



University of
Zurich^{UZH}

Design and Implementation of a Lightweight Decentralized Federated Learning Platform

Timothy-Till Näscher
Zurich, Switzerland
Student ID: 18-935-338

Supervisor: Chao Feng, Jan von der Assen
Date of Submission: August 24, 2025

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich,

Signature of student

Abstract

Federated Learning (FL) offers a privacy-preserving alternative to traditional Machine Learning setups, where training data is shared with a central instance. In Centralized Federated Learning (CFL), participants share their model parameters / gradients with a central aggregation server which redistributes the newly aggregated global model to the participants. As it relies on a central server for model aggregation, it suffers from communication bottlenecks and single points of failure. Decentralized Federated Learning (DFL) addresses these issues by enabling direct peer-to-peer communication, meaning it eliminates single points of failure and communication bottlenecks. However, the practical adoption of DFL is often hindered by the complexity, high resource requirements, and poor developer experience associated with existing DFL platforms. Many solutions are difficult to deploy, extend, and debug, creating a significant barrier for researchers as well as practitioners. This thesis presents the design, implementation, and evaluation of a lightweight DFL Platform created to address these shortcomings. The core contribution is a hybrid DFL architecture that logically separates the orchestration of the scenario from the training, parameter exchange and aggregation. A central coordinator is used exclusively for lightweight tasks such as scenario orchestration, monitoring (collecting statistics, metrics and logs) and participant synchronization while the computationally intensive processes of model training and aggregation remain fully decentralized, occurring directly between peer-to-peer nodes. This design combines the scalability and fault tolerance of DFL with the manageability of a centralized system. The platform was engineered with a strong focus on modularity, extensibility, and usability. Key features include simplified deployment procedures that avoid complex containerization, an architecture that facilitates the use of standard debugging tools, and a basic web interface for real-time monitoring. The evaluation considering both qualitative aspects of the developer and user experience as well as quantitative measurements of orchestration overhead, confirms the effectiveness of the proposed approach. The results show that the platform significantly lowers deployment complexity while providing a user-friendly and highly extensible environment for conducting DFL experiments. Ultimately, this work delivers a practical solution that aims to make DFL more accessible and enables rapid, reliable and reproducible research.

Abstract (Deutsch)

Federated Learning (FL) bietet eine datenschutzfreundliche Alternative zu traditionellen Machine-Learning-Ansätzen, bei denen Trainingsdaten mit einer zentralen Instanz geteilt werden. Im Zentralisierten Federated Learning (CFL) teilen die Teilnehmer ihre Modellparameter/Gradients mit einem zentralen Aggregationsserver, der das neu aggregierte globale Modell an die Teilnehmer weitergibt. Da es für die Modellaggregation auf einen zentralen Server angewiesen ist, leidet es unter Kommunikationsengpässen und Single Points of Failure. Dezentralisiertes Federated Learning (DFL) behebt diese Probleme, indem es eine direkte Peer-to-Peer-Kommunikation ermöglicht, wodurch Single Points of Failure und Kommunikationsengpässe eliminiert werden. Die praktische Einführung von DFL wird jedoch oft durch die Komplexität, den hohen Ressourcenbedarf und die mangelhafte Developer Experience bestehender DFL-Plattformen erschwert. Viele Lösungen sind schwierig zu implementieren, zu erweitern und zu debuggen, was eine erhebliche Hürde für Forscher und Praktiker darstellt. Diese Arbeit stellt das Design, die Implementierung und die Evaluierung einer leichtgewichtigen DFL-Plattform vor, die mit dem Ziel entwickelt wurde diese Mängel zu beheben. Der hauptsächliche Beitrag ist eine hybride DFL-Architektur, die die Orchestrierung des Szenarios logisch von Training, Parameteraustausch und Aggregation trennt. Ein zentraler Koordinator wird ausschließlich für einfache Aufgaben wie die Szenario-Orchestrierung, die Überwachung (Sammeln von Statistiken, Metriken und Logs) und die Teilnehmersynchronisation verwendet, während die rechenintensiven Prozesse des Modelltrainings und der Aggregation vollständig dezentralisiert bleiben und direkt zwischen Peer-to-Peer-Knoten stattfinden. Dieses Design kombiniert die Skalierbarkeit und Fehlertoleranz von DFL mit der Verwaltbarkeit eines zentralisierten Systems. Die Plattform wurde mit einem starken Fokus auf Modularität, Erweiterbarkeit und Benutzerfreundlichkeit konzipiert. Zu den Hauptmerkmalen gehören vereinfachte Bereitstellungsverfahren, die eine komplexe Containerisierung vermeiden, eine Architektur, die die Verwendung standardmäßiger Debugging-Tools erleichtert, und eine grundlegende Weboberfläche für die Echtzeitüberwachung. Die Evaluation, die sowohl qualitative Aspekte der Entwickler- und Benutzererfahrung als auch quantitative Messungen des Orchestrierungsoverheads berücksichtigt, bestätigt die Effektivität des vorgeschlagenen Ansatzes. Die Ergebnisse zeigen, dass die Plattform die Komplexität der Bereitstellung erheblich senkt und gleichzeitig eine benutzerfreundliche und hochgradig erweiterbare Umgebung für die Durchführung von DFL-Experimenten bietet. Letztendlich liefert diese Arbeit eine praktische Lösung, die DFL zugänglicher machen und schnelle, zuverlässige und reproduzierbare Forschung ermöglichen soll.

Acknowledgments

I would like to thank my supervisors Jan von der Assen and especially Chao Feng for his continual support and expert guidance throughout the duration of this thesis. I am equally appreciative of Prof. Dr. Burkhard Stiller for allowing me to complete my thesis in the Communication Systems Group.

Furthermore, I would like to acknowledge the use of language assistance tools while working on the report. Tools such as Grammarly, Gemini and ChatGPT were utilized for paraphrasing, as well as for improving grammar and spelling.

Contents

Declaration of Independence	i
Abstract	iii
Abstract (Deutsch)	v
Acknowledgments	vii
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	3
2 Background	5
2.1 Federated Learning	5
2.1.1 Centralized Federated Learning	6
2.1.2 Decentralized Federated Learning	7
3 Related Work	9
3.1 Federated Learning Platforms	9
3.2 Discussion	13

4	Design	17
4.1	Requirements	17
4.2	Technology Stack	20
4.2.1	Programming Language	20
4.2.2	Web Framework	21
4.2.3	Data Persistence	22
4.2.4	Machine Learning	22
4.2.5	Containerization	23
5	Implementation	25
5.1	Development Environment	25
5.2	Coordinator	25
5.2.1	Project Structure	26
5.2.2	Core Logic	27
5.2.3	Services	31
5.2.4	REST API	31
5.2.5	Data Persistence	38
5.2.6	Frontend	40
5.2.7	Deployment	46
5.3	Participant	46
5.3.1	Structure	47
5.3.2	State Machine	48
5.3.3	Federated Learning	49
5.3.4	Deployment	56
6	Evaluation	57
6.1	Network Topologies	57
6.2	Data Manipulation Attacks	60
6.3	Overhead	68
6.4	Usability	72

<i>CONTENTS</i>	xi
7 Conclusion and Future Work	75
7.1 Conclusion	75
7.2 Future Work	76
Bibliography	77
List of Figures	80
List of Tables	81
A Usability Evaluation Questions and Answers	85

Chapter 1

Introduction

This chapter introduces the topic of the thesis as well as the problems it attempts to address. After introducing Federated Learning (FL), the motivation of this thesis, a description of the stages of work as well as the outline of the report are given.

1.1 Motivation

Federated Learning has emerged as an alternative to traditional machine learning by collaboratively training machine learning models across multiple devices. By training locally on each participant, federated learning bypasses privacy concerns due to aggregation of sensitive data at a single location.

FL traditionally uses a centralized approach known as Centralized Federated Learning (CFL), which relies on a centralized server to handle the creation and distribution of a new global model by aggregating the individual model updates submitted to it. A newly emerging alternative to CFL, known as Decentralized Federated Learning (DFL), eliminates the need for a centralized server by allowing devices to communicate and share model data among each other through a peer-to-peer network.

Despite the growing interest, existing FL platforms typically focus only on centralized scenarios and do not support DFL. The platforms that do offer DFL support have notable limitations. Many existing solutions suffer from complexities that negatively impact both user and developer experiences. Deployment procedures frequently rely heavily on complicated containerization technologies that, while simplifying the initial setup, increase resource demands and complicate the debugging process. Developers commonly face difficulties utilizing standard debugging tools, hindering effective testing and troubleshooting.

Furthermore, existing DFL platforms often feature fragmented and inadequately documented implementations, making them difficult to extend, customize, or maintain. These issues hinder rapid development cycles, reduce platform usability, and restrict their practical applicability in real-world federated learning scenarios.

Motivated by these shortcomings, this thesis aims to create a lightweight decentralized federated learning platform that addresses the identified gaps. The main goals include simplifying deployment, minimizing resource overhead, enhancing usability, and ensuring effortless extensibility. A main focus is improving the developer experience through allowing the use of standard debugging tools. Ultimately, the goal is to enable the quick setup, easy management and simple extendability of reproducible, and efficient decentralized federated learning scenarios.

1.2 Description of Work

The project is divided into the following stages:

1. **Research:** The first stage consists of gaining an overview of the differences and benefits of FL, CFL and DFL respectively, alongside their main challenges. Research is also conducted on possible features platforms may provide to alleviate them.
2. **Review and Comparison:** Following this, a number of currently available FL platforms are compared, assessing each platform's strengths and shortcomings. This comparison aims to identify important features and architectural decisions taken by each platform and assesses their impact on usability, scalability and developer experience.
3. **Feature Definition:** Based on this review, commonly missing features are identified and added to the list of features the developed framework shall provide. Particular focus is placed on lightweight design, developer-friendliness, ease of use and reproducibility.
4. **Proof of Concept:** A prototype is developed based on initial design ideas and the feasibility of the proposed „hybrid DFL“ concept. This approach uniquely combines fully decentralized training and model aggregation with centralized orchestration, monitoring, and scenario control. Initial tests confirmed the viability of this concept and clarified technical requirements for further development.
5. **Design and Implementation:** The resulting design emphasizes a modular, extensible architecture that supports ease of development and debugging. The central coordinator manages orchestration signals to synchronize decentralized nodes effectively, while participant nodes individually perform local training and aggregation tasks without sending training data to the coordinator. The framework should include key functionalities such as a basic web interface, scenario orchestration logic, dataset integration and performance statistics collection.
6. **Evaluation and Testing:** Tests and evaluations are performed for both qualitatively and quantitatively. Qualitative evaluations include a user study on the ease of use and breadth of features of the platform, while the quantitative analysis focuses on verifying the correctness and reproducibility of implemented mechanisms such as gossiping, aggregation and poisoning attacks.

1.3 Thesis Outline

In Chapter 2 the background of this thesis is introduced. It gives an overview of Federated Learning in general, as well as both the centralized and decentralized variants. Chapter 3 summarizes some related work, focusing on existing platforms and their core strengths and limitations in the context of this thesis. Chapter 4 presents the system design, including the identified requirements and the architectural decisions made for all components. Chapter 5 describes the implementation of the platform. In Chapter 6, the developed platform is evaluated. Qualitative aspects such as usability as well as quantitative metrics like orchestration overhead, are assessed. In the final chapter, the results of this work are summarized and conclusions are drawn.

Chapter 2

Background

In this chapter, some information on the topics relevant for the thesis is provided. After a broad overview of Federated Learning itself, the differences in its centralized and distributed versions are discussed.

2.1 Federated Learning

Federated Learning (FL) is a Machine Learning (ML) technique introduced in [1], designed to train a shared model collaboratively across multiple devices or nodes without transferring training data to a central server. Instead of collecting training data from each node and using it to train a model on a central server, FL performs local model training directly on each nodes data. Only the resulting model updates (gradients and / or parameters), are shared and aggregated into a global model, which can then be redistributed to the nodes. [1]

FL addresses limitations of traditional centralized machine learning approaches. Centralized systems collect and aggregate all training data in one location. This makes centralized training impractical or even legally infeasible in many real-world applications working with sensitive data, such as healthcare. FL overcomes these constraints by allowing data to remain under local control while still contributing to the improvement of a global model [1]–[3].

Beyond privacy concerns, FL works well in scenarios where data is neither independent nor identically distributed (non-IID). Examples include usage patterns generated by smartphones, hospitals storing patient records, or autonomous vehicles collecting driving data. By incorporating data from a wide range of sources, FL improves the robustness and generalization of machine learning models, especially when the data distributions vary significantly across clients [4].

In conclusion, Federated Learning offers an alternative to centralized training. It reduces communication overhead, enhances data security, and enables the use of data for training that would not otherwise be available. These characteristics make FL an attractive

choice for distributed ML systems that operate in settings with privacy constraints or in bandwidth-limited environments. [1], [5]. The following sections will examine the centralized and decentralized variants of FL.

2.1.1 Centralized Federated Learning

A typical *Centralized Federated Learning* (CFL) scenario consists of four main components: [6]

- **Central server or aggregator:** The central server orchestrates the training rounds and is responsible for aggregating the received local model updates into a new global model each round. [6]
- **Participating nodes:** Nodes are devices or participants, that have locally available training data and use it to train a local model based on the global model distributed to it earlier. [6]
- **Communication framework:** The communication framework handles the communications between the central server and the nodes, as well as the transfer of model updates between itself and the nodes. [6]
- **Aggregation rule:** This algorithm defines how the local model updates from the participating nodes are combined into a new global model during each training round. [6]

Together, these components of the CFL Scenario form the CFL process, which typically consists of 4 steps: [6]

1. **Initialization:** In the initialization of the scenario and the training loop, the global model is initialized by the central server and then distributed to all participating nodes. This model is used as the starting point of the first local training round. [6]
2. **Local training:** The model received during initialization is further trained by each participating node using the local data. When training is finished, the resulting parameters and / or gradients that form a model update are sent to the central server to be used in the aggregation step. [6]
3. **Aggregation:** During the aggregation step, the central server combines all received model updates using the aggregation rule into the next global model. [6]
4. **Model redistribution:** The „new“ global model created during the aggregation step is redistributed to all participating nodes. It is then used to replace the local model before the next local training step. [6] The steps 2, 3 and 4 form a round. A scenario may have as many rounds as needed to achieve the expected model performance. [6]

While CFL is a straightforward way of implementing federated learning, it has notable limitations. One drawback is the by-design reliance on the central server, which can become a performance bottleneck in large-scale scenarios. Also, the server represents a single point of failure. If it is compromised, the entire learning process may be interrupted or corrupted. This is particularly problematic in scenarios that demand high levels of reliability and data protection. [7].

2.1.2 Decentralized Federated Learning

In DFL scenarios, participating nodes communicate directly with each another to exchange model updates and manage orchestration. Instead of sending their local model updates to a central aggregator, the nodes use some form of peer-to-peer networking and communications to receive model updates directly from other participating nodes. Similar to CFL, these model updates are used to create a new global model which, depending on the exact implementation, may be the same across all nodes. [8]. The DFL scenario distributes both computation and communication load more evenly across the network and enhances the fault tolerance of the system [7], [9].

DFL systems build on a network topology, such as *Ring*, *Tree*, or *Fully Connected*. Figure 2.1 shows a selection of network topologies, where each node represents a participant in the scenario and each edge a possible communication path. The network topology defines which node is able to communicate with other nodes, or in other words, with which nodes each node can exchange its model parameters with. Over multiple communication rounds, model updates of a certain client propagate throughout the network and lead to an improvement of the local models of clients, that have no direct connection to this node [7], [9]. A main advantage of DFL is its robustness. Since most DFL network topologies do not have a central point of failure, the setup can tolerate dropouts of any participating node. Moreover, in critical or sensitive applications, the lack of a central entity reduces the attack surface and limits the risk of data leakage through server compromise [7], [10].

However, DFL also presents unique challenges. First, achieving model convergence in the absence of centralized coordination is difficult, especially when data distributions are non-IID across nodes. Second, increased communication overhead, particularly in densely or fully connected network topologies, reduces communication efficiency. Finally, the design of reliable and secure aggregation and communication protocols in a fully decentralized setting remains an active area of research [7], [9].

In summary, DFL applies the principle of federated learning to a setting without a single node controlling orchestration and aggregation. It offers better scalability and resilience but requires more sophisticated communication and aggregation mechanisms. Considering its advantages over a centralized setup, DFL is positioned well to be the foundation of the next generation of distributed ML systems. [7], [11]

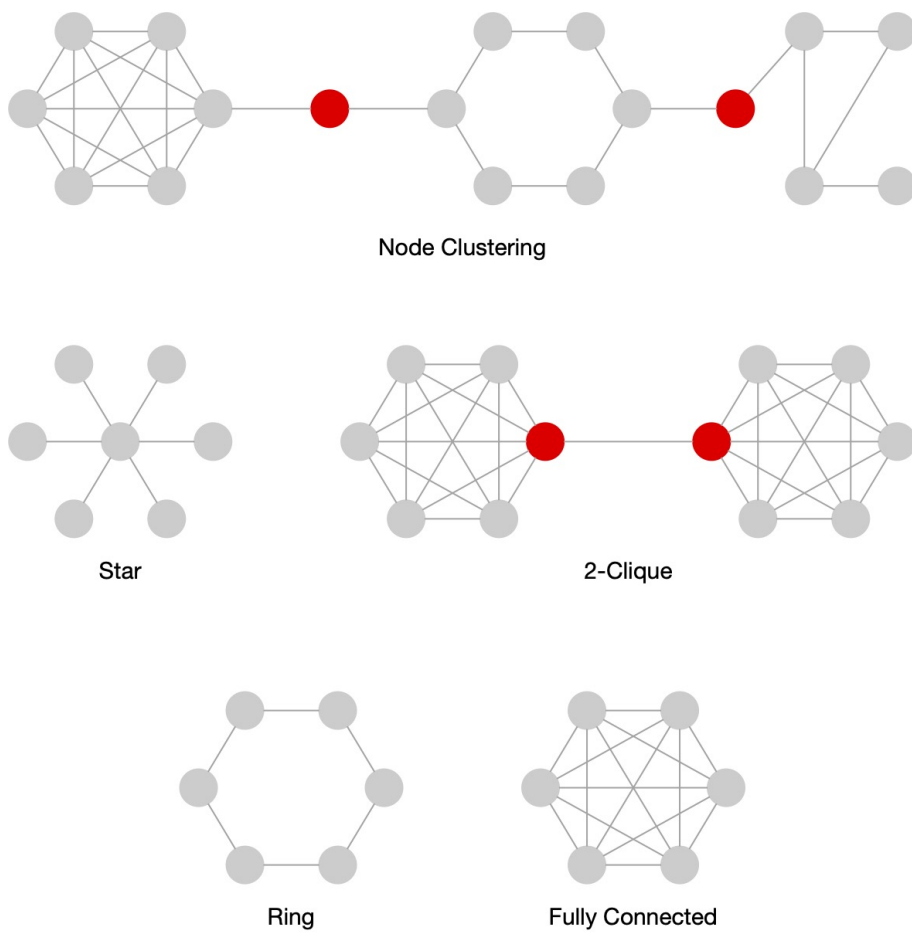


Figure 2.1: Network Topologies

Chapter 3

Related Work

This section introduces the most relevant FL platforms and compares their capabilities across several key criteria. Among others, these include the evaluation of their implementation of DFL, the supported topologies, the type of node execution, the underlying technology and the availability of source code.

3.1 Federated Learning Platforms

DFL Support

Various platforms for simulation of FL scenarios already exist, yet they differ significantly in richness of available features. Many of the frameworks only allow Federated Learning in its centralized version, lacking support for the decentralized case. Table 3.1 shows which frameworks allow for DFL training, as well as the available node topologies that can be used for simulation. 9 out of the total of 15 frameworks that were compared do not support DFL, offering only the classic CFL star topology. For frameworks that support DFL, an important aspect are the topologies in which DFL can be applied. FedStellar [12] and NEBULA [13] offer the widest range of topologies.

Table 3.1: Comparison of Federated Learning platforms by DFL support and topology.

Solution	DFL Support	Topologies
Flower ([14])	No	CFL ("Star")
FedStellar ([15])	Yes	Fully connected, Star, Ring, Random, Custom
NEBULA ([16])	Yes	Fully connected, Star, Ring, Random, Custom
HXSu DFL-Framework ([17])	Yes	Custom (manually configured)
p2pfl ([18])	Yes	Fully connected, Star, Ring, Line
FATE ([19])	No	CFL ("Star")
FedML ([20])	Yes	"Symmetric", "Asymmetric", Custom
FederatedScope ([21])	No	CFL ("Star")
OpenFL ([22])	No	CFL ("Star")
FedLab ([23])	No	CFL ("Star")
easyFL / FLGo ([24])	Yes	Custom
FedScale ([25], [26])	No	CFL ("Star")
PersonalizedFL ([27])	No	CFL ("Star")
EasyFL ([28])	No	CFL ("Star")
TensorFlow Federated ([29])	No	CFL ("Star")

Node Execution

An important consideration in terms of performance, scalability and ease of development is the architectural design used for the execution of individual nodes. Nodes can be modeled in various ways, such as:

- **Docker containers:** Containerizing nodes allows them to be isolated and easily reproduced, even across multiple devices. This approach simplifies dependency management and ensures that each node runs in a consistent environment. [30] However, it also introduces resource overhead and complexity. Debugging Docker containers is also very difficult. [31]
- **Processes:** Running nodes as individual processes is a more lightweight approach. This can lead to better performance and lower resource consumption. The downside is that it provides less isolation between nodes. Debugging is still challenging, as stalled nodes may affect the system beyond the node itself and might lead to timeouts and other unwanted behavior. [20]
- **Simulated:** Instead of simulating a DFL scenario with nodes operating in parallel, as they would in a real scenario, these frameworks decide on the order of node execution at the start of each round and simply execute them one after the other, without any parallelism. While this may not accurately reflect the complexities and challenges of a real-world decentralized system, this approach is the easiest to debug, as only a single process is required. [20]
- **Virtual Machines (VMs):** VMs provide the strongest isolation between nodes. However, they are generally more resource-intensive than containers, which can limit the number of nodes that can be simulated on a single machine. [32]

Table 3.2 shows the available node execution methods that each framework provides.

Table 3.2: Node execution models of Federated Learning platforms.

Solution	Node Execution
Flower ([14])	Docker / VM's / Processes ("Simulation")
FedStellar ([15])	Processes / Docker in later Versions
NEBULA ([16])	Docker
HXSu DFL-Framework ([17])	Processes
p2pfl ([18])	Ray (Framework for scaling dist. Apps)
FATE ([19])	Standalone (Simulation) / Docker ("Cluster")
FedML ([20])	Docker / Kubernetes / Processes
FederatedScope ([21])	Standalone (Simulation) / Docker
OpenFL ([22])	Docker / "Bare Metal" - Processes on Devices
FedLab ([23])	Standalone (Simulation) / Processes / Docker
easyFL / FLGo ([24])	Simulated
FedScale ([25], [26])	Docker / Processes ("Simulation")
PersonalizedFL ([27])	Simulated
EasyFL ([28])	Docker / Kubernetes
TensorFlow Federated ([29])	Simulated / partially agnostic

Node Communication

The choice of communication protocol between nodes in a federated learning system is another important aspect, as it affects efficiency, scalability, and ease of implementation. The frameworks surveyed employ a variety of mechanisms for communication and data serialization:

- **gRPC** (gRPC Remote Procedure Call) is a highly performant, open-source framework that makes use of Protocol Buffers (Protobuf) to share structured data across devices. It is a popular choice for FL platforms, as its efficiency with binary data make it well-suited for transferring large model parameters. [33]
- **REST** (Representational State Transfer) is an architectural style that uses standard HTTP methods and data encoded in JSON format for data exchange. While its ease of implementation make it a popular choice, it can be less performant due to the larger size of text-based JSON payloads compared to binary formats. [34]
- **Protobuf** (Protocol Buffers) is a language/platform-neutral method for serializing structured data that was designed to be more efficient than conventional, text-based formats such as JSON. [35]
- **IPFS** (InterPlanetary File System) is a peer-to-peer protocol used for storing and sharing data in a distributed file system. Instead of addressing content by its location, IPFS uses a content-based address, which is a hash of the file itself. Its distributed nature make it a natural fit for DFL, as it provides a way to share model updates without relying on a central server. [36]

- **Torch Distributed** is a module within the PyTorch library that was designed for distributed machine learning applications. It provides optimized communication primitives for tensor operations, making it an efficient choice for PyTorch-based FL frameworks. [37]
- **MQTT** (Message Queuing Telemetry Transport) is a communication protocol based on the publish/subscribe principle. While its publish/subscribe model can be used to share model updates with multiple nodes, the reliance on a centralized message broker to share models among clients are in conflict with the decentralized nature of DFL. [38], [39]
- **Unix Sockets** are a highly efficient communication method that operates in the OS kernel, eliminating overhead produced by networking protocols. While this results in faster and lower latency communication between nodes, it can only be used for simulations running on a single machine. [40]
- **Simulated** FL does not require any additional communication protocols as communication occurs directly within the memory of a single process. This is the simplest and most controlled method, but it does not reflect real-world network conditions.

Table 3.3: Languages, ML Libraries, and Node Communication Mechanisms of Federated Learning Platforms.

Solution	Language	ML Libraries	Node Communications
Flower ([14])	Python	Framework Agnostic	gRPC
FedStellar ([15])	Python	PyTorch	REST
NEBULA ([16])	Python	Lightning / PyTorch	REST / Protobuf
HXSu DFL-Framework ([17])	Python	PyTorch	IPFS
p2pfl ([18])	Python	PyTorch / Tensorflow	gRPC / Unix Sockets
FATE ([19])	Python	PyTorch	Torch Distributed
FedML ([20])	Python	PyTorch / partly Agnostic	gRPC / Protobuf / MQTT
FederatedScope ([21])	Python	PyTorch / Tensorflow	gRPC / Protobuf
OpenFL ([22])	Python	PyTorch / Tensorflow	gRPC / Protobuf
FedLab ([23])	Python	PyTorch	Torch Distributed
easyFL / FLGo ([24])	Python	PyTorch	Simulated
FedScale ([25], [26])	Python	PyTorch / Tensorflow	gRPC / Protobuf
PersonalizedFL ([27])	Python	PyTorch	Simulated
EasyFL ([28])	Python	PyTorch	gRPC / Protobuf
TensorFlow Federated ([29])	Python / C++	Tensorflow	Simulated / Protobuf

Development Activity

It is examined whether each platform is under active development by tracking commits, releases, and community engagement (GitHub Stars) / issue handling. If there are open issues without discussion older than 6 months or no commits and releases in the same time period, the development activity is considered ceased. The development activity data was gathered from the GitHub API on June 6th, 2025.

Table 3.4 shows the development status of each framework, as well as its popularity and source code availability. Of the 15 compared frameworks, only 7 are still under active development.

The most used frameworks (over 1000 GitHub stars) are Flower [14], FATE [19], FedML [20], TensorFlow Federated [29] and FederatedScope [21]. Only Flower and TensorFlow Federated are still in active development.

Table 3.4: Development Status, Popularity, and Open Source Availability of Federated Learning Platforms.

Solution	Under Active Development	GitHub Stars (9.6.2025)	Open Source
Flower ([14])	Yes	5973	Yes
FedStellar ([15])	No	14	Yes
NEBULA ([16])	Yes	64	Yes
HXSu DFL-Framework ([17])	11 Months	0	Yes
p2pfl ([18])	Yes	108	Yes
FATE ([19])	4 Months	5897	Yes
FedML ([20])	10 Months	3869	Yes
FederatedScope ([21])	2 Years	1430	Yes
OpenFL ([22])	Yes	780	Yes
FedLab ([23])	7 Months	773	Yes
easyFL / FLGo ([24])	Yes	594	Yes
FedScale ([25], [26])	2 Years	398	Yes
PersonalizedFL ([27])	2 Years	389	Yes
EasyFL ([28])	2 Years	298	Yes
TensorFlow Federated ([29])	Yes	2379	Yes

3.2 Discussion

This section summarizes the main points from the comparison of Federated Learning Platforms. The strengths and weaknesses in terms of DFL support, the node execution, the communication and development activity are examined.

DFL Support and Topological Flexibility

A key finding is that most FL Platforms have not adopted DFL yet. Out of the 15 examined frameworks, 9 only support CFL ("star"-topology). If only considering platforms that moved past an early prototype stage, the adoption rate of DFL is even smaller. It seems like the FL landscape inclines towards a centralized setup. Among the few platforms that do support DFL, FedStellar and NEBULA stand out by providing the most flexible range of topologies, including fully connected, ring, and custom configurations. This topological flexibility is essential for accurately modeling real-world decentralized systems

where centralized aggregation is not always feasible or desirable due to external constraints (data protection, legal reasons, ...).

Node Execution Models

Node execution architecture plays an important role in the fidelity and usability of FL simulations. Simulated execution models, which are somewhat common among the surveyed frameworks, provide ease of debugging and resource efficiency but fail to reflect the complexities of diverse deployments and communications in real-world FL systems. Conversely, Docker- or process-based models, as seen in FedML, FATE, and NEBULA, offer more isolation and concurrency but come with higher complexity and resource overhead.

Communication Protocols: From Centralized to Decentralized

The analysis of node communication methods further shows the centralization bias of most platforms. gRPC and REST seem dominant due to their maturity and ease of use. Due to scalability issues, they may not be ideally suited for a highly decentralized applications. Notably, only the "HXSu DFL-Framework", which seems to be just a proof-of-concept, employs IPFS, a content-addressed, peer-to-peer communication protocol that aligns well with DFL principles. These findings suggest a general under-exploration of network architectures and protocols that are designed for the support of fully decentralized FL systems.

Development Activity and Sustainability

Finally, examining the development activity and the popularity of these platforms reveals concern about the sustainability of the FL platforms. More than half the platforms (8 out of 15) show little to no activity in the last 6 to 12 months. This makes it difficult to rely on these projects long-term, as important updates, bug fixes or community support may not remain available. On the other hand, there are also very popular projects like Flower or FATE, that are consistently maintained and actively developed. However, both projects have shortcomings that prevent using them as the base for this thesis. For example, the codebases of both projects are complex and very large, which contributes to debugging, the implementation of new features and the development experience in general being quite lacking. Also, both lack any kind of DFL support. Smaller, lesser-known platforms suffer from minimal community engagement and limited documentation, further impeding adoption.

In summary, while there are available FL platforms, most share clear areas that need improvement. Many platforms lack full DFL support, have unreliable or unreproducible communication frameworks, have a very complex codebase or just don't provide the functionality necessary for practical use. A platform meeting the defined criterias mentioned in chapter 4 doesn't seem to exist. These findings indicate a strong need for a DFL framework that is actively maintained and lightweight, while still providing essential functionality.

Such a framework would significantly lower the entry barrier for experimenting with DFL scenarios and help advance research in DFL.

Chapter 4

Design

This chapter details the functionality that the developed framework should provide and which technologies were used to achieve the desired functionality. First, an overview is given of the requirements that the components of the framework should provide, ranging from scenario management and node behavior to the data types that should be logged. Following this the choice of libraries and technologies used for the framework are detailed and justified.

4.1 Requirements

Scenario Management

The system must support full lifecycle management of training scenarios via a web-based user interface. This includes the ability to:

- Create scenarios using a structured form interface
- Export and import scenario configurations in JSON format to ensure portability and reproducibility
- Launch and stop scenarios directly from the interface
- List all existing scenarios and inspect their configurations and statuses
- Monitor the active scenario in real time
- Evaluate scenario outcomes and delete obsolete or completed scenarios as needed

Coordinator

A central coordinator component is responsible for scenario orchestration and user interface functionality. Specifically, the coordinator must:

- Host the web-based scenario management interface
- Manage a registry of all participating nodes and track their statuses
- Orchestrate scenario execution using an event-driven model aligned with a defined state transition graph
- Accept incoming events pushed by nodes (e.g., status updates, metrics, logs)
- Distribute outgoing events to nodes and the user interface to ensure responsive, real-time updates

Core Platform

At its core, the platform must support decentralized execution of federated learning rounds. The following capabilities are required:

- A fully decentralized training loop, in which training, communication (via gossip protocols), and aggregation are handled exclusively by the participating nodes without coordinator involvement
- Support for multiple communication topologies, including ring, star, random, and fully connected networks
- Modular support for different aggregation strategies, datasets, and model architectures
- Built-in support for simulating adversarial attacks within the training and gossiping process
- Minimal overhead in terms of networking and setup time, enabling rapid deployment and execution
- Well-defined interfaces and abstraction layers for core components such as trainers, aggregators, datasets, and attack modules, allowing easy extension and integration of new functionality

Node / Participant

Each node (or participant) operates independently during training rounds and must fulfill the following requirements:

- Receive scenario configuration and networking information from the coordinator at initialization
- Communicate status updates, logs, and metrics back to the coordinator during execution
- Perform peer-to-peer model parameter exchanges using a decentralized gossip protocol. No aggregation or training communication should involve the coordinator
- Conduct local aggregation of received model updates at the end of each round

Monitoring, Logging

The platform must offer comprehensive real-time monitoring features, including:

- A scenario-level dashboard displaying overall progress, phase status, and scenario metadata.
- A node-level status table showing detailed progress for each individual node.
- A scrollable live log feed with filtering capabilities by scenario, node, and log level. This feed should be available as a standalone view.
- A dynamic metrics grid populated via WebSocket, displaying reported values such as accuracy, loss, communication stats, and more from each node in real time.

Metrics

Structured logging and efficient metric handling are essential for observability and debugging:

- All logs must be collected in a structured format, including metadata such as unique identifiers, severity levels, messages, scenario ID, timestamps, and sender identity.
- The system must provide both historical access and live log/metric streams via the web interface.
- Integration with Weights & Biases (wandb) is required for experiment tracking and visualization.
- While TensorBoard integration was initially considered, practical constraints with merging tfevent files from multiple nodes proved prohibitive. Each node would need to upload its logs to the coordinator, which would then merge and serve them via a unified TensorBoard instance. This approach was tested but abandoned due to complexity and inconsistent results.

Goals

The platform is guided by both qualitative and quantitative design goals:

Qualitative Goals:

- **Decentralized Aggregation:** The training and aggregation process must be completely independent of the coordinator
- **Modular Architecture:** New datasets, models, topologies, and attacks should be easy to add via a well-structured interface
- **Extensibility:** The platform should allow seamless addition of new logging mechanisms, metrics, and scenario types
- **Lightweight Design:** The system should avoid large, complex dependencies (as seen in projects like Flower) and require minimal setup
- **Flexible Deployment:** The platform must be runnable in diverse environments, including Docker containers and bare-metal systems

Quantitative Goals:

- **Scalability:** The platform should function reliably with at least 15 participating nodes.
- **Overhead Efficiency:** The system should complete a single training round using a dummy model in approximately $5 \times N$ seconds, where N is the number of nodes.

4.2 Technology Stack

This section outlines the decisions taken in terms of library usage for the system architecture. After selecting the programming language used for development of the framework, the choice of key libraries for the WebUI and backend tasks is justified.

4.2.1 Programming Language

The choice of programming language used for the FL framework hinges on the availability of pre-existing libraries to simplify implementation of many of the framework's components.

The defacto standard programming language for development and research of ML/AI applications is Python. This is due to its widespread community support and the availability of different libraries for such tasks, such as PyTorch or TensorFlow. While ML libraries exist for other programming languages such as R, Julia or C++, they tend to

lack the widespread support and richness of features that Python libraries provide. Most of the DevOps platforms used for tracking and evaluation of ML experiments, such as WandB or CometML, are also only officially available for Python. While community-led contributions sometimes provide unofficial bindings for other languages, they are not guaranteed to be maintained long-term, making them less attractive for the development of a long-lived platform.

Next to the ML aspect, the FL framework also requires a variety of additional functionality, such as intra-node communication, synchronization across rounds, database management, etc. The Python ecosystem has matured far enough to offer robust and efficient solutions for each of these components.

As seen in chapter 3, FL frameworks are almost exclusively developed in Python, with only one framework providing support for a different language (C++).

Python's simple syntax and dynamic typing support simple prototyping and iterative development, while its cross-platform nature also guarantees that it can run on a wide variety of operating systems with minimal to no modification. All these factors make Python stand out as the only reasonable choice for this project.

4.2.2 Web Framework

The coordinator is responsible for managing multiple concurrent tasks, ranging from handling participant messages and broadcasting websocket updates to serving API endpoints for the scenario management.

The web framework used should minimize the overhead induced by the task concurrency. FastAPI was chosen for this purpose. It is built on `asyncio` and `starlette`, lightweight libraries for concurrency and data routing respectively. As such, FastAPI enables efficient usage of non-blocking I/O, which makes it a suitable choice for handling many connected nodes at the same time, while reducing overhead from threading and serving websockets to a minimum.

FastAPI offers extensive support for websockets out of the box, while alternatives either require additional, external libraries (e.g. Flask¹) or provide feature bloat that unnecessarily slows down the performance (e.g. Django²).

Next to its lightweight structure that offer good performance and low latency, the modern, Pythonic coding style used for FastAPI offers type hints, which can be used to automatically validate requests and responses through the integration with Pydantic³.

¹<https://flask.palletsprojects.com/en/stable/>

²<https://www.djangoproject.com>

³<https://docs.pydantic.dev/latest/>

4.2.3 Data Persistence

Since the framework should be able to persist data across multiple sessions, a database is required to keep track of data generated from simulation experiments.

The data generated by experiments consists of configuration parameters for the setup, as well as ML metrics obtained during the federation rounds for a given scenario. Since the data that should be collected may change depending on the type of scenario, a flexible data storage solution is necessary for this "semi-structured" data.

MongoDB was chosen for this purpose, as its document-based, JSON-like storage model is well-suited for storing this type of data. It can easily work with varied data structures from one experiment to another, without the need for complex schema migrations that would be necessary in a traditional, relational database. MongoDB's schema-less nature allows for highly agile development and prototyping, making it particularly suitable for the inherently exploratory nature of research that the framework is targeted at.

Moreover, the ecosystem around MongoDB provides extensive Python support through the Motor⁴ library. It is described as a "coroutine-based API for non-blocking access to MongoDB", which precisely matches the requirements of the FastAPI-based coordinator.

Another benefit of MongoDB is its support for "Change Streams", which works similarly to the Publish/Subscribe model used for many decentralized IoT applications. The Python backend can subscribe to real-time database updates, which allows the framework to provide instant feedback to the user interface whenever new data comes in. This helps guarantee a responsive and fluid user experience, with new data instantaneously displayed on the scenario dashboard. A prerequisite for using Change Streams is that the MongoDB instance must be initialized as a "replica set". Since data redundancy is not required otherwise, a workaround was achieved by using a Docker Compose file with two containers: one running the primary MongoDB instance, and a second, temporary container that uses the MongoDB CLI to initialize the replica set on the main instance before exiting.

4.2.4 Machine Learning

The core purpose of the proposed framework is to simplify decentralized machine learning experiments. In that regard, the choice of ML library is an important aspect to consider, as the framework must integrate with standard ML workflows while remaining agnostic to the specific model architecture or dataset used.

The most common ML libraries are PyTorch⁵ and TensorFlow⁶. PyTorch was chosen as the ML library for this framework, as section 3 shows that it is commonly used in related FL frameworks. Development with PyTorch is also considered to be more Pythonic and tends to better align with the framework's goals of rapid prototyping, developer-friendliness, and research-oriented experimentation.

⁴<https://motor.readthedocs.io/en/stable/>

⁵<https://pytorch.org/>

⁶<https://www.tensorflow.org/>

4.2.5 Containerization

To ensure reproducibility across deployments and different environments, the framework is designed with encapsulation in mind. Docker is a common choice for bundling software into "containers", ensuring that application dependencies are met without requiring additional user intervention. Containers are standardized, executable components that bundle the application source code together with a minimal operating system, allowing application deployment regardless of hardware configuration.

Chapter 5

Implementation

This chapter describes the implementation process of the FL framework. In the first half of the chapter, the implementation process of the Coordinator is described. After an overview of the Coordinator code structure, the state machine that controls the federated learning process is introduced.

Next, the structure and implementation process of the individual participant nodes is elaborated further. After an overview of the code structure, the behavior is described based on the state machine

5.1 Development Environment

The proposed framework was developed using the JetBrains PyCharm IDE. There are various benefits to working with an integrated development environment, ranging from powerful autocompletion and debugging features to run configurations and readily available plugins for e.g. viewing databases. By reducing the amount of encapsulation, e.g. through Docker containers or programmatically launched processes, breakpoints may also be set for individual nodes by launching them as debug tasks.

5.2 Coordinator

As the name suggests, the Coordinator is responsible for managing and coordinating the FL process across the participant nodes, without directly intervening in the training process itself.

This section describes the functionalities and their implementation provided by the Coordinator.

5.2.1 Project Structure

The following tree lists the structure of the Coordinator codebase:

```
coordinator
├── .dockerignore
├── .gitignore
├── Dockerfile
├── README.md
├── requirements.txt
├── app/
│   ├── main.py
│   ├── api/
│   │   ├── participants.py
│   │   ├── scenarios.py
│   │   ├── logs.py
│   │   ├── monitor.py
│   │   ├── orchestrator.py
│   │   └── confusion_matrices.py
│   ├── core/
│   │   ├── states.py
│   │   └── topology.py
│   ├── db/
│   │   └── mongo.py
│   ├── frontend/
│   │   ├── pages.py
│   │   ├── static/
│   │   │   ├── css/
│   │   │   │   └── style.css
│   │   │   ├── images/
│   │   │   │   └── favicon.ico
│   │   │   └── js/
│   │   │       └── main.js
│   │   └── templates/
│   │       ├── base.html
│   │       ├── index.html
│   │       └── scenario.html
│   └── services/
│       ├── log_service.py
│       ├── scenario_service.py
│       ├── statistics_service.py
│       └── wandb.py
```

Next to necessary configuration files in the root directory, the contents of the Coordinator functionality are grouped by task in the `app/` directory. There are directories for:

- `app/api`: Defines the REST API through which participant nodes communicate with the Coordinator.

- **app/core**: Contains the core application logic, including the state machine for scenarios and network topology management.
- **app/db**: Handles the connection and interaction with the MongoDB database
- **app/frontend**: Manages the web-based user interface.
- **app/services**: Contains services that provide specific functionalities, such as generating Docker Compose files, managing scenarios and handling logs.

5.2.2 Core Logic

The core logic of the Coordinator is located in the **app/core** directory. It contains the implementation of the finite state machine, which is responsible for the entire scenario execution flow. The state machine is composed of several states, each representing a specific phase of the scenario. The state transitions are triggered by the participant nodes' actions, which are communicated to the Coordinator via the REST API. While the Coordinator and participant nodes mirror their current state in the state machine, they behave differently during each state.

State Machine Implementation

To manage the system behavior during the scenario, a finite state machine is implemented. Each state is represented by a class that inherits from a base **State** class, implemented in **state/base.py**. The **State** class defines a common interface for all states, which contains the name of the state and a **run** method. The **run** method contains the logic for the specific state and determines the next state in the flow. Figure 5.1 shows the designed state machine. Since the federated learning process is chronologically structured into multiple stages (setup, training, gossip, aggregation), a state machine is a natural fit for this problem.

The following is a list of the states implemented in the Coordinator and their behavior:

1. **WAITING_FOR_NODES**: This is the initial state of the state machine. In this state, the Coordinator waits for all participant nodes to connect to the orchestration Web-Socket. Once all nodes are connected, the state transitions to **ALL_NODES_CONNECTED**.
2. **ALL_NODES_CONNECTED**: In this state, the Coordinator sends the list of all participant nodes to each node. This allows the nodes to establish connections with each other. The state then transitions to **START_TRAINING**.
3. **START_TRAINING**: This state marks the beginning of a training round. The Coordinator sends a message to all participant nodes to start the training process. The state then transitions to **WAITING_FOR_TRAINING_DONE**.

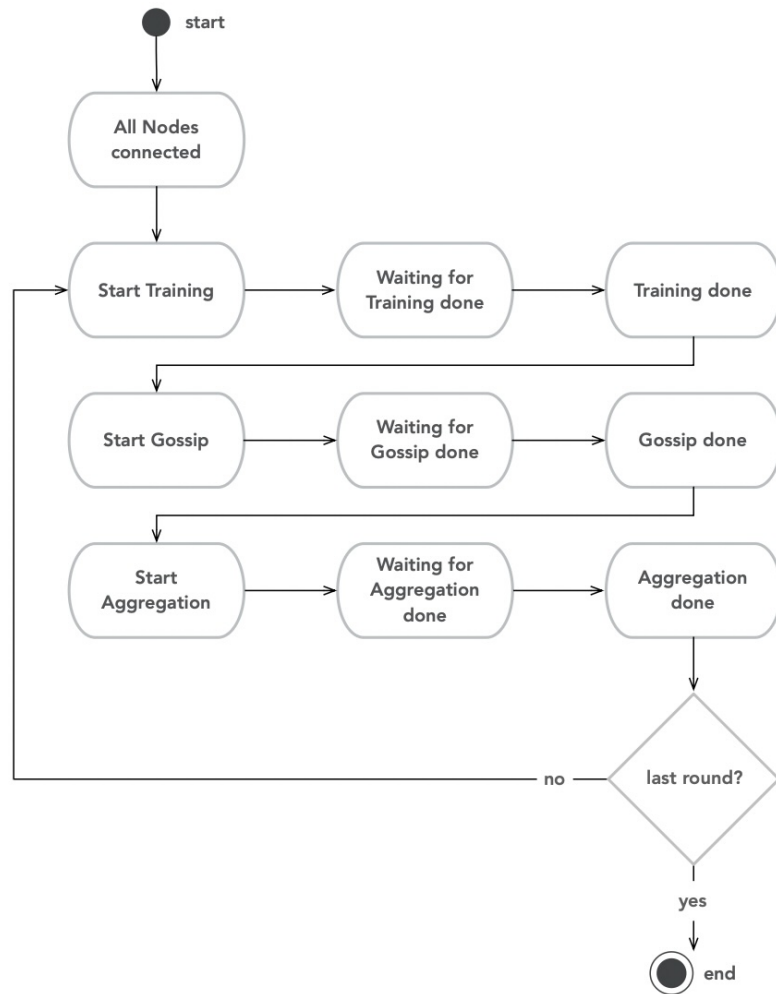


Figure 5.1: State machine for scenario flow

4. **WAITING_FOR_TRAINING_DONE**: In this state, the Coordinator waits for all participant nodes to finish their training process. Once all nodes are done, the state transitions to **TRAINING_DONE**.
5. **TRAINING_DONE**: This state marks the end of a training round. The state then transitions to **START_GOSSIP**.
6. **START_GOSSIP**: This state marks the beginning of a gossip round. The Coordinator sends a message to all participant nodes to start the gossip process. The state then transitions to **WAITING_FOR_GOSSIP_DONE**.
7. **WAITING_FOR_GOSSIP_DONE**: In this state, the Coordinator waits for all participant nodes to finish their gossip process. Once all nodes are done, the state transitions to **GOSSIP_DONE**.
8. **GOSSIP_DONE**: This state marks the end of a gossip round. The state then transitions to **START_AGGREGATION**.
9. **START_AGGREGATION**: This state marks the beginning of an aggregation round. The Coordinator sends a message to all participant nodes to start the aggregation process. The state then transitions to **WAITING_FOR_AGGREGATION_DONE**.
10. **WAITING_FOR_AGGREGATION_DONE**: In this state, the Coordinator waits for all participant nodes to finish their aggregation process. Once all nodes are done, the state transitions to **AGGREGATION_DONE**.
11. **AGGREGATION_DONE**: This state marks the end of an aggregation round. If there are more rounds to go, the state transitions to **START_TRAINING**. Otherwise, the state transitions to **SCENARIO_FINISHED**.
12. **SCENARIO_FINISHED**: This is the final state of the state machine. In this state, the Coordinator sends a message to all participant nodes that the scenario is finished.

Topology Builder

```

1  def build_neighbour_map(nodes: list[dict], topo: str) -> dict[str, list[
2  dict]]:
3      """
4      Builds neighbour map {uid: [peer, ...]} according to a given
5      topology.
6
7      Args:
8          nodes: List of node dicts. Each must contain a unique string key
9          "UID".
10         topo: Topology name. One of: "fullyconnected", "ring", "star", "
11         random"
12
13     Returns:
14         Mapping from each node's UID to a list of peer node dicts (same
15         objects).

```

```

13  Raises:
14      KeyError: If a node is missing the "UID" key.
15      ValueError: If UIDs are not unique.
16
17  Notes:
18      - Invalid `topo` falls back to "fullyconnected"
19      """
20  n = len(nodes)
21  if n < 1:
22      raise ValueError("At least 1 node is required to build a
23  neighbour map.")
24  uid_at = [p["UID"] for p in nodes]
25  idx_of = {p["UID"]: i for i, p in enumerate(nodes)}
26
27  def peers(index: int, req_topo: str) -> List[dict]:
28      if req_topo not in {"fullyconnected", "ring", "star", "random"}:
29          logging.error(
30              f"[TopologyBuilder] Unknown / Not Implemented topology {
31              req_topo}. Falling back to fully-connected."
32          )
33          req_topo = "fullyconnected"
34
35      if req_topo == "fullyconnected":
36          return [p for p in nodes if p["UID"] != uid_at[index]]
37
38      elif req_topo == "ring":
39          if n <= 1:
40              return []
41          prev_i = (index - 1) % n
42          next_i = (index + 1) % n
43          neighbour_indexes = {prev_i, next_i}
44          neighbour_indexes.discard(index)
45          return [nodes[i] for i in neighbour_indexes]
46
47      elif req_topo == "star":
48          return [p for p in nodes[1:]] if index == 0 else [nodes[0]]
49
50      elif req_topo == "random":
51          pool = [p for p in nodes if p["UID"] != uid_at[index]]
52          k = min(2, len(pool))
53          return random.sample(pool, k)
54
55  return {uid: peers(i, topo) for uid, i in idx_of.items()}

```

Listing 5.1: Topology Builder

The `app/core/topology.py` file contains the logic for building the network topology of a given scenario. Once a scenario is launched and all participant nodes are connected, the Coordinator shares information on the configured topology with all participant nodes. The following topologies are supported for now:

- **Fully Connected:** All nodes are connected to each other.
- **Ring:** Nodes are always connected to two neighboring nodes, forming a ring, i.e a single, closed path.

- **Star:** Consists of a central hub node and an arbitrary number of leaf nodes. Each leaf node is only connected to the hub node.
- **Random:** Each node is connected to a random subset of the other nodes.

The topology is built by the `build_neighbour_map()` function, which takes a list of nodes and the desired topology as input and returns a dictionary that maps each node's UID to a list of its neighbors.

5.2.3 Services

In the `app/services` directory, scripts are provided for the following:

- `docker_compose_generator.py`: Generates Docker compose files for a given scenario configuration
- `log_service.py`: Provides functionality for logging messages to the database and displaying them in the web UI.
- `scenario_service.py`: Manages the lifecycle of scenarios. It is used to create new scenarios from user-defined configurations and to safely delete past scenarios and their associated data (logs, statistics, etc.).
- `statistics_service.py`: Responsible for storing metrics obtained during a scenario (e.g accuracy, loss) in the database, which can then be used for further analysis or presentation in the web UI.
- `wandb_service.py`: Provides integration to the Weights & Biases (WandB) platform.

5.2.4 REST API

The Coordinator exposes a REST API for managing and monitoring the federated learning process. This API is the primary interface for interacting with the Coordinator, which enables access to the platform's functionalities over the web-based user interface.

The REST API is defined by separate files for:

- Scenario management (`scenarios.py`)
- Participant node registration and management (`participants.py`)
- Participant node orchestration (`orchestrator.py`)
- Logging (`logs.py`)
- Experiment monitoring (`monitor.py`)

- Generating confusion matrices (`confusion_matrices.py`)
- Storing statistics for the currently running scenario (`statistics.py`)

Table 5.1 shows the available API endpoints at each path, with a description of its purpose.

Table 5.1: API Endpoints Overview by Functional Group

Group	Method	Path	Short Description
Scenario Management	POST	<code>/store_config</code>	Creates a new scenario configuration.
	POST	<code>/launch_scenario/{scenario_id}</code>	Launches a specific scenario.
	POST	<code>/stop_scenario</code>	Stops the currently running scenario.
	DELETE	<code>/scenario/{scenario_id}</code>	Deletes a specific scenario.
	DELETE	<code>/scenarios</code>	Deletes all scenarios.
	GET	<code>/scenario/{scenario_id}/export</code>	Exports a scenario to a JSON file.
	POST	<code>/scenario/import</code>	Imports a scenario from a JSON file.
Participants	POST	<code>/join_scenario</code>	Allows a node to join a scenario.
	POST	<code>/report_status</code>	Receives the status of a node.
	POST	<code>/report_statistics</code>	Receives a statistic from a node.
	POST	<code>/report_confusion_matrix</code>	Receives a confusion matrix from a node.
	POST	<code>/logging</code>	Stores a log entry from a node.
Web UI / Sockets	WEBSOCKET	<code>/ws/orchestrator</code>	Facilitates orchestrator-node communication.
	WEBSOCKET	<code>/ws/monitor</code>	Monitors the currently running scenario.
	WEBSOCKET	<code>/ws/logs</code>	Streams logs in real time.
	WEBSOCKET	<code>/ws/statistics</code>	Streams statistics for a scenario.

POST /store_config*Creates a new scenario configuration.***Form Data**

Parameter	Type	Required	Description
scenario_name	string	Yes	Name of the scenario.
scenario_description	string	Yes	Description of the scenario.
rounds	integer	Yes	Number of rounds for the scenario.
dataset	string	Yes	Dataset to be used.
trainer	string	Yes	Trainer to be used.
learning_rate	float	Yes	Learning rate for the model.
batch_size	integer	Yes	Batch size for training.
epochs	integer	Yes	Number of epochs for training.
node_count	integer	Yes	Number of nodes in the scenario.
topology	string	Yes	Network topology.
device	string	Yes	Device to be used for training.
aggregator	string	Yes	Aggregator to be used.
attack	string	Yes	Attack to be simulated.
source_class	integer	No	Source class for label flipping attack.
target_class	integer	No	Target class for label flipping attack.
deployment	string	Yes	Deployment type.

Responses

- 303 See Other: Redirects to the newly created scenario's page.

POST /launch_scenario/{scenario_id}*Launches a specific scenario.***Path Parameters**

Parameter	Type	Required	Description
scenario_id	string	Yes	The ID of the scenario to launch.

Form Data

Parameter	Type	Required	Description
force	integer	No	Whether to force launch if another scenario is running. Defaults to 0.

Responses

- 303 See Other: Redirects to the scenario monitor page on successful launch.
- 400 Bad Request: If another scenario is running and force is not set to 1.

POST /stop_scenario*Stops the currently running scenario.***Responses**

- 303 See Other: Redirects to the scenarios list page.
- 400 Bad Request: If no scenario is currently running.

DELETE /scenario/{scenario_id}*Deletes a specific scenario.***Path Parameters**

Parameter	Type	Required	Description
scenario_id	string	Yes	The ID of the scenario to delete.

Responses

- 200 OK: Returns a confirmation message.

DELETE /scenarios*Deletes all scenarios.***Responses**

- 200 OK: Returns a confirmation message.

GET /scenario/{scenario_id}/export*Exports a scenario to a JSON file.***Path Parameters**

Parameter	Type	Required	Description
scenario_id	string	Yes	The ID of the scenario to export.

Responses

- 200 OK: Returns the scenario as a JSON file.
- 400 Bad Request: If the scenario ID is malformed.
- 404 Not Found: If the scenario is not found.

POST /scenario/import

Imports a scenario from a JSON file.

Request Body

A JSON object representing the scenario to import.

Responses

- 200 OK: Returns a confirmation message with the new scenario ID.
- 400 Bad Request: If the import fails.

POST /report_confusion_matrix

Receives a confusion matrix from a node.

Request Body

Parameter	Type	Required	Description
uid	string	Yes	The UID of the node.
value	list	Yes	The confusion matrix.
round	integer	Yes	The current round.
type	string	Yes	The type of the confusion matrix.
class_labels	list	No	The labels for the classes.

Responses

- 200 OK: Returns a confirmation message.
- 400 Bad Request: If required parameters are missing or if no scenario is running.

POST /logging

Adds a log entry.

Request Body

Parameter	Type	Required	Description
uid	string	Yes	The UID of the node.
msg	string	Yes	The log message.
level	string	Yes	The log level.
exc	string	No	Exception information.
scenario_id	string	No	The ID of the scenario.

Responses

- 200 OK: Returns the ID of the created log entry.
- 400 Bad Request: If required parameters are missing.

WEBSOCKET /ws/logs

WebSocket for streaming logs.

Query Parameters

Parameter	Type	Required	Description
sid	string	Yes	The ID of the scenario to stream logs for.

WEBSOCKET /ws/monitor

WebSocket for monitoring the currently running scenario.

WEBSOCKET /ws/orchestrator

WebSocket for the orchestrator to communicate with nodes.

Query Parameters

Parameter	Type	Required	Description
uid	string	Yes	The UID of the connecting node.

POST /join_scenario

Allows a node to join a scenario.

Request Body

Parameter	Type	Required	Description
uid	string	Yes	The UID of the node.
ip	string	Yes	The IP address of the node.

Responses

- 200 OK: Returns scenario details for the joining node.
- 400 Bad Request: If required parameters are missing, the UID has already joined, or the maximum number of nodes has been reached.
- 404 Not Found: If the scenario is not found.

POST /report_status*Receives the status of a node.***Request Body**

Parameter	Type	Required	Description
uid	string	Yes	The UID of the node.
status	string	Yes	The new status of the node.
round	integer	Yes	The current round.

Responses

- 200 OK: Returns a confirmation message.
- 400 Bad Request: If required parameters are missing or if no scenario is running.

POST /report_statistics*Receives a metric from a node.***Request Body**

Parameter	Type	Required	Description
uid	string	Yes	The UID of the node.
type	string	Yes	The type of the statistic.
value	float	Yes	The value of the statistic.
round	integer	Yes	The current round.

Responses

- 200 OK: Returns a confirmation message.
- 400 Bad Request: If required parameters are missing or if no scenario is running.

WEBSOCKET /ws/statistics*WebSocket for streaming statistics.***Query Parameters**

Parameter	Type	Required	Description
sid	string	Yes	The ID of the scenario to stream statistics for.

5.2.5 Data Persistence

The MongoDB database has two main purposes:

- Storing experimental data across scenarios.
- Sharing data with participant nodes for the currently running scenario.

The database is organized into several collections based on the data's purpose:

- **scenarios**: Stores definitions and configurations of the scenarios, including ML hyperparameters, node configurations and other metadata required to run a scenario.
- **system**: Stores the UID of the currently running scenario.
- **statistics**: Stores metrics generated during scenario execution.
- **logs**: Stores all logs generated by the participant nodes during scenario execution.
- **confusion_matrices**: Stores confusion matrices from each participant node for each training round, both from before and after aggregation. For the confusion matrices, the underlying data as well as the rendered image encoded in Base64 is stored.

Docker Compose

To set up the MongoDB database, it can either be installed manually or using the `docker-compose-db-only.yml`. It is also possible to launch the MongoDB database together with the Coordinator.

```

1 services:
2   db-only-mongodb:
3     image: mongodb/mongodb-community-server:7.0-ubi8
4     container_name: mongo
5     command: ["--replSet", "rs0", "--bind_ip_all", "--port", "27017"]
6     ports:
7       - "27017:27017"
8     extra_hosts:
9       - "host.docker.internal:host-gateway"
10    healthcheck:
11      test: ["CMD", "mongosh", "--quiet", "--eval", "db.adminCommand('
ping')"]
12      interval: 5s
13      timeout: 30s
14      retries: 6
15    volumes:
16      - db-only-mongo_data:/data/db
17
18  db-only-mongo-init-replica:
19    image: mongodb/mongodb-community-server:7.0-ubi8
20    depends_on:

```

```

21     db-only-mongodb:
22         condition: service_healthy
23     entrypoint: >
24         bash -c '
25             echo "Initialising single-node replica-set rs0 ";
26             mongosh "db-only-mongodb:27017" --eval "
27                 rs.initiate({
28                     _id: \"rs0\",
29                     members: [{ _id: 0, host: \"db-only-mongodb:27017\" }]
30                 });
31             ";
32             echo "Replica-set rs0 initialised.";
33         '
34     restart: "no"
35
36 volumes:
37     db-only-mongo_data:

```

Listing 5.2: Docker Compose file for instantiating MongoDB replica set

The Docker Compose configuration shown in 5.2 defines two services - `db-only-mongodb` and `db-only-mongo-init-replica` - plus a named volume to set up a MongoDB single-node replica set.

db-only-mongodb:

1. Runs MongoDB with a replica set identifier (`rs0`), using the official MongoDB Community Server image.
2. Binds to all network interfaces, and listens on port 27017 after exposing it on the host for external connections.
3. Adds an extra host entry (`host.docker.internal`) for accessing services running on the Docker host.
4. Periodically pings the server to ensure that the MongoDB instance behaves correctly.
5. Raw data of MongoDB is stored in the named volume `db-only-mongo_data`.

db-only-mongo-init-replica:

1. Also runs MongoDB with the official MongoDB Community Server image.
2. Waits until `db-only-mongodb` reports healthy before starting.
3. Runs a one-time entrypoint script that initializes the `rs0` replica set with the single member `db-only-mongodb:27017`.
4. Terminates and does not restart, as its job is a one-off initialization.

db-only-mongo_data volume:

- To ensure data persistence across container restarts, a named Docker volume is used to store MongoDB's data files.

With this setup, a MongoDB database is configured for storing data. Since it is configured as a replica set, the 'Change Streams' feature can also be used to update the frontend with incoming data in real-time.

5.2.6 Frontend

The platform is operated through a web-based user interface, which allows the user to:

- Configure, create and delete scenarios.
- View past scenarios and their:
 - Training metrics (stats)
 - Confusion matrices
 - Scenario logs
- Monitor the currently running scenario.

The frontend is implemented using the Jinja template engine and is made up of the following components:

- `pages.py`: Defines the routes for the web interface.
- `templates/`: Contains the HTML templates for the web pages.
- `static/`: Contains the static assets, such as CSS stylesheets, JavaScript files, and images.

Scenario Configuration

Figure 5.2 shows the configuration window for setting up a new scenario. It allows the user to:

- Add a name and description for the scenario.
- Configure the number of nodes and their topology.
- Choose the number of malicious nodes, their attack type and parameters relevant for the attack.
- Configure training hyperparameters and select the dataset.

Configurator
Scenarios

Scenario Configurator

Scenario

Name

Description

Topology

Nodes

Type

Attack

Type

Number of Malicious Nodes

Training

Rounds

Dataset

Trainer

Epochs

Batch Size

Learning Rate

Device & Aggregator

Device

Aggregator

Deployment

Deployment Method

Store Configuration

Figure 5.2: Scenario configuration window

- Select the device type and aggregation mechanism.
- Choose the deployment method.

Clicking on the "Store Configuration" button saves the selected configuration, after which it can be launched.

Scenario Launch Overview

After storing the scenario configuration, it can be launched from the launch window depicted in Figure 5.3 by pressing the "Launch Scenario" button. The scenario can also be exported in JSON format by pressing the "Export Scenario" button.

Live Scenario Dashboard

Launching the scenario opens the live dashboard seen in Figure 5.4. The dashboard provides an overview of all the relevant data pertaining to the current scenario. This includes:

- The JSON configuration data for the scenario.

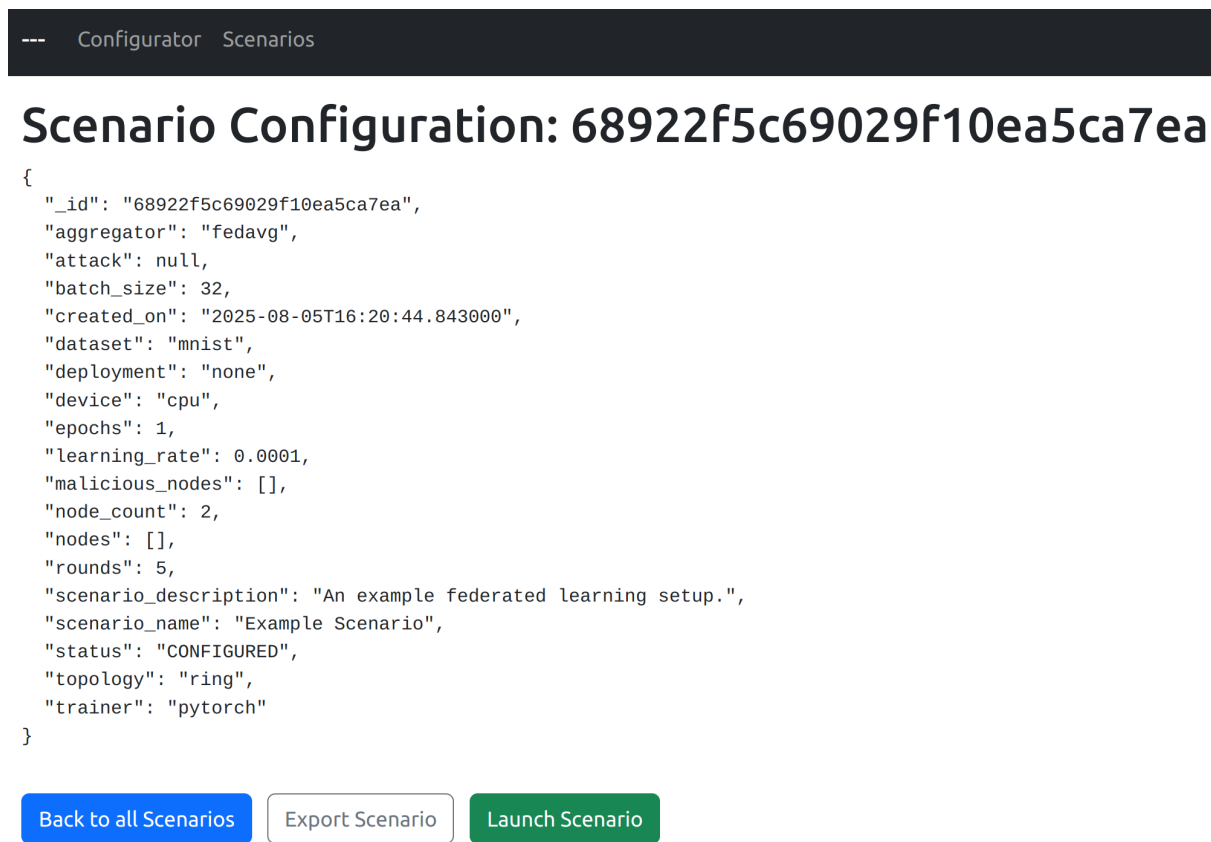


Figure 5.3: Scenario launch overview window

--- Configurator Scenarios

Scenario Monitor – 6891ce5769029f10ea5ca663

WebSocket: Live

Status: FINISHED

Last updated: 6 h ago

```

{
  "_id": "6891ce5769029f10ea5ca663",
  "scenario_name": "Example Scenario",
  "scenario_description": "An example federated learning setup.",
  "rounds": 5,
  "dataset": "mnist",
  "trainer": "pytorch",
  "learning_rate": 0.0001,
  "batch_size": 32,
  "epochs": 1,
  "node_count": 2,
  "topology": "ring",
  "device": "cpu",
  "aggregator": "fedavg",
  "attack": null,
  "nodes": [
    {
      "UID": "65de71ad-da96-4a25-b599-592a2bf79e32",
      "IP": "127.0.0.1",
      "friendly_name": "node_0",
      "port": 45000,
      "dataset": "mnist",
      "data_split": 0,
      "data_total_splits": 2,
      "status": "aggregation_done_4",
      "training_done_round": 4,
      "gossip_done_round": 4,
      "aggregation_done_round": 4
    },
    {
      "UID": "127.0.0.1",
      "IP": "127.0.0.1",
      "friendly_name": "node_1",
      "port": 45001,
      "dataset": "mnist",
      "data_split": 0,
      "data_total_splits": 2,
      "status": "aggregation_done_4",
      "training_done_round": 4,
      "gossip_done_round": 4,
      "aggregation_done_round": 4
    }
  ]
}

```

Node Status

Node	Status
node_0	aggregation_done_4
node_1	aggregation_done_4

Live Logs

```

9:27:01 AM INFO *9e32: [MNISTDataset] Loading data from flwr-datasets (MNIST)...
9:27:01 AM STATUS *45ff: All nodes connected
9:27:01 AM STATUS *9e32: All nodes connected
9:27:01 AM INFO *45ff: Received scenario config: {'detail': 'Joined Successfully!', 'scenario_id': '6891ce5769029f10ea5ca663', 'uid': '0ae8a70f-8a8e-4105-9936-e66e845f45ff', 'port': 45001, 'dataset': 'mnist', 'data_split': 1, 'data_total_splits': 2, 'epochs': 1, 'learning_rate': 0.0001, 'batch_size': 32, 'wandb_entity': 'tnaescher-ma-uzh', 'wandb_name': 'node_1_0ae8a70f-8a8e-4105-9936-e66e845f45ff', 'wandb_project': '6891ce5769029f10ea5ca663', 'wandb_api_key': '8efb6ce8bba52f51ba39943fc7ba35fe2311d45a', 'attack': None}
9:27:01 AM STATUS *45ff: Joined scenario
9:27:01 AM INFO *9e32: Received scenario config: {'detail': 'Joined Successfully!', 'scenario_id': '6891ce5769029f10ea5ca663', 'uid': '0ae8a70f-8a8e-4105-9936-e66e845f45ff', 'port': 45001, 'dataset': 'mnist', 'data_split': 0, 'data_total_splits': 2, 'epochs': 1, 'learning_rate': 0.0001, 'batch_size': 32, 'wandb_entity': 'tnaescher-ma-uzh', 'wandb_name': 'node_0_0ae8a70f-8a8e-4105-9936-e66e845f45ff', 'wandb_project': '6891ce5769029f10ea5ca663', 'wandb_api_key': '8efb6ce8bba52f51ba39943fc7ba35fe2311d45a', 'attack': None}

```

Statistics

Open W&B Dashboard
Export CSV
Confusion Matrices

ACCURACY (AFTER AGGREGATION)

Round	node0 *45ff	node1 *9e32
0	82.2400	82.2400
1	90.9100	90.9100
2	91.8200	91.8200
3	92.5300	92.5300
4	93.1700	93.1700

Figure 5.4: Live scenario dashboard window

- The current status of each participant node.
- A window with timestamped logs from each node that is updated in real time.
- Buttons for opening the current experiment in WandB, exporting the training results to .csv format, and viewing each participant node's confusion matrices for each training round.
- Each participant node's training metrics, updated after every round.

Scenario History

--- Configurator Scenarios

All Scenarios

Import scenario (JSON)

Choose File
No file chosen
Import

Delete All Scenarios

ID	Scenario Name	Description	Status	Rounds	Nodes	Created On	Actions
6891ce5769029f10ea5ca663	Example Scenario	An example federated learning setup.	Finished	5	2	2025-08-05 09:26	Logs Confusion Matrices Stats Export Delete
6891cdf569029f10ea5ca659	Example Scenario	An example federated learning setup.	Stopped	5	2	2025-08-05 09:25	Logs Confusion Matrices Stats Export Delete

Figure 5.5: Scenario history window

Past scenarios can be viewed by pressing on the "Scenarios" button in the header. This opens the view visible in Figure 5.5. The history window provides information on past scenarios and buttons to open up more detailed views on run statistics.

Scenario Logs

--- Configurator Scenarios

Scenario Logs

ID: 6891ce5769029f10ea5ca663

Example Scenario – An example federated learning setup.

node_1 (*45ff) STATUS Filter

Timestamp	Level	Node	Message
2025-08-05 09:29:32	STATUS	node_1 - *45ff	Scenario FINISHED
2025-08-05 09:29:32	STATUS	node_1 - *45ff	AGGREGATION_DONE r4
2025-08-05 09:29:32	STATUS	node_1 - *45ff	WAITING_FOR_AGGREGATION_DONE r4
2025-08-05 09:29:28	STATUS	node_1 - *45ff	GOSSIP_DONE r4
2025-08-05 09:29:26	STATUS	node_1 - *45ff	TRAINING_DONE r4
2025-08-05 09:29:26	STATUS	node_1 - *45ff	WAITING_FOR_TRAINING_DONE r4
2025-08-05 09:29:06	STATUS	node_1 - *45ff	AGGREGATION_DONE r3
2025-08-05 09:29:06	STATUS	node_1 - *45ff	WAITING_FOR_AGGREGATION_DONE r3
2025-08-05 09:29:01	STATUS	node_1 - *45ff	GOSSIP_DONE r3
2025-08-05 09:28:59	STATUS	node_1 - *45ff	TRAINING_DONE r3
2025-08-05 09:28:59	STATUS	node_1 - *45ff	WAITING_FOR_TRAINING_DONE r3
2025-08-05 09:28:36	STATUS	node_1 - *45ff	AGGREGATION_DONE r2
2025-08-05 09:28:36	STATUS	node_1 - *45ff	WAITING_FOR_AGGREGATION_DONE r2
2025-08-05 09:28:31	STATUS	node_1 - *45ff	GOSSIP_DONE r2
2025-08-05 09:28:29	STATUS	node_1 - *45ff	TRAINING_DONE r2
2025-08-05 09:28:29	STATUS	node_1 - *45ff	WAITING_FOR_TRAINING_DONE r2
2025-08-05 09:28:09	STATUS	node_1 - *45ff	AGGREGATION_DONE r1
2025-08-05 09:28:09	STATUS	node_1 - *45ff	WAITING_FOR_AGGREGATION_DONE r1
2025-08-05 09:28:04	STATUS	node_1 - *45ff	GOSSIP_DONE r1

Figure 5.6: Scenario logs window

Pressing on the "Logs" button in the history window opens up the window visible in Figure 5.6. All the logs generated by participant nodes during the FL process are displayed here. Each log message also includes a timestamp, log level and information on the node that output the message. Messages may also be filtered for a specific node and log level.

Scenario Training Metrics

Pressing on the "Stats" button in the history window opens up the window visible in Figure 5.7. Training metrics for participating nodes are displayed in this window. Currently, both

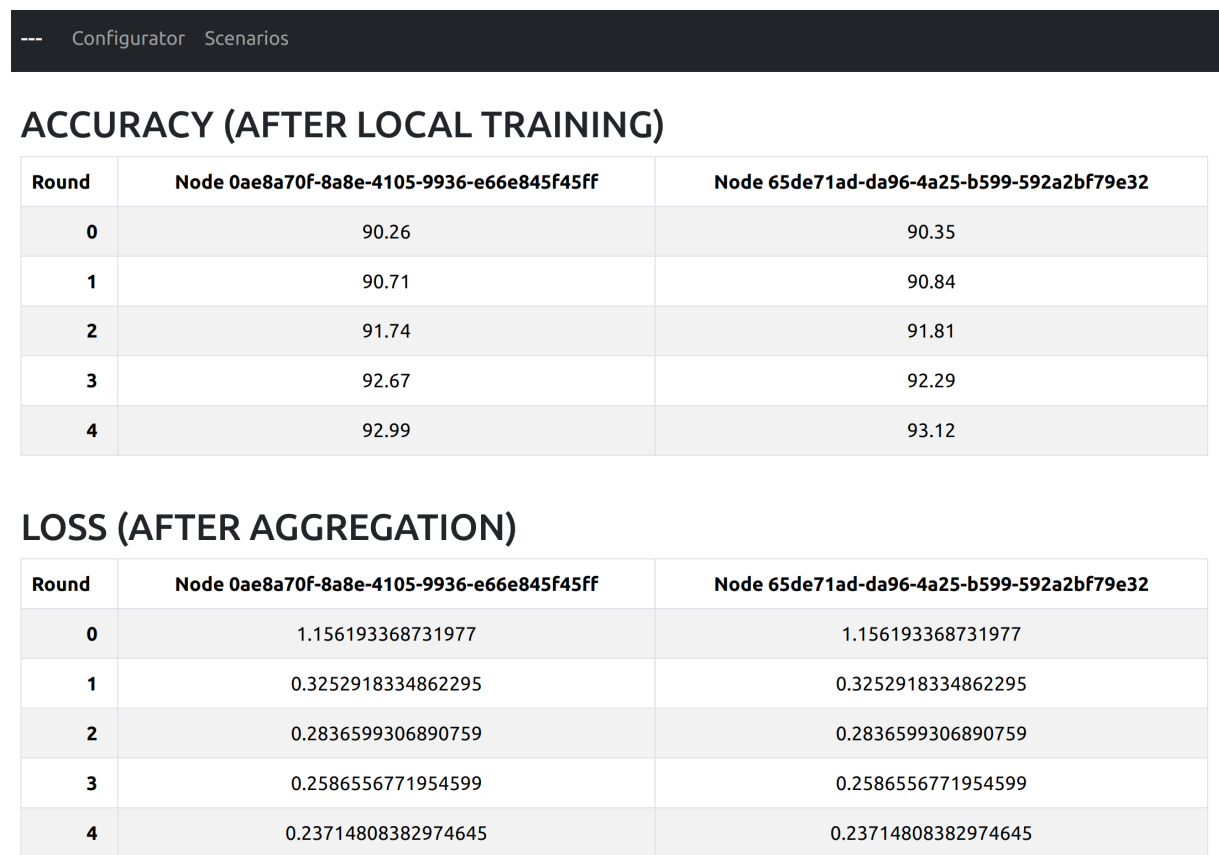


Figure 5.7: Scenario stats window

the accuracy and loss of each participant node on their respective test data is displayed. Metrics are displayed for all participant nodes each round, immediately following their local training and also after aggregation.

Scenario Confusion Matrices

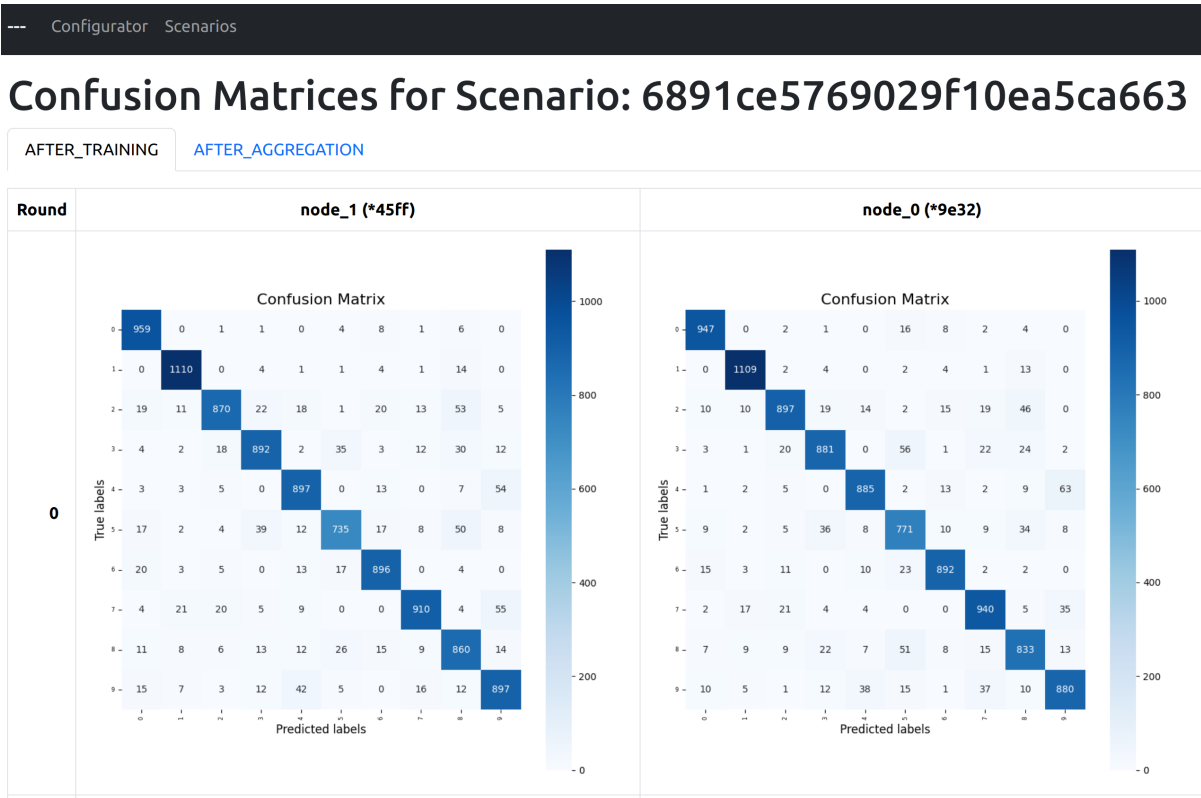


Figure 5.8: Scenario confusion matrices window

Pressing on the "Confusion Matrices" button in the history window opens up the window in Figure 5.8. This window displays confusion matrices for all participant nodes after each training round. Similarly to the "Stats" window, confusion matrices are also generated twice, once after local training and also after aggregation.

5.2.7 Deployment

5.3 Participant

Participant nodes are the second main component of the framework. Each participant node trains a local model on its own data and shares it with neighboring nodes after each training round. The participant nodes' training process is orchestrated by the central Coordinator.

5.3.1 Structure

The following tree lists the structure of the Participant codebase:

```

.
├── .dockerignore
├── .gitignore
├── docker-compose.yml
├── Dockerfile
├── README.md
├── requirements.txt
├── app/
│   ├── async_tools.py
│   ├── config.py
│   ├── ma_logging.py
│   ├── main.py
│   ├── state.py
│   ├── aggregators/
│   ├── api/
│   ├── attacks/
│   ├── event_handlers/
│   ├── ma_datasets/
│   ├── services/
│   └── trainers/

```

The main components of the participant nodes are in the **app/** directory and are structured in the following way:

- **aggregators/**: Contains the aggregators available for the FL training process.
- **api/**: Contains the REST API endpoint to allow communication with other participant nodes.
- **attacks/**: Implements various adversarial attacks.
- **event_handlers/**: Contains the logic for the state machine and manages the behavior of participant nodes upon receiving events from the Coordinator. Each file corresponds to a specific state in the state machine, and defines the actions the participant should take in that state.
- **ma_datasets/**: Manages the datasets used for training.
- **services/**: Provides services for various tasks, such as joining a scenario, reporting statistics, and handling the gossip protocol.
- **trainers/**: Contains the logic for training the machine learning model.
- **wandb/**: Contains WandB data for past scenarios.

The `app/` root contains additional files for keeping track of data relevant for the scenario (`state.py`), additional configuration parameters like timeout intervals (`config.py`) or logging (`ma_logging.py`).

Many of the framework's components are abstracted in a way to allow easy modifications and extensions, as such there are base classes for aggregation (`BaseAggregator`), datasets (`BaseDataset`) and model trainers (`BaseTrainer`), with more specific classes implementing the functionality of the corresponding base class.

5.3.2 State Machine

As previously mentioned, the participant nodes use the same states as the Coordinator's state machine shown in Figure 5.1. Only the behavior of the participant nodes in a given state differs from the Coordinator's. While the Coordinator ensures a coherent training process and orchestrates the training process, the participant nodes perform the federated learning process and inform the Coordinator of their progress.

1. **WAITING_FOR_NODES**: Upon launching the participant node, it attempts to join the currently running scenario. The participant node remains in this state until the Coordinator announces that all nodes have joined the scenario. While waiting, it receives updates on the number of connected nodes.
2. **ALL_NODES_CONNECTED**: Once all participant nodes are connected to the Coordinator, participant nodes receive a list of all other participants from the Coordinator.
3. **START_TRAINING**: After receiving data on their neighboring nodes, participants begin the local training process. This includes:
 - Loading a partition of the dataset. If the node is malicious, it also manipulates the dataset depending on the attack.
 - Initializing the trainer (first round only).
 - Training the model on the local data.
 - Evaluating the model and reporting training metrics to WandB and the Coordinator.
4. **WAITING_FOR_TRAINING_DONE**: Participant nodes that finish wait for all other nodes to finish their training.
5. **TRAINING_DONE**: This state marks the end of the training round for all participants. The participant transitions to the gossip phase.
6. **START_GOSSIP**: After the training phase, participants begin the gossip phase. Participant nodes send their model parameters to the neighbors they learned about in the **ALL_NODES_CONNECTED** state.
7. **WAITING_FOR_GOSSIP_DONE**: Participants that finish early again wait for the other participant nodes to finish the gossip phase.

8. **GOSSIP_DONE**: Once all participants finish the gossip phase and receive model parameters from neighboring nodes, they transition to the aggregation phase.
9. **START_AGGREGATION**: Participants aggregate its own model parameters with the model parameters received from its neighbors using the selected aggregation method (e.g., FedAvg). The model is evaluated once more and metrics are reported to the Coordinator.
10. **WAITING_FOR_AGGREGATION_DONE**: Participant nodes wait for all other nodes to finish the aggregation phase.
11. **AGGREGATION_DONE**: This state marks the end of the aggregation phase and the current federation round. If there are more rounds to go, participants transition back to the **START_TRAINING** state. Otherwise, it transitions to the **SCENARIO_FINISHED**.
12. **SCENARIO_FINISHED**: This is the final state of the state machine and concludes the scenario.

5.3.3 Federated Learning

The federated learning pipeline is an integral component of the developed framework. In order to facilitate development for a wide range of applications, the components that make up the FL process (dataset, training, aggregation) are abstracted in a way to allow for simple extensions.

Data / Datasets

For the data handling process, the Flower Datasets¹ (`flwr-datasets`) library is used. It was created by the same authors that also published the **Flower** framework² [14]. It enables quick and easy creation of datasets for federated learning applications. It enables heterogeneity (non-iidness) simulation and division of datasets with the preexisting notion of IDs. The library was created by the Flower Labs team that also created the Flower framework.

Datasets inherit from a `BaseDataset` class and implement the `get_train_data()`, `get_test_data()` and `load_data()` functions. Code snippet 5.3 shows the `BaseDataset` class.

```

1 class BaseDataset:
2     def __init__(self, partition_index=None, partitioners=None, split="
    train", test_split="test"):
3         self.partition_index = partition_index
4         self.partitioners = partitioners if partitioners else {"train":
    1}
5         self.split = split
6         self.test_split = test_split
7

```

¹<https://flower.ai/docs/datasets/>

²<https://flower.ai/>

```

8     def get_train_data(self):
9         raise NotImplementedError("[BaseDataset] get_train_data() not
implemented.")
10
11    def get_test_data(self):
12        raise NotImplementedError("[BaseDataset] get_test_data() not
implemented.")
13
14    def load_data(self):
15        raise NotImplementedError("[BaseDataset] load_data() not
implemented.")

```

Listing 5.3: BaseDataset class

```

1  from flwr_datasets import FederatedDataset
2
3  class MNISTDataset(BaseDataset):
4      def __init__(
5          self, partition_index=None, partitioners=None, split="train",
6          test_split="test"):
7          super().__init__(partition_index=partition_index, partitioners=
8          partitioners, split=split, test_split=test_split)
9          self.train_dataset = None
10         self.test_dataset = None
11         self.load_data()
12
13     def get_train_data(self):
14         return self.train_dataset
15
16     def get_test_data(self):
17         return self.test_dataset
18
19     def load_data(self):
20         print("[MNISTDataset] Loading data from flwr-datasets (MNIST)...
21         ")
22         fds = FederatedDataset(dataset="mnist", partitioners=self.
23         partitioners)
24         if self.partition_index is not None:
25             self.train_dataset = fds.load_partition(self.partition_index
26             , self.split)
27         else:
28             self.train_dataset = fds.load_split(self.split)
29             self.test_dataset = fds.load_split(self.test_split)

```

Listing 5.4: MNISTDataset class

A concrete implementation of the `BaseDataset` class is shown in code snippet 5.4, where the MNIST dataset is implemented. To ensure that no two nodes use the same data, the dataset is first divided into as many partitions as there are participant nodes. The `flwr-dataset` library provides the `FederatedDataset` class to handle the download and partitioning of datasets for FL.

Training

The training process is implemented in the `app/trainers` directory. Similarly to the datasets, an abstract base class `BaseTrainer` is again defined, which ensures that all trainers have a consistent interface. The abstract `BaseTrainer` class can be seen in code snippet 5.5.

```

1 class BaseTrainer:
2     def __init__(self, dataset, epochs=1, batch_size=32, learning_rate=1
  e-3, device="cpu"):
3         self.dataset = dataset
4         self.epochs = epochs
5         self.batch_size = batch_size
6         self.learning_rate = learning_rate
7         self.device = device
8         self.model = None
9
10    def train(self):
11        print("BaseTrainer train() called - no operation.")
12
13    def evaluate(self):
14        print("BaseTrainer evaluate() called - no operation.")
15        return None
16
17    def get_model_params(self):
18        print("BaseTrainer get_model_params() called - returning None.")
19        return None
20
21    def set_model_params(self, params):
22        print("BaseTrainer set_model_params() called - no operation.")

```

Listing 5.5: BaseTrainer class

```

1 class PyTorchTrainer(BaseTrainer):
2     def __init__(self, dataset, epochs, batch_size, learning_rate,
  device="cpu"):
3         super().__init__(dataset, epochs, batch_size, learning_rate,
  device)
4         self.model = MNISTModelMLP().to(self.device)
5         self.criterion = nn.CrossEntropyLoss()
6         self.optimizer = optim.Adam(self.model.parameters(), lr=self.
  learning_rate)
7         wandb.login(key=state.wandb_api_key, relogin=True)
8         self._wandb_run = wandb.init(...),
9         )
10
11    def train(self):
12        train_ds = self.dataset.get_train_data()
13        train_ds = train_ds.with_transform(apply_transforms)
14        train_loader = DataLoader(train_ds, batch_size=self.batch_size,
  shuffle=True)
15
16        self.model.train()
17        for epoch in range(self.epochs):
18            ...
19

```

```

20     def evaluate(self, rndplus10=None):
21         test_ds = self.dataset.get_test_data()
22         test_ds = test_ds.with_transform(apply_transforms)
23         test_loader = DataLoader(test_ds, batch_size=self.batch_size,
shuffle=False)
24
25         with torch.no_grad():
26             for batch in test_loader:
27                 ...
28
29         accuracy = 100.0 * correct / total
30         avg_loss = running_loss / len(test_loader)
31         return accuracy, avg_loss
32
33     def get_model_params(self):
34         return self.model.state_dict()
35
36     def set_model_params(self, params):
37         self.model.load_state_dict(params)
38         self.optimizer = optim.Adam(self.model.parameters(), lr=self.
learning_rate)
39
40     def serialize(self, model):
41         ...
42
43     def deserialize(self, serialized_data):
44         ...

```

Listing 5.6: PyTorchTrainer class

Code snippet 5.6 shows an abridged version of the concrete trainer class implementation `PyTorchTrainer`, which uses the PyTorch library to train the model. The trainer is responsible for all aspects of the training process, including loading the data, defining the model, and performing the training loop. It also calculates and reports metrics such as loss and accuracy.

- The `__init__()` method initializes the core components for training. This includes the neural network (`MNISTModelMLP`), loss function (`nn.CrossEntropyLoss`) and optimization algorithm (`optim.Adam`). A connection to the WandB platform is also initialized to keep track of metrics and the training process.
- The `train()` function implements the training loop. It initializes the data loader from the selected dataset and applies necessary transformations (e.g., converting images to tensors and normalizing them). The training loop itself implements the standard PyTorch training steps of performing a forward pass, calculating the loss, backpropagating the compute gradients and finally updating the model parameters with the optimizer.
- The `evaluate` method works in a similar fashion but instead operates on the test dataset. It calculates the accuracy, loss and also plots the confusion matrix for the current round.

- The `get_model_params()` and `set_model_params()` methods are instrumental for the FL process. To share local model parameters with neighboring nodes, the `get_model_params()` is used to extract the current parameters. After the participant node receives model parameters from neighboring nodes, the `set_model_params()` is used to update the local model parameters with the aggregated result.
- Finally, while not shown, the `serialize()` and `deserialize()` functions are used for communication with neighboring nodes during the gossip phase. Model parameters are prepared for network transmission with the `serialize()` method, which converts them into a byte stream. Conversely, the `deserialize()` method is used to convert the byte streams received from neighboring nodes back to model parameters for aggregation.

Aggregation

The final core component that makes up a federated learning scenario is the aggregation method. An abstract base class `BaseAggregator` is used to define the function signatures that need to be implemented by concrete aggregator classes. The `BaseAggregator` class is shown in code snippet 5.7.

```

1 class BaseAggregator:
2     def __init__(self):
3         pass
4
5     def aggregate_params(self, list_of_params):
6         print("[BaseAggregator] aggregate_params() called but not
7         implemented.")
8         return None

```

Listing 5.7: BaseAggregator class

The `aggregate_params` method is designed to take a list of model parameters from multiple participant nodes and aggregate them into a single set of updated model parameters. A concrete implementation of this class is the `FedAvgAggregator` shown in code snippet 5.8, which implements the commonly used Federated Averaging (FedAvg) algorithm.

```

1 import copy
2 import torch
3 from .base_aggregator import BaseAggregator
4
5 class FedAvgAggregator(BaseAggregator):
6
7     def aggregate_params(self, list_of_params):
8         if not list_of_params:
9             print("[FedAvgAggregator] No params to aggregate, returning
10             None.")
11             return None
12
13         aggregated_params = copy.deepcopy(list_of_params[0])
14
15         num_models = len(list_of_params)
16         for idx in range(1, num_models):

```

```

16         current_params = list_of_params[idx]
17         for key in aggregated_params.keys():
18             if isinstance(aggregated_params[key], torch.Tensor):
19                 aggregated_params[key] += current_params[key]
20             else:
21                 aggregated_params[key] += current_params[key]
22         for key in aggregated_params.keys():
23             if isinstance(aggregated_params[key], torch.Tensor):
24                 aggregated_params[key] = aggregated_params[key] /
num_models
25             else:
26                 aggregated_params[key] = aggregated_params[key] / float(
num_models)
27         return aggregated_params

```

Listing 5.8: FedAvgAggregator class

The implemented `aggregate_params()` method receives a list of model parameters and simply averages the parameters from each model. The aggregation process consists of creating a deep copy of the first participant’s parameters and iteratively summing up all subsequent participants’ parameters element-wise, before dividing each parameter by the number of participants. The computed arithmetic mean represents the collective knowledge of all participating clients.

Poisoning Attacks

To investigate the robustness of the federated learning process, several poisoning attacks were implemented. These attacks are designed to disrupt the training process and degrade the performance of the global model. The implemented attacks can be categorized into two main types: data poisoning attacks and model poisoning attacks.

Data Poisoning Attacks

Data poisoning attacks aim to manipulate the training data of the participants to compromise the learning process. The following data poisoning attacks were implemented:

- **Fang Label Flipping:** This attack, as described in [41], essentially inverts the order of the data’s labels. More specifically, given L classes in total, a zero-indexed label l is flipped to $L - l - 1$. The implementation is shown in code snippet 5.9.
- **Targeted Label Flipping:** This is a targeted version of the label flipping attack. Instead of flipping all labels, this attack specifically targets a single class and changes its labels to another target class. The implementation is shown in code snippet 5.10
- **Untargeted Label Flipping:** Instead of flipping one specific class to another specific class, the untargeted attack indiscriminately flips labels to a random class. This provides a more generalized approach to label flipping.

```

1 def label_flipping_fang(dataset):

```

```

2 if hasattr(dataset, "class_to_idx") and dataset.class_to_idx:
3     num_classes = len(dataset.class_to_idx)
4 else:
5     num_classes = 10
6 def flip_labels_batch(batch):
7     labels = np.array(batch["label"])
8     for i in range(len(labels)):
9         original_label = labels[i]
10        flipped_label = num_classes - original_label - 1
11        labels[i] = flipped_label
12    batch["label"] = labels
13    return batch
14
15 dataset.train_dataset = dataset.train_dataset.map(flip_labels_batch,
16                                                    batched=True, batch_size=256)
17 return dataset

```

Listing 5.9: Fang label flipping attack

```

1
2 def label_flipping_targeted_specific(dataset, label_og: int, label_goal:
   int):
3
4     def flip_labels(batch):
5         labels = np.array(batch["label"])
6         for i in range(len(labels)):
7             if labels[i] == label_og:
8                 labels[i] = label_goal
9         batch["label"] = labels
10        return batch
11
12    dataset.train_dataset = dataset.train_dataset.map(flip_labels,
13    batched=True, batch_size=256)
14    return dataset

```

Listing 5.10: Targeted label flipping attack

Model Poisoning Attacks

Model poisoning attacks, on the other hand, directly manipulate the model updates before they are sent to the aggregator. A demo update manipulation attack was implemented and is shown in code snippet 5.11. The attack replaces weights with large values to degrade the aggregated model performance of neighboring nodes.

```

1 def update_manipulation_demo(parameters):
2     malicious_parameters = {}
3     for key, value in parameters.items():
4         if key.endswith("bias"):
5             malicious_parameters[key] = value
6         else:
7             new_weights_list = []
8             for weights in value:
9                 new_weights = []
10                for _ in weights:
11                    new_weights.append(99)
12                new_weights_list.append(new_weights)

```

```
13     malicious_parameters[key] = torch.tensor(new_weights_list)
14     return malicious_parameters
```

Listing 5.11: Update manipulation demo attack

5.3.4 Deployment

The participant nodes are designed to be deployed as multiple separate instances of the same source file `app/main.py`. Once the Coordinator has launched and a scenario is ran, one instance is launched for each participant node. The participant nodes individually connect to the Coordinator over a WebSocket and receive their role in the scenario (e.g, benign or malicious node). A debug process may also be attached to individual nodes to facilitate project development.

Alternatively, the repository also includes a `Dockerfile` that can be used to build a Docker image for the scenario. After configuring the scenario in the web interface, a `docker-compose.yml` file can be exported and deployed. This setup simplifies the process of scaling up the federated learning environment to a large number of participants.

Chapter 6

Evaluation

In this chapter the performance and correctness of the developed federated learning framework is evaluated. First, the correctness of the various topologies and poisoning attacks is verified. This is followed by an investigation on the scalability of the platform. Finally, since the platform should be accessible for a wide range of users, a user study is also performed to gauge the usability of the framework.

6.1 Network Topologies

A number of different topologies were implemented for this framework. To verify the correctness of the implemented topologies, unit tests were implemented for each topology. The unit tests compare the output produced by the `build_neighbour_map()` method against its expected output. Each topology is tested with 0, 1, 2 and 5 nodes. Code snippet 6.1 shows the implemented unit tests. Some of the test cases were omitted for brevity. In each case, the input consists of a dictionary with a `UID` key and a separate `UID` for each participant node. This input is passed to the `build_neighbour_map()` method, together with the topology type and returns a deterministic list of neighbors for each participant node. Figure 6.1 shows the output from each unit tests, clearly showing that the topologies were correctly implemented. Moreover, the unit tests achieve 100% code coverage.

```
1 class TestTopologyBuilder(unittest.TestCase):
2
3     def test_fully_connected_n_0(self):
4         nodes = []
5         with self.assertRaises(ValueError):
6             build_neighbour_map(nodes, "fullyconnected")
7
8     def test_fully_connected_n_1(self):
9         nodes = [{"UID": "1"}]
10        expected = {"1": []}
11        self.assertEqual(build_neighbour_map(nodes, "fullyconnected"),
12                           expected)
```

test_topology		4 ms
■ TestTopologyBuilder		4 ms
test_duplicate_uids	passed	1 ms
test_fully_connected_n_0	passed	0 ms
test_fully_connected_n_1	passed	0 ms
test_fully_connected_n_2	passed	0 ms
test_fully_connected_n_5	passed	0 ms
test_missing_uid	passed	0 ms
test_random_n_0	passed	0 ms
test_random_n_1	passed	0 ms
test_random_n_2	passed	2 ms
test_random_n_5	passed	0 ms
test_ring_n_0	passed	0 ms
test_ring_n_1	passed	0 ms
test_ring_n_2	passed	0 ms
test_ring_n_5	passed	0 ms
test_star_n_0	passed	0 ms
test_star_n_1	passed	0 ms
test_star_n_2	passed	0 ms
test_star_n_5	passed	0 ms
test_unknown_topology	passed	1 ms

Figure 6.1: Tests

```

13 def test_fully_connected_n_2(self):
14     nodes = [{"UID": "1"}, {"UID": "2"}]
15     expected = {
16         "1": [{"UID": "2"}],
17         "2": [{"UID": "1"}],
18     }
19     self.assertEqual(build_neighbour_map(nodes, "fullyconnected"),
20                      expected)
21
22 def test_fully_connected_n_5(self):
23     ...
24
25 def test_ring_n_0(self):
26     ...
27
28 def test_ring_n_1(self):
29     ...
30
31 def test_ring_n_2(self):
32     nodes = [{"UID": "1"}, {"UID": "2"}]
33     expected = {
34         "1": [{"UID": "2"}],
35         "2": [{"UID": "1"}],
36     }
37     result = build_neighbour_map(nodes, "ring")
38     for uid in result:
39         result[uid] = sorted(result[uid], key=lambda x: x["UID"])
40     self.assertEqual(result, expected)
41
42 def test_ring_n_5(self):

```

```

42     ...
43
44     def test_star_n_0(self):
45         ...
46
47     def test_star_n_1(self):
48         ...
49
50     def test_star_n_2(self):
51         nodes = [{"UID": "1"}, {"UID": "2"}]
52         expected = {
53             "1": [{"UID": "2"}],
54             "2": [{"UID": "1"}],
55         }
56         self.assertEqual(build_neighbour_map(nodes, "star"), expected)
57
58     def test_star_n_5(self):
59         ...
60
61     def test_random_n_0(self):
62         ...
63
64     def test_random_n_1(self):
65         ...
66
67     @patch('random.sample')
68     def test_random_n_2(self, mock_sample):
69         ...
70
71     @patch('random.sample')
72     def test_random_n_5(self, mock_sample):
73         nodes = [{"UID": str(i)} for i in range(5)]
74         def mock_sample_func(pool, k):
75             return pool[:k]
76         mock_sample.side_effect = mock_sample_func
77
78         result = build_neighbour_map(nodes, "random")
79         for i in range(5):
80             self.assertEqual(len(result[str(i)]), 2)
81             self.assertNotIn({"UID": str(i)}, result[str(i)])
82
83     def test_unknown_topology(self):
84         nodes = [{"UID": str(i)} for i in range(5)]
85         with self.assertLogs('app.core.topology', level='ERROR') as cm:
86             result_unknown = build_neighbour_map(nodes, "unknown")
87             self.assertEqual(len(cm.output), 5)
88             for log_message in cm.output:
89                 self.assertIn("Unknown / Not Implemented topology
unknown. Falling back to fully-connected.", log_message)
90
91             result_fully_connected = build_neighbour_map(nodes, "
fullyconnected")
92
93             for uid in result_unknown:
94                 result_unknown[uid] = sorted(result_unknown[uid], key=lambda
x: x["UID"])

```

```

95         for uid in result_fully_connected:
96             result_fully_connected[uid] = sorted(result_fully_connected[
uid], key=lambda x: x["UID"])
97             self.assertEqual(result_unknown, result_fully_connected)
98
99         def test_missing_uid(self):
100             nodes = [{"UID": "1"}, {"foo": "bar"}]
101             with self.assertRaises(KeyError):
102                 build_neighbour_map(nodes, "fullyconnected")
103
104         def test_duplicate_uids(self):
105             nodes = [{"UID": "1"}, {"UID": "1"}]
106             with self.assertRaises(ValueError):
107                 build_neighbour_map(nodes, "fullyconnected")

```

Listing 6.1: Topology Builder Unit Tests

6.2 Data Manipulation Attacks

The efficacy and correctness of the data manipulation attacks was also verified on a qualitative basis. A simple model outlined in code snippet 6.2 was used for this purpose. It was kept small and simple to minimize the complexity and thus training speed, while still ensuring sufficient performance and complexity to correctly learn the data without malicious participants present. Verification consists of including malicious participants and viewing confusion matrices generated after each training round to check if the model exhibits the expected behavior during each attack. All attacks are evaluated with the MNIST dataset.

The model was kept as similar as possible to the implementation in Nebula/FedStellar to facilitate comparison with previous works [42], [43], apart from small differences due to the difference in training framework (PyTorch vs Lightning).

```

1 class SimplePyTorchModel(nn.Module):
2     """Dummy model for faster scenario runtime"""
3     def __init__(self, input_dim=784, hidden_dim=128, output_dim=10):
4         super(SimplePyTorchModel, self).__init__()
5         self.fc1 = nn.Linear(input_dim, hidden_dim)
6         self.relu = nn.ReLU()
7         self.fc2 = nn.Linear(hidden_dim, output_dim)
8
9     def forward(self, x):
10         x = x.view(x.size(0), -1)
11         x = self.fc1(x)
12         x = self.relu(x)
13         x = self.fc2(x)
14         return x

```

Listing 6.2: Dummy Model

It is important to note that the behavior of an attack heavily depends on the topology, as it may facilitate or hinder propagation respectively. Figure 6.2 shows an example of a

federation with five participants and one malicious node, with (a) illustrating the behavior for a ring topology and (b) for a fully connected topology. As the aggregation only occurs with neighboring participants, the ring topology causes the attack to propagate much slower, affecting all nodes after the third round. In contrast, the fully connected setting leads to the malicious participant sharing data with all participants in the first round, affecting their performance after just one round.

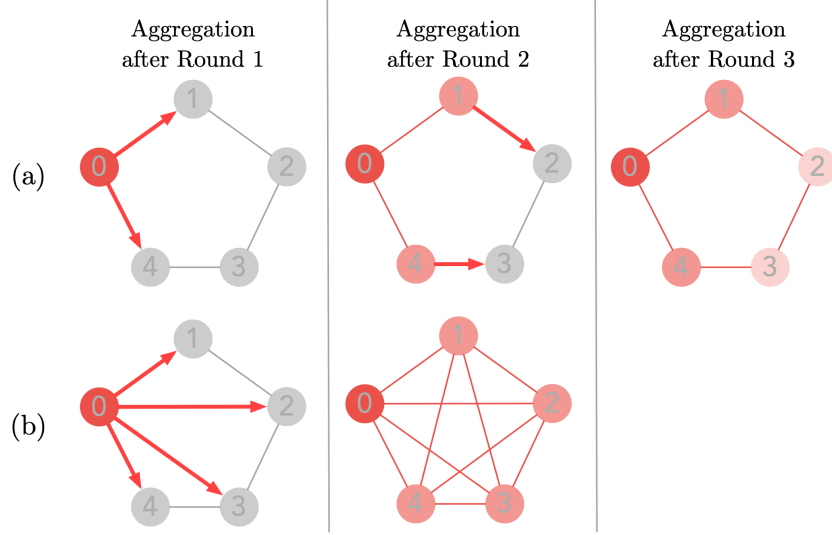


Figure 6.2: Propagation of attack for ring (a) and fully connected (b) topology

Label Flipping Attack (Fang [41])

The Fang label flipping attack [41] works by inverting the order of the image labels. Table 6.1 shows the scenarios that were evaluated for the Fang label flipping attack. This attack was evaluated with a total of 4 participant nodes, incrementing the number of malicious nodes until all of them are malicious.

Figure 6.3 shows the confusion matrices for scenarios containing between 1 and 4 malicious nodes.

- With just one malicious node, the performance of the aggregated model in **LF_1** does not degrade significantly, the confusion matrix still exhibits a clear diagonal from top left to bottom right.
- Increasing the number of malicious nodes to two in **LF_2**, i.e. when half the nodes are malicious, the aggregated model no longer produces a clear diagonal in the confusion matrix, but rather a cross-shaped pattern. The diagonals from the two benign nodes match up against the anti-diagonals from the malicious nodes and equally affect the aggregated performance.
- Further increasing the number of malicious nodes in **LF_3** and **LF_4** fully poisons the model, producing a clear anti-diagonal in the confusion matrix from inverting the label ordering.

#	Topology	n	f	AGR	Rounds	Attack
LF_1	Fully Connected	4	1	FedAvg	5	Fang [41]
LF_2	Fully Connected	4	2	FedAvg	5	Fang [41]
LF_3	Fully Connected	4	3	FedAvg	5	Fang [41]
LF_4	Fully Connected	4	4	FedAvg	5	Fang [41]

Table 6.1: Evaluation scenarios for Fang label flipping attack [41]

#	Setup	n	f	AGR	Rounds	Changed Labels
LTS_0	Fully Connected	5	0	FedAvg	5	None (No Attack, Baseline)
LTS_1	Fully Connected	5	1	FedAvg	5	1 $\rightarrow$ 7
LTS_2	Fully Connected	5	2	FedAvg	5	1 $\rightarrow$ 7
LTS_3	Fully Connected	5	3	FedAvg	5	1 $\rightarrow$ 7
LTS_4	Fully Connected	5	4	FedAvg	5	1 $\rightarrow$ 7
LTS_5	Fully Connected	5	5	FedAvg	5	1 $\rightarrow$ 7

Table 6.2: Evaluation scenarios for targeted, specific label flipping attack

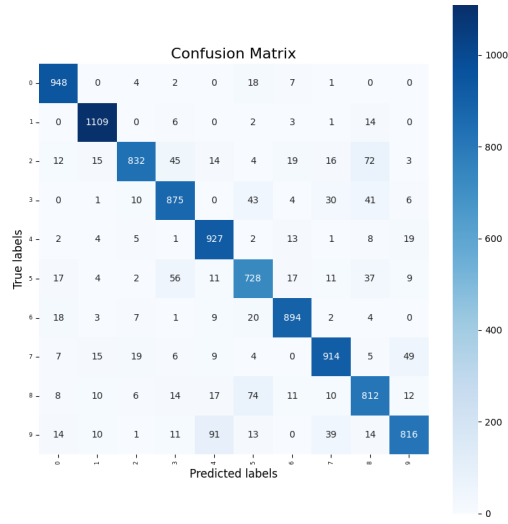
This is in line with expectations and shows that the Fang label flipping attack was correctly implemented.

Label Flipping Attack (targeted, specific)

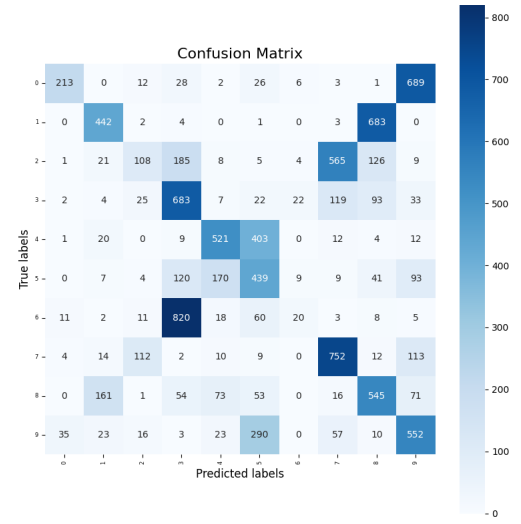
In the targeted, specific label flipping attack, only specific labels are changed to a different label. Table 6.2 shows the evaluation scenarios for the targeted, specific label flipping attack. A total of 5 nodes are used for the evaluation and the number of malicious nodes is incremented from 0 nodes in LTS_0 to all 5 nodes in LTS_5. In this attack the labels of images showing a 1 are changed from 1 to 7.

Figure 6.4 shows the confusion matrices for the targeted, specific label flipping attacks.

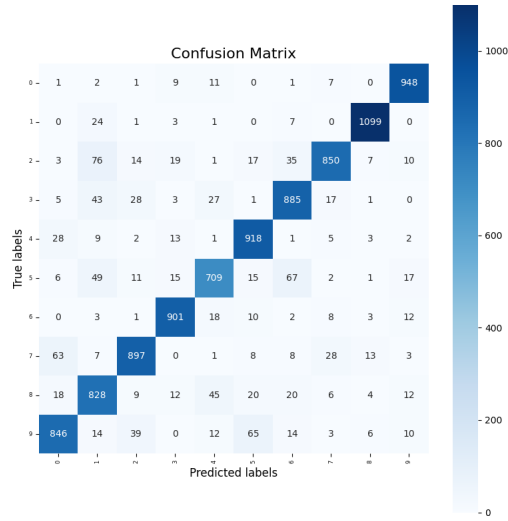
- In the case of 0 or 1 malicious nodes in **LTS_0** and **LTS_1** respectively, the attack does not appear to affect the results of the confusion matrix. As a large majority of nodes are still benign, the effects of the malicious nodes is suppressed.
- Beginning with **LTS_2**, the aggregated model begins to show the effects of the label flipping attack.
- Once the number of malicious nodes reaches majority in **LTS_3**, the effect is significantly more pronounced, with the model never predicting a 1 anymore.
- Increasing the number of malicious nodes further in **LTS_4** and **LTS_5** only marginally increases the performance of the label flipping attack, increasing the total number of predictions of "7" from 1026 in **LTS_3** to 1070 in **LTS_4** and 1091 in **LTS_5**.



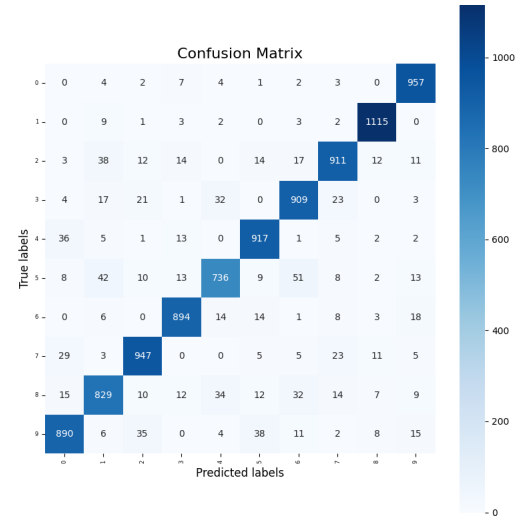
LF_1



LF_2



LF_3



LF_4

Figure 6.3: Confusion matrices for Fang label flipping attack [41]

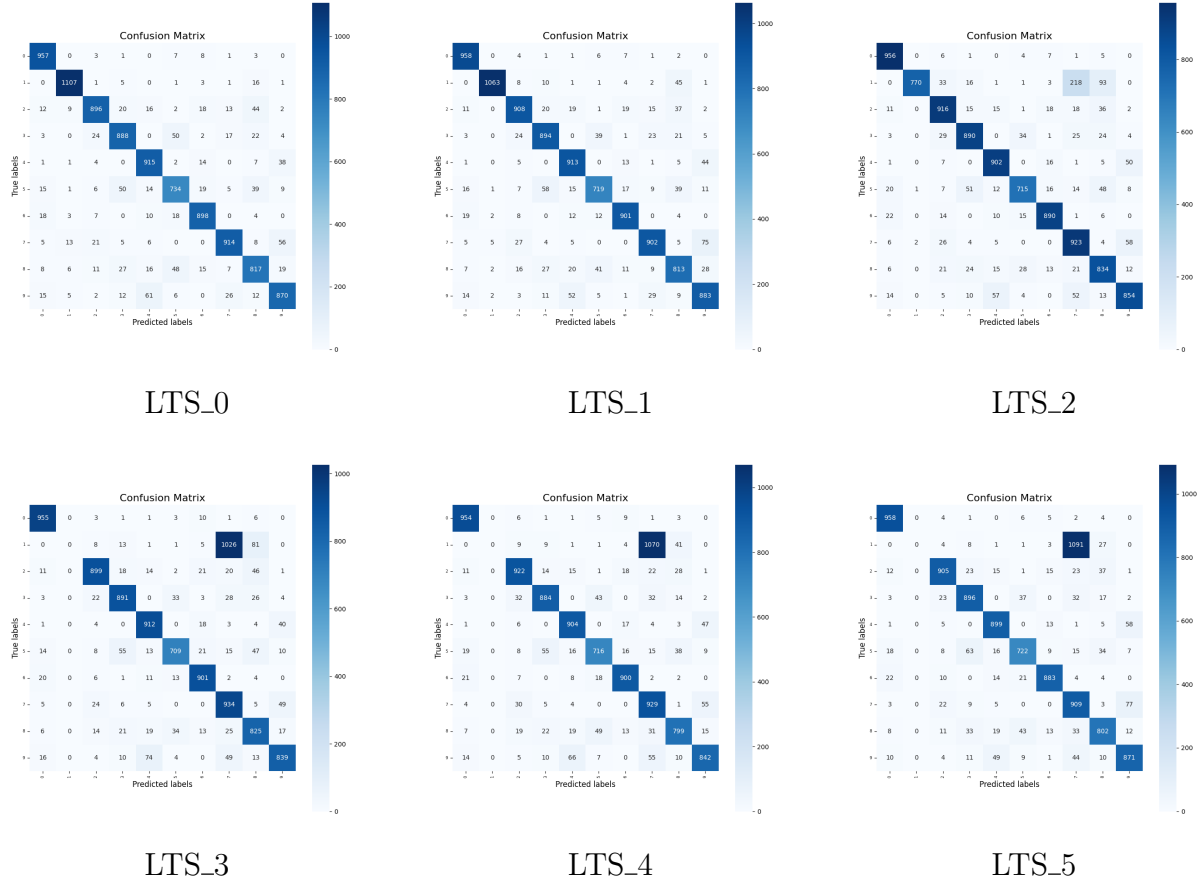


Figure 6.4: Confusion matrices for targeted, specific label flipping attack

These results are in line with evaluations from previous works [42], [43] and support the correctness of the implementation.

Figure 6.5 shows the validation accuracy of each participant node during the 5 round training process. The accuracy is evaluated on the test set after finishing the training epoch, but before model aggregation has occurred. Malicious nodes exhibit an accuracy that is roughly 10 percentage points lower than benign nodes, e.g. 75% instead of 85%. This is expected, as a similar amount of images per label and 10 labels in total (numbers from 0 to 9) means that one class makes up 10% of all images. If the targeted class no longer gets predicted, the accuracy ceiling drop from 100% to 90%.

Across all scenarios, there is a slight dip in accuracy after the second training round (round 1). This is likely because each model becomes highly specialized to its local data after the first round. The aggregated model following the first training round is a generalized compromise and a single round of retraining does not appear to be sufficient to overcome this loss in specialized accuracy. Further training rounds increase the local model performance for each participant node.

Figure 6.6 shows the validation accuracy of the aggregated model after each training round. Unlike the local models, the global, aggregated model did not exhibit a drop in accuracy after the second training round. The final accuracy after the 5 round training

process is depicted in Table 6.3. As long as the malicious nodes are the minority, there is only a slight drop in accuracy when increasing their number (e.g. from **LTS_0** to **LTS_1** and **LTS_2**). The model degradation saturates once malicious nodes reach majority, as almost all targeted images get misclassified. Accuracy drops from 89.96% with only benign nodes (**LTS_0**), to around 78.5% with only malicious nodes (**LTS_5**), roughly 11.5 percentage points. This is in line with the expected drop of 10 percentage points, being slightly higher likely due to imbalances in the label distribution.

Scenario	Accuracy (%)	Drop from Previous (pp)
LTS_0	89.96	–
LTS_1	89.54	-0.42
LTS_2	86.50	-3.04
LTS_3	78.65	-7.85
LTS_4	78.50	-0.15
LTS_5	78.45	-0.05

Table 6.3: Accuracy per LTS Scenario and drop compared to the previous number of malicious nodes

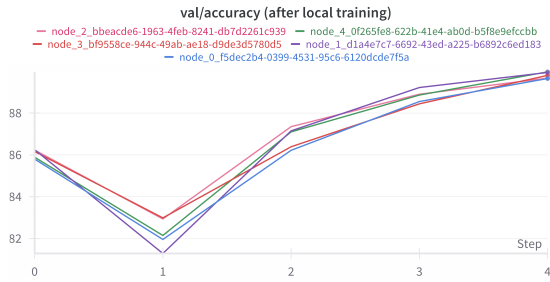
Label Flipping (untargeted)

The final implemented attack that is evaluated is the untargeted label flipping attack. Unlike the Fang and targeted attacks, where specific labels are manipulated in a certain way (e.g inversion or flipping to a specific other class), the untargeted attack indiscriminately flips a fraction of all labels to a random class. Table 6.4 shows the scenario setup for the untargeted label flipping scenarios. The federation trains for two rounds, with malicious nodes randomly changing the labels of 80% of the local training data. Six experiments were performed in total, incrementally increasing the number of malicious nodes from 0 (**LU_0**) to 5 (**LU_5**).

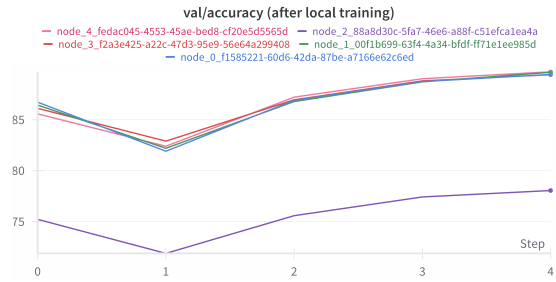
Figure 6.7 show the confusion matrices of the untargeted label flipping scenarios. Upon visual inspection, the performance does not significantly degrade when increasing the number of malicious nodes. Interestingly, the confusion matrix still exhibits a mostly clear diagonal, even when all 5 nodes are malicious (**LTS_5**). Since there are a total of 10

#	Setup	n	f	AGR	Rounds	Changed %
LU_0	Fully Connected	5	0	FedAvg	2	None (No Attack, Baseline)
LU_1	Fully Connected	5	1	FedAvg	2	80
LU_2	Fully Connected	5	2	FedAvg	2	80
LU_3	Fully Connected	5	3	FedAvg	2	80
LU_4	Fully Connected	5	4	FedAvg	2	80
LU_5	Fully Connected	5	5	FedAvg	2	80

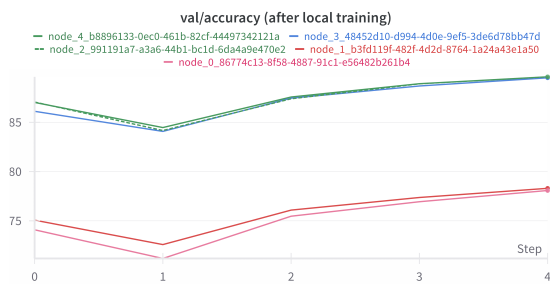
Table 6.4: Confusion matrices for untargeted label flipping attack



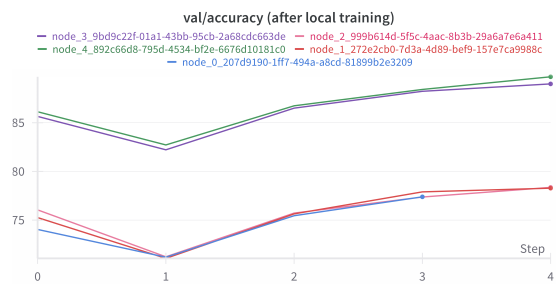
LTS_0



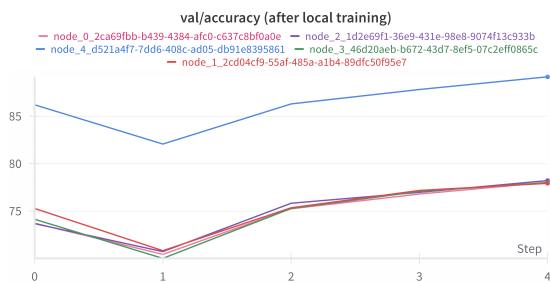
LTS_1



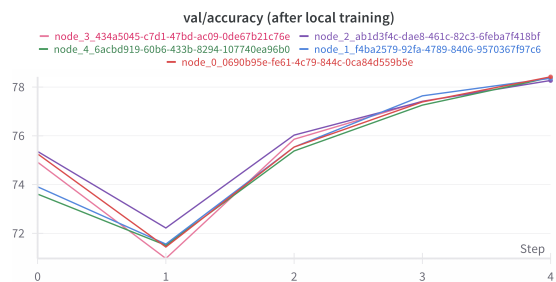
LTS_2



LTS_3

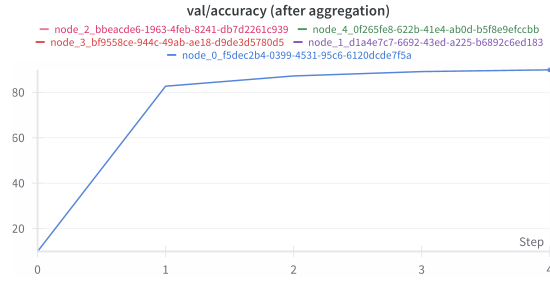


LTS_4

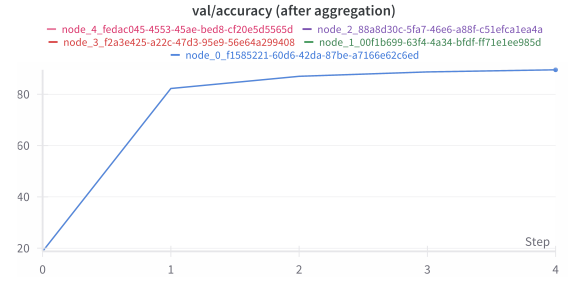


LTS_5

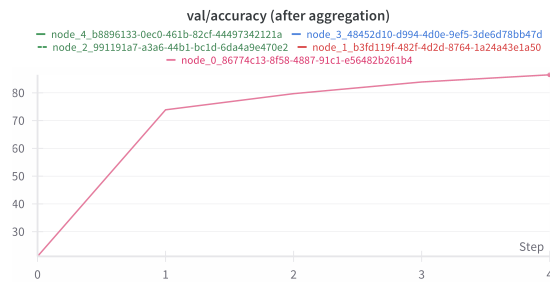
Figure 6.5: Local model validation accuracy (pre-aggregation) for targeted, specific label flipping attack



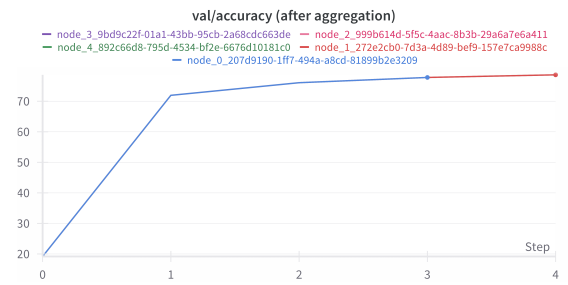
LTS_0



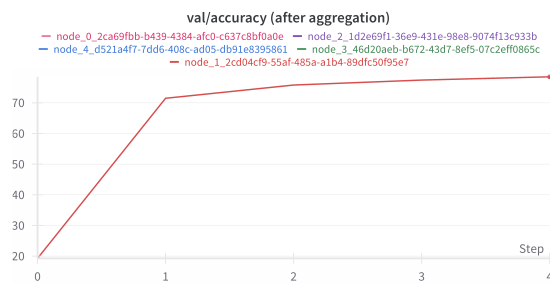
LTS_1



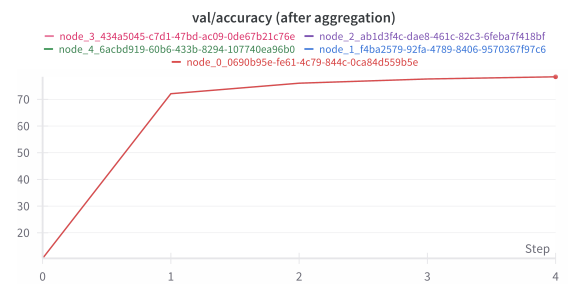
LTS_2



LTS_3



LTS_4



LTS_5

Figure 6.6: Global model validation accuracy (post-aggregation) for targeted, specific label flipping attack

classes, flipping 80% of the labels to a different class means that for a given class, 20% of labels are still correctly labeled. The remaining 80% are evenly divided across the other 9 classes. This means that the true class still remains the most likely. While 80% of the training data is now noise, the model is still able to learn from the correct 20%.

The global model accuracy is also shown in Table 6.5. This clearly shows that the model performance does indeed gradually degrade when incrementing the number of malicious nodes. Unlike in the other attacks, the degradation isn't most significant when malicious nodes enter majority (**LU_4**). Instead, the degradation gets continuously more significant when increasing the number of malicious clients.

Scenario	Accuracy (%)	Drop from Previous (pp)
LU_0	87.82	—
LU_1	87.82	0.00
LU_2	86.60	-1.22
LU_3	85.14	-1.46
LU_4	80.96	-4.18
LU_5	72.94	-8.02

Table 6.5: Accuracy per LU Scenario and drop compared to the previous number of malicious nodes

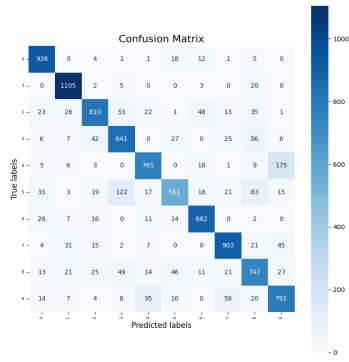
6.3 Overhead

Next to the correctness of the topologies and poisoning attacks, the performance of the framework is also investigated. The scalability of scenarios was investigated by increasing the number of training rounds for a given scenario configuration. The same model that was used for the poisoning attacks is used for this experiment. It is shown in code snippet 6.2. Since this experiment should evaluate the overhead induced by factors outside of the model training itself, a small and simple model was chosen that does not require long and intensive training.

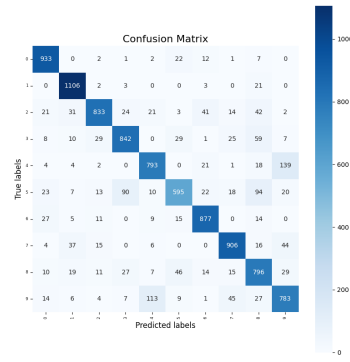
For this experiment, the same scenario was ran for 1, 5, 50, 100 and 1000 rounds. The resulting scenario runtime durations are shown in the log-log plot depicted in Figure 6.8. Two primary types of overhead could affect the scalability of the framework:

- **Fixed overhead** from the initial scenario setup would cause the curve to flatten at a low number of rounds, as this initial cost would dominate the total runtime.
- **Cumulative overhead** from e.g. growing database accesses would cause the curve to become progressively steeper

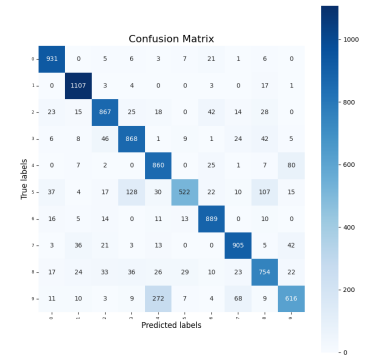
Since the plot shows a strong linear curve, it is evident that neither overhead significantly impacts scenario runtime. This highlights the efficiency of the developed framework.



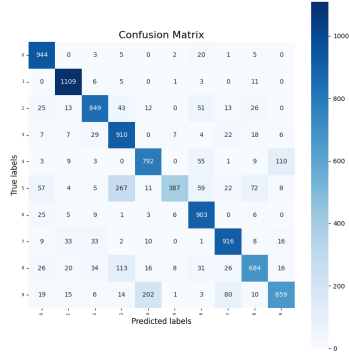
LU_0



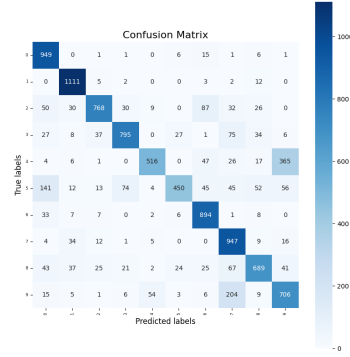
LU_1



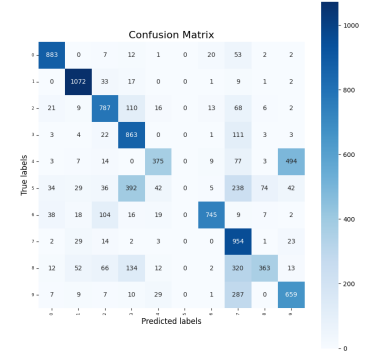
LU_2



LU_3



LU_4



LU_5

Figure 6.7: Confusion matrices for untargeted label flipping attack

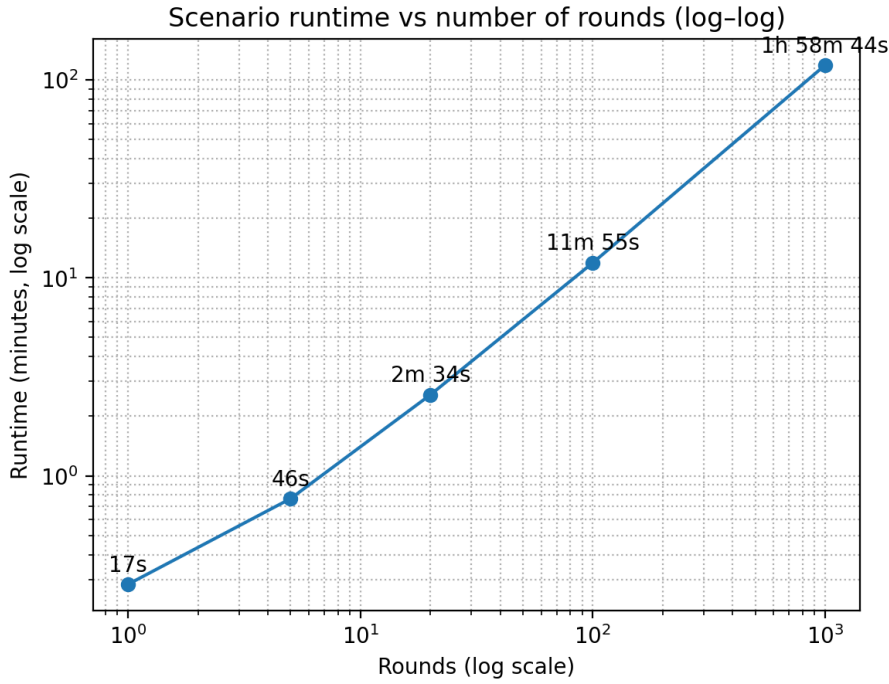


Figure 6.8: Log-Log plot of scenario runtime depending on number of rounds

Runtime measurements were also performed for different topologies to investigate their impact on the scenario performance. Table 6.6 presents the total scenario runtime for Fully Connected, Ring and Star topologies at 20 and 100 rounds of training.

Given the number of messages required for each topology:

- **Fully connected** topology requires $n(n - 1)$ messages.
- **Ring** topology requires $2n$ messages.
- **Star** topology requires $2(n - 1)$ messages.

The fully connected network requires the most communication between nodes while the star topology requires the least. A minor relationship between network connectivity and runtime is observable, where the fully connected topology consistently runs the longest and the star topology the shortest. The difference in runtime is extremely marginal however, which suggests that the overhead from exchanges messages between nodes is very small compared to the cost of local model training in each round.

Table 6.7 shows the effect on the runtime of the scenario when increasing the number of participating nodes . It considers the fully connected and ring topology, running 5 rounds per scenario.

For both topologies, the runtime increases steadily with the number of nodes, aligning with the expectations based on the growth of communication overhead and local training. As expected, with the fully connected case the runtime increases much faster. While five nodes only take 55 seconds, scaling up to 50 nodes increases runtime to over eight minutes.

This steep growth aligns with the quadratic communication cost of the fully connected topology, where the number of required messages grows as $n(n - 1)$.

In contrast, the runtimes of the ring topology increases slower, more linearly. Starting from 46 seconds at five nodes, the runtime only rises to 3 minutes and 24 seconds at 50 nodes. This more modest increase reflects the linear communication complexity of the ring, which requires only $2n$ messages.

This evaluation shows that communication cost becomes a significant factor when the system scales to larger node counts. While the differences between topologies were marginal at a fixed node count (Table 6.6), Table 6.7 demonstrates that topology makes a much larger difference when scaling up the number of participants. While the fully connected network quickly becomes impractical at scale, the ring topology remains relatively efficient. The findings align with the expectations, indicating that the implementation behaves correctly.

Topology	Runtime
20 Rounds	
Fully Connected	3m 23s
Ring	3m 19s
Star	3m 16s
100 rounds	
Fully Connected	16m 27s
Ring	16m 21s
Star	16m 15s

Table 6.6: Scenario runtime depending on topology and number of rounds

Node Count	Topology	Rounds	Runtime
5	Fully Connected	5	55s
15	Fully Connected	5	1m 27s
25	Fully Connected	5	2m 18s
50	Fully Connected	5	8m 28s
5	Ring	5	46s
15	Ring	5	1m 17s
25	Ring	5	1m 58s
50	Ring	5	3m 24s

Table 6.7: Scenario runtime depending on topology and number of nodes

6.4 Usability

This section outlines the findings from a usability study conducted on the platform. The goal was to evaluate the ease of use and overall user experience for individuals with varying levels of expertise in Federated Learning (FL). The study involved two participants who were asked to perform a core task on the platform after a brief introduction.

Participants

Person 1: A university student with a foundational understanding of Machine Learning (ML) concepts and experience with libraries like Scikit-learn and PyTorch. Person 1 has no prior experience with Federated Learning or its specific frameworks.

Person 2: A university graduate with some practical experience in Federated Learning, having previously used other frameworks, notably FedStellar and NEBULA, for simple experiments.

Methodology

Both participants were given the source code of both the Coordinator as well as the participant repository. They were first asked to read the project’s README.md files as their main source of documentation. They were then given the same task to complete. Their experience and feedback were recorded through an oral post-task session.

Task

The following task was given to the participants: *Using the web interface, configure and run a federated learning scenario with 5 participants for 5 rounds in a fully connected network. Configure the scenario to simulate a label flipping attack, with 2 nodes changing all "1" to "7". After launching the scenario, find where you can monitor its progress and view the logs generated by the participants. Then, measure the loss in precision / recall of the targeted class ("1") by looking at the confusion matrix after the aggregation of the final round.*

Questions and Findings

- *Did you find the background information (provided in the ‘README.md’) clear and easy to follow? Were there any parts that felt ambiguous or confusing?*

The participants generally found the documentation helpful for getting started. However, some parts, particularly some technical terms were unclear and lacked explanations. Supplementary materials such as overview diagrams or example configurations would have supported users.

- *Did you face any challenges during setting up and executing the scenario? If yes, what kind of challenges?*
Scenario setup and execution were completed successfully by both participants. However, one participant encountered issues installing dependencies. Overall, the platform was perceived as lightweight and easy to get started.
- *How confident were you that the configuration is correct after running the task?*
The participants reported moderate to high confidence in their scenario configuration. Some uncertainty regarding the attack configuration was reported.
- *How straightforward was it to analyze the progress of the scenario and view the logs?*
The live log feed and metrics view were found to be intuitive and useful. Participants appreciated the real-time updates and structured presentation of node-level logs and metrics. However, it was suggested that filtering capabilities (by node or log level) could enhance usability further.
- *How would you rate the overall user experience of the platform on a scale of 1 to 5 (1 being very poor and 5 being excellent)?*
The overall usability of the platform was rated positively. The participants highlighted the interface, logical workflow and minimal setup effort. Critique points were mostly focused on the lack of contextual explanations in the user interface.
- *Based on your first interaction, do you have any suggestions for improving the platform to make tasks like this easier for new users?*
Participants recommended several enhancements to improve the user experience, including a quickstart tutorial, more informative tooltips and sanity checks before launching scenarios.

The documentation of both the Coordinator as well as the participant repository were improved after concluding the usability study. Additionally, some parts of the user interface were changed (tooltips, button labels) to address the findings mentioned earlier.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis. After a summary of the work done in this thesis, some of the limitations of the developed framework are discussed, as well as some possible avenues for future work.

7.1 Conclusion

This thesis designed, implemented, and evaluated a lightweight platform for decentralized federated learning (DFL) that addresses shortcomings of existing solutions. Next to the development of the framework itself, an additional core contribution is the introduction of a hybrid architecture that keeps orchestration of the federated training process in a central coordinator while leaving model training, gossiping, the peer-to-peer parameter exchange and aggregation fully decentralized across participants. This architectural innovation allows to keep the benefits of a central control and monitoring instance, without sacrificing the advantages of decentralized setups. The platform was designed with modularity in mind, using object-oriented programming patterns to simplify inclusion of new attacks, aggregation methods or training libraries. Scenario management and experiment tracking are available through the highly responsive Web UI, which also provides extensive information on logs produced by the coordinator and participants, together with training metrics. Communication between the central coordinator and the participant nodes is implemented with WebSockets, which combined with the MongoDB database's 'change stream' feature allows for real-time updates of the running scenario. The platform supports multiple topologies (fully connected, ring, star, random), data and model poisoning attacks, such as targeted and untargeted label flipping attacks or a model update manipulation attack. By providing clear locations for their inclusion, new poisoning attacks can be quickly tested and evaluated.

During the evaluation phase the correctness of the implementation was verified through unit tests for the topology and qualitative analysis of poisoning attack results with varying fractions of malicious clients. Label-flipping experiments reproduced expected effects as the number of malicious nodes increased (including the sharp degradation once attackers

reach a majority), confirming the correctness of attack implementations and the aggregation loop. Tests were also conducted, showing that the developed framework exhibits very little overhead, as there was a linear relationship between the scenario runtime and the number of training rounds. Also, runtime differences were marginal across topologies, which suggests a very low communication overhead relative to the local training cost.

A usability study was performed with two participants, one with no experience in Federated Learning and the other with prior experience working with frameworks such as FedStellar and NEBULA. It showed that the framework was simple to setup and operate regardless of experience, with the interface and logical workflow being highlighted as clear positives.

7.2 Future Work

Some avenues for future work include:

- **Improve Scalability:** The system has been tested with up to 50 nodes across different topologies. However, in fully connected setups the platform begins to struggle, mainly due to the large number of direct communication paths between the nodes. For a scenario involving a very large number of nodes, alternative approaches may be necessary. One promising direction is to make use of protocols such as IPFS, in order to only share the addresses of stored parameters instead of the parameters themselves. Such indirect communication strategies are rarely seen employed, but they seem to offer strong potential for improving scalability.
- **Networking:** Support for multi-host deployments through an overlay network such as Tailscale¹ could enable in-field deployments.
- **Resilience:** Mechanisms to handle node churn (join, leave, fail, rejoin) could be implemented to extend the functionality of the framework and improve the resilience of the running scenario.
- **Database (MongoDB):** Filtering / searching within the documents lead to slow-downs, especially in large scenarios. Separating the logging and statistic service to a relational database, or even running them on a separate host, would improve performance, scalability and frontend responsiveness.
- **Frontend UX:** The user interface could be further extended with features, such as configuration sanity checks, richer tooltips, and advanced log filtering to further lower the barrier of entry for new users.
- **General:** The framework could be extended with additional features present in other platforms (e.g. more topologies, poisoning attacks, ...)

¹<https://tailscale.com/>

Bibliography

- [1] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, “Federated learning of deep networks using model averaging”, *CoRR*, vol. abs/1602.05629, 2016. arXiv: 1602.05629. [Online]. Available: <http://arxiv.org/abs/1602.05629>.
- [2] J. Ding, E. Tramel, A. K. Sahu, S. Wu, S. Avestimehr, and T. Zhang, *Federated learning challenges and opportunities: An outlook*, 2022. arXiv: 2202.00807 [cs.LG].
- [3] 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, May 2020. DOI: 10.1109/msp.2020.2975749. [Online]. Available: <https://doi.org/10.1109%2Fmsp.2020.2975749>.
- [4] X. Yu, L. Li, X. He, S. Chen, and L. Jiang, “Federated learning optimization algorithm for automatic weight optimal”, *Computational Intelligence and Neuroscience*, vol. 2022, 2022. DOI: 10.1155/2022/8342638. [Online]. Available: <https://doi.org/10.1155/2022/8342638>.
- [5] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, “Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges”, *IEEE Communications Surveys & Tutorials*, 2023.
- [6] M. Moshawrab, M. Adda, A. Bouzouane, H. Ibrahim, and A. Raad, “Reviewing federated learning aggregation algorithms; strategies, contributions, limitations and future perspectives”, *Electronics*, vol. 12, no. 10, p. 2287, 2023.
- [7] E. T. M. Beltrán, M. Q. Pérez, P. M. S. Sánchez, *et al.*, *Decentralized federated learning: Fundamentals, state-of-the-art, frameworks, trends, and challenges*, 2023. arXiv: 2211.08413 [cs.LG].
- [8] T. Sun, D. Li, and B. Wang, “Decentralized federated averaging”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 4, pp. 4289–4301, 2023. DOI: 10.1109/TPAMI.2022.3196503.
- [9] T. Vogels, H. Hendrikx, and M. Jaggi, *Beyond spectral gap: The role of the topology in decentralized learning*, 2022. arXiv: 2206.03093 [cs.LG].
- [10] A. Bellet, A.-M. Kermarrec, and E. Lavoie, “D-cliques: Compensating noniidness in decentralized federated learning with topology”, *CoRR*, vol. abs/2104.07365, 2021. arXiv: 2104.07365. [Online]. Available: <https://arxiv.org/abs/2104.07365>.

- [11] S. Thomas and P. Myakala, “Beyond the cloud: Federated learning and edge ai for the next decade”, *Journal of Computer and Communications*, vol. 13, pp. 37–50, 2025. DOI: 10.4236/jcc.2025.132004.
- [12] E. T. M. Beltrán, *Fedstellar: Framework for decentralized federated learning*, 2023. [Online]. Available: <https://federatedlearning.inf.um.es/>.
- [13] E. T. M. Beltrán, *Nebula: Framework for decentralized federated learning*, 2024. [Online]. Available: <https://federatedlearning.inf.um.es/>.
- [14] D. J. Beutel, T. Topal, A. Mathur, *et al.*, “Flower: A friendly federated learning research framework”, *arXiv preprint arXiv:2007.14390*, 2020.
- [15] E. T. Martínez Beltrán, O. L. Perales Gómez, C. Feng, *et al.*, “Fedstellar: A platform for decentralized federated learning”, *Expert Systems with Applications*, vol. 242, p. 122 861, 2024, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2023.122861.
- [16] E. T. Martínez Beltrán, M. Quiles Pérez, P. M. Sánchez Sánchez, *et al.*, “Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges”, *IEEE Communications Surveys Tutorials*, vol. 25, no. 4, pp. 2983–3013, 2023. DOI: 10.1109/COMST.2023.3315746.
- [17] HXSu, *Decentralized federated learning framework based on smart contract and ipfs*, <https://github.com/HXSu/DFL-Framework>, Accessed: 2025-06-09, 2025. [Online]. Available: <https://github.com/HXSu/DFL-Framework>.
- [18] G. Syros, G. Yar, S. Boboila, C. Nita-Rotaru, and A. Oprea, *P2pfl: Peer-to-peer federated learning*, <https://github.com/p2pfl/p2pfl>, Accessed: 2025-06-09, 2023. [Online]. Available: <https://github.com/p2pfl/p2pfl>.
- [19] Y. Liu, T. Fan, T. Chen, Q. Xu, and Q. Yang, “Fate: An industrial grade platform for collaborative learning with data protection”, *Journal of Machine Learning Research*, vol. 22, no. 226, pp. 1–6, 2021. [Online]. Available: <http://jmlr.org/papers/v22/20-815.html>.
- [20] C. He, S. Li, J. So, *et al.*, “Fedml: A research library and benchmark for federated machine learning”, *CoRR*, vol. abs/2007.13518, 2020. arXiv: 2007.13518. [Online]. Available: <https://arxiv.org/abs/2007.13518>.
- [21] Z. Wang, W. Kuang, Y. Xie, *et al.*, *Federatedscope-gnn: Towards a unified, comprehensive and efficient package for federated graph learning*, 2022. arXiv: 2204.05562 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2204.05562>.
- [22] P. Foley, M. J. Sheller, B. Edwards, *et al.*, *Openfl: The open federated learning library*, 2022. DOI: 10.1088/1361-6560/ac97d9. [Online]. Available: <http://iopscience.iop.org/article/10.1088/1361-6560/ac97d9>.
- [23] D. Zeng, S. Liang, X. Hu, H. Wang, and Z. Xu, *Fedlab: A flexible federated learning framework*, 2023. [Online]. Available: <http://jmlr.org/papers/v24/22-0440.html>.
- [24] Z. Wang, X. Fan, Z. Peng, *et al.*, *Flgo: A fully customizable federated learning platform*, 2023. arXiv: 2306.12079 [cs.LG].
- [25] F. Lai, Y. Dai, S. S. Singapuram, *et al.*, “Fedscale: Benchmarking model and system performance of federated learning at scale”, in *International Conference on Machine Learning (ICML)*, 2022.

- [26] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, “Oort: Efficient federated learning via guided participant selection”, in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
- [27] W. Lu and J. Wang, *Personalizedfl: Personalized federated learning toolkit*, <https://github.com/microsoft/PersonalizedFL>.
- [28] W. Zhuang, X. Gan, Y. Wen, and S. Zhang, “Easyfl: A low-code federated learning platform for dummies”, *IEEE Internet of Things Journal*, 2022.
- [29] G. T. contributors, *Tensorflow federated: A framework for machine learning and other computations on decentralized data*, <https://github.com/google-parfait/tensorflow-federated>, Accessed: 2025-06-09, 2025.
- [30] A. Nandi, F. Xhafa, and R. Kumar, “A docker-based federated learning framework design and deployment for multi-modal data stream classification”, *Computing*, vol. 105, no. 10, pp. 2195–2229, May 2023, ISSN: 0010-485X. DOI: 10.1007/s00607-023-01179-5. [Online]. Available: <https://doi.org/10.1007/s00607-023-01179-5>.
- [31] T. Charboneau, *How to fix and debug docker containers like a superhero*, Docker Blog, Accessed: 2025-07-22, Oct. 2022. [Online]. Available: <https://www.docker.com/blog/how-to-fix-and-debug-docker-containers-like-a-superhero/>.
- [32] P. Saha, A. Beltre, P. Uminski, and M. Govindaraju, “Evaluation of docker containers for scientific workloads in the cloud”, in *Proceedings of the Practice and Experience on Advanced Research Computing*, ser. PEARC ’18, ACM, Jul. 2018, pp. 1–8. DOI: 10.1145/3219104.3229280. [Online]. Available: <http://dx.doi.org/10.1145/3219104.3229280>.
- [33] gRPC Authors, *Grpc: A high-performance, open-source universal rpc framework*, Accessed: 2025-07-22, 2016. [Online]. Available: <https://grpc.io/>.
- [34] R. T. Fielding, “Architectural styles and the design of network-based software architectures”, Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: <https://ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [35] G. LLC, *Protocol buffers*, Accessed: 2025-07-22, 2024. [Online]. Available: <https://protobuf.dev/>.
- [36] J. Benet, *Ipfs - content addressed, versioned, p2p file system*, 2014. arXiv: 1407.3561 [cs.NI]. [Online]. Available: <https://arxiv.org/abs/1407.3561>.
- [37] W. Constable and W. Feng, *Getting started with distributed data parallel*, Accessed: 2025-07-22, 2024. [Online]. Available: https://docs.pytorch.org/tutorials/beginner/dist_overview.html.
- [38] A. Stanford-Clark and A. Nipper, *Mqtt for sensor networks*, Accessed: 2025-07-22, 2007. [Online]. Available: http://mqtt.org/MQTT-SN_spec_v1.2.pdf.
- [39] O. Standard, “Mqtt version 3.1.1”, OASIS, Tech. Rep., 2014. [Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [40] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, 2010, ISBN: 9781593272203.

- [41] M. Fang, X. Cao, J. Jia, and N. Z. Gong, “Local model poisoning attacks to byzantine-robust federated learning”, *CoRR*, vol. abs/1911.11815, 2019. arXiv: 1911.11815. [Online]. Available: <http://arxiv.org/abs/1911.11815>.
- [42] T.-T. Näscher and W. Rozek, *Feature integration for an open-source decentralized federated learning platform*, Zürich, Switzerland: Communication Systems Group, Department of Informatics, Jan. 2025. [Online]. Available: <https://files.ifi.uzh.ch/CSG/staff/feng/external/theses/MPA-tnaescher-wrozek.pdf>.
- [43] T.-T. Näscher, *Poisoning attack behavior detection in federated learning*, Zürich, Switzerland: Communication Systems Group, Department of Informatics, May 2023. [Online]. Available: <https://files.ifi.uzh.ch/CSG/staff/feng/external/theses/BA-timothy-naescher.pdf>.

List of Figures

2.1	Network Topologies	8
5.1	State machine for scenario flow	28
5.2	Scenario configuration window	41
5.3	Scenario launch overview window	42
5.4	Live scenario dashboard window	43
5.5	Scenario history window	43
5.6	Scenario logs window	44
5.7	Scenario stats window	45
5.8	Scenario confusion matrices window	46
6.1	Tests	58
6.2	Propagation of attack for ring (a) and fully connected (b) topology	61
6.3	Confusion matrices for Fang label flipping attack [41]	63
6.4	Confusion matrices for targeted, specific label flipping attack	64
6.5	Local model validation accuracy (pre-aggregation) for targeted, specific label flipping attack	66
6.6	Global model validation accuracy (post-aggregation) for targeted, specific label flipping attack	67
6.7	Confusion matrices for untargeted label flipping attack	69
6.8	Log-Log plot of scenario runtime depending on number of rounds	70

List of Tables

3.1	Comparison of Federated Learning platforms by DFL support and topology.	10
3.2	Node execution models of Federated Learning platforms.	11
3.3	Languages, ML Libraries, and Node Communication Mechanisms of Federated Learning Platforms.	12
3.4	Development Status, Popularity, and Open Source Availability of Federated Learning Platforms.	13
5.1	API Endpoints Overview by Functional Group	32
6.1	Evaluation scenarios for Fang label flipping attack [41]	62
6.2	Evaluation scenarios for targeted, specific label flipping attack	62
6.3	Accuracy per LTS Scenario and drop compared to the previous number of malicious nodes	65
6.4	Confusion matrices for untargeted label flipping attack	65
6.5	Accuracy per LU Scenario and drop compared to the previous number of malicious nodes	68
6.6	Scenario runtime depending on topology and number of rounds	71
6.7	Scenario runtime depending on topology and number of nodes	71

Appendix A

Usability Evaluation Questions and Answers

These responses are paraphrased from oral interviews. Participant 1:

- *Did you find the background information (provided in the ‘README.md’) clear and easy to follow? Were there any parts that felt ambiguous or confusing?*
“It was mostly understandable, but some of the terms like the different attack types or scenario structure weren’t explained in much detail. A diagram or working example would’ve helped clarify things.”
- *Did you face any challenges during setting up and executing the scenario? If yes, what kind of challenges?*
“Yes, there was an error installing PyTorch, so this dependency was missing at first. Once I fixed that, everything ran smoothly. Generally, it was quite easy to get started.”
- *How confident were you that the configuration is correct after running the task?*
“I was fairly confident overall. The form made most of the config straightforward, but I wasn’t sure whether the label flipping was actually applied correctly. There wasn’t really any feedback about that part, apart from the stuff shown in the JSON.”
- *How straightforward was it to analyze the progress of the scenario and view the logs?*
“The logs and metrics were clear and updated in real time, which was nice. I’d have liked to filter the logs though. For me, there’s too much happening at once. Also, it wasn’t visually clear which node is the attacker.”
- *How would you rate the overall user experience of the platform on a scale of 1 to 5 (1 being very poor and 5 being excellent)?*
“I’d give it a 4. The experience was smooth, but a few explanations or tooltips in the UI would’ve made it even better.”
- *Based on your first interaction, do you have any suggestions for improving the platform to make tasks like this easier for new users?*

“A short quickstart guide with steps or screenshots would really help. Also, something like a config check before launching a scenario would give more confidence that things are set up correctly.”

Participant 2:

- *Did you find the background information (provided in the ‘README.md’) clear and easy to follow? Were there any parts that felt ambiguous or confusing?*
“The documentation was well-structured and got me up and running quickly, but I would’ve liked a system overview or architecture diagram. That would help especially new users understand what components talk to each other.”
- *Did you face any challenges during setting up and executing the scenario? If yes, what kind of challenges?*
“No real challenges. I first tried to install the dependencies with conda, which didn’t work. Switching to pip according to the instructions made everything work for me.”
- *How confident were you that the configuration is correct after running the task?*
“I was confident about most of the config, but there’s no validation step — so to be 100% sure that the attack configuration was actually applied, i had to search through the provided configuration document. It would be great to get a confirmation or warning if something’s missing.”
- *How straightforward was it to analyze the progress of the scenario and view the logs?*
“It was very intuitive. The structured log view made it easy to follow what was happening. It’d be useful to have filters or log export though — especially when you’re debugging. The Live Log Feed is nearly unusable, due to the scenario progressing way too fast to read anything.”
- *How would you rate the overall user experience of the platform on a scale of 1 to 5 (1 being very poor and 5 being excellent)?*
“I’d rate it a 5. The UI is lightweight and focused. You don’t get overwhelmed with features, and the important settings are right there.”
- *Based on your first interaction, do you have any suggestions for improving the platform to make tasks like this easier for new users?*
“I’d suggest adding confirmations before scenario launch and maybe a short tutorial video that shows how to configure and execute a full scenario.”