModel.* The model is trained using the *FedAvg* algorithm, which is implemented to operate over several training rounds in a federated setup. Finally, the tutorial concludes by evaluating the model's performance using federated evaluation methods, focusing on accuracy and loss metrics for both training and test datasets.

Building your own FL algorithm with TFF tutorial, offers an in-depth look at constructing a custom FL algorithm using TFF lower-level FC, which allows greater control over the learning process. It starts by explaining the four main components of federated learning: server-to-client broadcast, client update, client-to-server upload, and server update. The tutorial explains how to create custom federated algorithms beyond the standard APIs by using TFF low-level interfaces. A basic *FedAvg* algorithm was developed by defining the *initialize\_fn* and *next\_fn* functions, which integrate TensorFlow operations within the federated communication process. The tutorial wraps up by combining these elements into a custom iterative process for federated learning, including an evaluation of the model performance after a few training rounds.

Summarizing TFF, it is likely the most robust FL framework, but its tutorials are more suited to academic scenarios and are far from real-world applications.

In summary, all five FL frameworks offer a wide range of options for setting up FL systems. Based on the evaluation conducted, Table 5.6 presents their rankings based on an equal-weighted average.

*FedML* stands out as the best option, offering intuitive tutorials that closely mirror real-world settings and a website capable of orchestrating and controlling FL experiments. *Flower* and *FATE* closely follow, with their tutorials and seamless deployment making them very robust for experimenting with FL settings. While all five FL frameworks are focused on research, the evaluation determined that *TFF* and *PySyft* appear to be designed primarily for academic research, as their tutorials serve mainly as proofs of concept.

In conclusion, all five FL frameworks can be enhanced due to their open-source nature, allowing for the customisation of a FL system to match the specific requirements of a project.

Section 10.2., of the annex provides additional information on how to deploy and implement the tutorials for each of the FL frameworks evaluated.

This chapter describes how the FL server was built, including its architecture, components, communication protocols, server functions, and client coordination. A high-level overview of the project file structure is shown in Figure 6.1. The server is orchestrated by *server.py*, with clients connecting to the server via *client.py*. There are two scenarios for training: *medical* and *technological*. Data for these scenarios was generated using JNs stored in the *FLServer/JNs* directory. Finally, a front-end page (*index.html*) is provided to interact with the server. To format the page *styles.css* was used, and *script.js* gave the logic to interact with the server and dynamically update the HTML content.

The designed FL server has an architecture to fit in multiple client nodes, for this experiment five clients were connected to the server into different ports. The server coordinated the entire process, aggregating the global the model after local client training and sending back weights into the clients for further training. The architecture is illustrated in Figure 6.2.

The server was run across four scenarios, *technological* and *medical,* eachin its IID and non-IID variants. After the five clients connected, the training for the *Technological IID* scenario iterated over five rounds, followed by the same process for *Technological nonIID, Medical IID* and *Medical nonIID*. After the final training scenario, the server was shut down. A video is available to illustrate this process, as well as Figure 6.3.

This file includes fifteen functions responsible for tasks such as client registration, storing their details (client ID, host, and port), and updating the server when a registered client is ready to begin the training process. The server is prepared to coordinate with a client for specific training datasets (Technological *IID, Technological nonIID, Medical IID* and *Medical nonIID)*. It waits for all clients to be ready before instructing them to start. The training process begins by sending a signal to all clients to start their local training. After training, the server receives model updates from the clients, aggregates the model weights from all clients, and updates cumulative metrics over training rounds. Additionally, it updates client statuses on the server and debugs by returning the current state of all registered clients.

A logic was implemented to refresh the server, as hitting the reload page every time was not an option. The refresh function sets a flag indicating that the server state requires refreshing. Another important functionality was to refresh the server without disconnecting the server and client consoles, allowing for a smooth transition from one training scenario to another. Finally, the index function renders the main dashboard page, displaying the status and metrics of all registered clients, as well as the local and global model metrics.

This script comprises thirteen functions responsible for enabling data loading for the specified client and scenario (*Technological IID* or *non-IID* and *Medical* *IID* or *non-IID*). Additionally, models for each scenario are defined and compiled based on the selected scenario. Functions for training management include prepare, start, and run. After training has finished, the weights are sent back to the server, and a receive function updates the local client model. The *reset\_client* function resets the client's state, reloads the data, and re-registers the client back into the server, preparing it for the next round of training. The final function allows the server to shut down clients.

Communication within the FL app was facilitated using the HTTP protocol between the central node (server.py) and the clients. Two fundamental *REST* operations, *GET* and *POST*, were employed. The *GET* method was primarily used by the server to retrieve the current state of all clients and to check if they needed to refresh their state or restart training. The *POST* method was used on both the server and client sides. It facilitated communication by sending data whenever an endpoint was invoked, with tasks like client registration, sending model updates, or initiating the training process being examples. A key distinction between these methods is that *GET* requests are non-intrusive; they do not alter the server's state and are intended solely for querying and retrieving data. In contrast, POST requests can modify the server's state. Across both nodes, a total of eleven endpoints have been defined, as illustrated in Figure 6.3.

This section describes the ML models used within the FL server. The models were designed based on the identified client population. The samples, reflecting the majority of the literature review, were categorized into medical and technological scenarios.

The medical model utilised was a CNN, designed for image classification tasks in the medical domain. In this scenario, the data consisted of images labelled as *lung* or *not lung*. The CNN was trained to classify these images based on the labels. This approach aimed to emulate similar medical experiments observed while reviewing the FL frameworks. The model layers are detailed in Table 6.4.1, and model architecture is illustrated in Figure 6.4.1.

The CCN was implemented using the *TensorFlow* and *Keras* libraries and compiled using the categorical cross-entropy loss function optimised with the *Adam* optimiser.

The technological model employed was a NN designed to handle structured data for a binary classification task. The data was contained in a CSV file with seven features and a target column with two categories. This election tried to find a case where any *“Tech”* company could face a similar binary classification task, such as determining if a product could be potentially sold based on a binary target, if a mortgage can be given based on a binary target, etc. The model layers are detailed in Table 6.4.2, and model architecture is illustrated in Figure 6.4.2.

The NN was implemented using the *TensorFlow* and *Keras* libraries and compiled using the binary cross-entropy loss function optimised with the *Adam* optimiser.

A simple algorithm was introduced for this project, The Federated Weighted Average *(FedWAvg).* It was designed for the distributed task of training five clients in parallel within the FL server. The server aggregated updates sent by the clients using a weighted averaging method based on the number of data points. The aggregated global model was then distributed to all clients for the next round of training. As shown in Figure 6.6., the server initialises the global model with weighs *w0.* In each round, five clients participate, training the model locally and updating the weights *wt.* The server then collects the updated weights from all clients, computes a weighted average to update the global model, and finally sends the updated global model back to the clients.

The data collection process for this project was accomplished using JNs, with synthetic CSV datasets created for the technological scenario and images for the medical scenario. At this stage, it was necessary to determine the classification and category of the FL server. In terms of FL classification, it clearly does not fall under cross-silo due to the limited amount of data each client holds. While it closely resembles cross-device FL, since the clients are not actual devices, this research classifies it as cross-client FL. The categorization would be horizontal FL, as the clients share the same dataset feature space but differ in the samples they hold.

The technological data used in this project was synthetically generated, with the primary requirement being a binary target and features suitable for modelling a NN. To further validate both the model and the artefact itself, two sets of datasets were created: IID and non-IID. Python libraries such as *numpy*, *pandas*, and *Faker* were utilized for this purpose. For the IID datasets, each client received a dataset with normally distributed feature columns and a balanced binary target variable, each consisting of 5,000 rows. The logic for IID data generation is illustrated in Figure 6.7.1.

In the other hand, the non-IID datasets introduced variability and imbalance, with one class dominating the target variable and features multiplied by random factors to achieve non normally distributed data. The number of rows in these datasets ranged between 3,000 and 7,000 in total. The logic for non-IID data generation is illustrated in Figure 6.7.2.

In total, five clients were simulated, each receiving both an IID and a non-IID dataset, all of which were saved in CSV format within specific directories.

For this scenario, the RSNA Chest X-ray and MINST datasets were combined. The X-ray images were downloaded, resized, normalised and labelled as *Lung*. The MNIST dataset was similarly resized, converted to RGB, and labelled as *Not Lung.* This process is depicted inFigure 6.7.3.

For the IID scenario, datasets were created with an equal distribution of *Lung* and *Not Lung* images, randomly shuffled and then split into the training and test subsets. In the non-IID scenario, class imbalanced was introduced by varying the distribution of the images across clients, some clients would receive more X-ray images and others would receive more MNIST images. The images for each client were saved in folders labelled as IID and non-IID, each containing subfolders for the test and train subsets. This process is demonstrated inFigure 6.7.4.

A high-level overview of the data generation for the technological and medical scenarios is illustrated in Figure 6.7.5.

As introduced in Chapter 3.5., (FL Server Implementation), the artefact was validated according to the methodologies by and using IID and non-IID data. In sections below the results are presented for the technological and medical scenarios under both IID and non-IID variants.

The synthetic data generated for this scenario followed a normal distribution for the seven features, and the target variable was equally balanced across its two categories for the IID variant, as shown in Figure 7.1. Shapiro Tests (STs) were conducted for feature generation, and with an alpha (α) of 5%, the null hypothesis (H0) was accepted. This situation is unlikely to occur in real-life scenarios.

In contrast, the non-IID data was not normally distributed, and its target variable was unbalanced, as described in STs were conducted for feature generation, with α = 5%, resulted in the rejection H0. This scenario is likely to occur in real-life situations.

After five rounds, the medical scenario with IID data revealed the following results, performance trends by clients and global model.

Two clients showed improvement, client 2 and client 4. Client 2’s accuracy increased from 0.4927 in round 1 to 0.5092 in round 5, while its loss decreased from 0.8455 to 0.8204. Similarly, client 4’s accuracy improved from 0.5132 to 0.5160, and its loss reduced from 0.8316 to 0.8121. These clients demonstrated improvement, whereas clients 1, 3 and 5 experienced declines, with both accuracy and loss. The global model also showed a decrease in accuracy from 0.5098 in round 1 to 0.5069 in round 5, and a loss increase from 0.8268 to 0.8288. This suggests potential overfitting and indicates that the FL server design and NN architecture may not be optimal for IID data. The results are depicted in Figures 7.1.3 and 7.1.4.

The non-IID data variant produced the following results after training. Clients 1 and 5 showed the most consistent improvements. Client 1’s accuracy increased from 0.5706 in round 1 to 0.5779 in round 5, while its loss decreased from 0.8448 to 0.7945. Similarly, Client 5’s accuracy improved from 0.6950 to 0.7739, with a corresponding loss reduction from 0.7808 to 0.7205. Clients 2 and 3 experienced declines in accuracy and increases in loss. However, the most interesting insight came from Client 4. Its accuracy fluctuated significantly, rising from 0.2203 in round 1 to 0.8133 in round 3, then dropping to 0.2058 in round 4 before bouncing back to 0.8249 in round 5. This erratic performance might be due to communication issues, such as delays in sending accuracy metrics for aggregation or other communication-related problems, which should be addressed in future work. Overall, the model performed well with non-IID data, as the global model's accuracy improved from 0.5807 in round 1 to 0.6621 in round 5, with a corresponding decrease in loss from 0.7911 to 0.7649. The results are shown in Figures 7.1.5 and 7.1.6.  
As explained in Section 6.7.2, the medical data used was a combination of X-ray (lung images) and MNIST (number images) datasets. The distribution of training and testing images across clients for this scenario is described in Figure 7.2.1. In the IID variant, all sets were equally balanced, while in the non-IID variant, the sets were introduced with imbalances. *Balanced* refers to having 150 images per set, whereas in the *unbalanced* version, the number of images varied across sets.

The IID variant of the medical scenario produced the following results after training. All clients maintained an accuracy of one, and the global accuracy also remained at one. Jittering was introduced in Figure 7.2.2 to prevent the trend lines from overlapping. Additionally, the loss scores were stable and close to zero, as shown in Figure 7.2.3. These results are somewhat unrealistic due to the inherently simplified nature of the IID data.

The non-IID variant produced the following results after training. Clients 3 and 5 maintained an accuracy of 1.0000 throughout the training, and their loss values improved. Client 2 showed an increase in accuracy from 0.6957 in round 1 to 0.8125 in round 5, along with a slight improvement in loss from 0.3353 to 0.3261. However, Clients 1 and 4 did not perform well in either accuracy or loss. Overall, the global model’s accuracy decreased from 0.8769 in round 1 to 0.8601 in round 5, while the global loss increased from 0.1766 to 0.2093, highlighting the complexity of dealing with non-IID data. The results are depicted in Figures 7.2.3 and 7.2.4.