SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

FLOps: Practical Federated Learning via Automated Orchestration (on the Edge)

Alexander Malyuk

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

FLOps: Practical Federated Learning via Automated Orchestration (on the Edge)

TODO

Author:	Alexander Malyuk
Supervisor:	Prof. Dr-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan, Giovanni Bartolomeo
Submission Date:	15.09.2024

Acknowledgments

Abstract

Kurzfassung

Contents

Abbreviations

This is a list of repeatedly occurring acronyms in the thesis. Abbreviations that are only used once are explained in the text and omitted from this list, to focus on the important once. This list also includes acronyms that are well known and that are not explicitly explained in the text. For completion they are included here.

Specific Acronyms :

FL Federated Learning
CFL Clustered Federated Learning
HFL Hierarchical Federated Learning
PFL Personalized Federated Learning
MLOps Machine Learning Operations
CI Continuous Integration
CD Continuous Delivery & Deployment
IID Independent and Identically distributed

Common Acronyms :

AI Artificial Intelligence
ML Machine Learning
DNN Deep Neural Network
API Application Programming Interface
GUI Graphical User Interface
SLA Service-Level Agreement
CLI Command-Line Interface
IoT Internet of Things

1 Introduction

The number of smart devices has been rapidly growing in the last several years. Improvements in connectivity (Cloud Computing & Internet of Things—IoT), connection speeds (5G), and computing power enable this increasing fleet of edge/mobile devices to generate enormous amounts of data (BigData). Combined with the expansion of AI/ML, this data is a driving factor for current successful workflows and future advancements. This complementing union of technologies plays a key role in elevating various domains, from agriculture and healthcare to education and the security sector, to Industry 4.0 and beyond. Diverse and complex challenges arise from this swiftly evolving landscape. [3]

1.1 Problem Statement

With great access to data comes great responsibility that can be easily exploited. Many of the aforementioned machines are personal user devices or belong to companies and organizations that handle customer or internal resources. These devices store and handle sensitive private data.

In classic (large-scale) Machine Learning, data gets sent from client devices to a centralized server, which usually resides in the cloud. The collected data is used on the server to train ML models or perform inference serving. This approach provides direct access to this sensitive data and the power to trace back its origin, creating a breach of privacy. [12]

Governments and organizations have established laws and regulations to prohibit potential abuse of sensitive data. Examples include the European Parliament regulation to protect personal data [16] or the California Consumer Privacy Act (CCPA) [11]. These measures aim to support cooperation between organizations and nations while protecting trade secrets. However, some laws and regulations prohibit sharing or moving data to other countries or even off-premises. [12]

Ignoring and no longer using this large amount of data would heavily limit current workflows and further developments for many data-dependent and data-hungry technologies. In 2017, a team of Google researchers introduced Federated Learning (FL) as one possible solution to utilize sensitive data while keeping it private. Instead of collecting the data on a server and training ML models centralized, in FL, the model training occurs directly on the client devices. Afterward, the individually trained models get sent to the server, which combines the collected models into a single shared one. This so-called global model can then be distributed to the clients again for further training cycles. Therefore, FL enables training a shared model on sensitive data while keeping that data secure on the local client devices. [13]

While many researchers are actively engaged in the field of FL, the majority of them are focused on enhancing existing FL components, strategies, and algorithms or devising better ways of doing FL. However, there is a noticeable scarcity of work that concentrates on the crucial aspects of the initial setup, deployment, and usability of FL. We delve into this issue in greater detail in the dedicated background section (2.1.4).

Because FL is a relatively modern technique, it lacks a sophisticated production-grade ecosystem that includes frameworks and libraries that improve ease of use by automating its setup and execution. As a result, contributing to the field of FL or reproducing findings is a task ranging from non-trivial to improbable due to the lack of documented steps regarding setup, deployment, management, and execution. Instead of using a shared set of tools for bootstrapping to make progress on novel work more efficiently, one needs to set up and manage FL from the ground up. Note that a small set of emerging libraries and frameworks exists for FL. Instead of orchestrating FL on real distributed devices, they focus on executing FL algorithms and processes, often via virtual simulations. Not to mention utilizing more advanced techniques to increase productivity that other domains have already been using for several years, such as modern DevOps practices like continuous deployment. We review existing FL tools in detail in the dedicated section (2.1.6).

1.2 Motivation

Building or contributing to a novel FL framework or library focusing on the previously mentioned challenges could soften or entirely alleviate those problems. We are talking about a tool that sees Docker [6] and Kubernetes [10] as role models and strives to be comparable to them but for the discipline of FL. It should specialize in the setup, deployment, component management, and automation, in short, FL orchestration. Allowing researchers, developers, and end-users to set up, perform, reproduce, and experiment with FL in a more accessible way.

The goal of this tool should be to automate and simplify complex tasks, reducing the required level of expertise in various domains, ranging from ML/FL, dependency management, containerization technologies, and automation to orchestration. Such a tool would empower less experienced individuals to participate and contribute to the field of FL. As a result, FL could be adapted and used by more people in more areas.

This tool would streamline and accelerate existing workflows and future progress by utilizing reliable automation to avoid error-prone manual tasks. With its potential to optimize, standardize and unify processes, our envisioned tool could become a significant part of the emerging FL ecosystem, contributing to the development and progress of the entire field.

1.3 Objectives

The motivation allows us to distill the following key objectives for such a tool.

Improve Accessibility
Making FL more accessible by abstracting away and automating complexities, enables more individuals to engage with it. Expanding FL to more areas will increase its usage and user base, raising general interest and relevance for its field, which should aid its development.

Utilize Automation
Automating tedious, error-prone, and repetitive manual tasks necessary to perform FL will free up time and resources for more critical work, leading to further advancements.

Prioritize Tangible Applicability
As we discuss in (2.1.4), FL struggles with a gap between research/virtual-simulation and practical application in real production environments. This tool should focus on being usable in real physical conditions on distributed devices. It should be feasible to incorporate this tool into existing workflows.

Embrace Plasticity
Because FL is such a young field, it faces constant change. Naturally, our tool should welcome change in the form of extendability and adaptability. This tool should be flexible and applicable to a myriad of use cases and scenarios. It should be easy to modify to accommodate evolving needs. Likewise, this tool should profit from existing technologies to offer a higher level of quality than creating everything from square one.

1.4 Contribution

We introduce FLOps to fulfill the objectives above. It enables individuals to use, develop, and evaluate tangible FL. FLOps enriches FL with modern best practices from automation, DevOps/MLOps, and orchestration. FLOps improves accessibility by enabling users without experience in FL, MLOps, or orchestration to do FL and still benefit from these technologies via automated orchestration.

To do FL, users simply provide a link to their ML git repository. Note that this code needs to satisfy some simple structural prerequisites. This repository code gets automatically augmented by FLOps to support FL. FLOps creates a containerized image with all necessary dependencies to do FL training. These images are automatically built and adhere to best practices, ensuring they are as fast and lightweight as possible. FLOps can build these images for multiple different target platforms. Thus, FL components can run on ARM edge-devices like Raspberry Pis or Nvidia Jetsons. FLOps enables FL on all devices that support containerization technologies like Docker [6] or containerd [5]. This approach eliminates the need for tedious device setup and the struggle to configure heterogeneous dependencies to match the necessary requirements for training, thereby streamlining the process and saving time.

FLOps automatically performs FL training based on the user-requested configuration. Users can, for example, specify resource requirements, the number of training rounds, the FL algorithm, the minimum number of participating client devices, and more. During runtime, users can observe this training process via a sophisticated GUI, which allows users to monitor, compare, store, export, share, and organize training runs, metrics, and trained models. FLOps can automatically build inference servers based on the trained model. This inference server can be pulled as a regular image. FLOps can also directly deploy this trained-model image as an inference server.

A multitude of diverse technologies and areas are necessary for FLOps to provide its services. Instead of reimplementing these complex features in a subpar fashion from scratch, we benefit from combining and extending existing solutions and technologies in unique and novel ways. This includes the use of Anaconda [1] and Buildah [4] to manage dependencies and build images. We utilize a pioneering FL framework called Flower [8] to execute the FL training loop. The mentioned runtime observability features are available via a mature MLOps tool called MLflow [14]. Because FL pushes model training to client devices, especially edge devices, we decided to use an orchestrator native to the edge environment. With the help of Oakestra [2], FLOps can deploy and orchestrate its components. FLOps is implemented as an separate addon for Oakestra. Because their interaction is based on general API endpoints and SLAs, FLOps can be modified to support other Orchestrators.

It is noteworthy that these different tools do not natively support each other. FLOps combines them in unprecedented ways to achieve its goals. As an example, FLOps supports hierarchical FL (HFL), which is not directly supported or offered by Flower. To the best of our knowledge, FLOps is the first work that combines Flower with MLflow and allows HFL, as well as automatically converts ML code into FL enabled containerized images.

As far as we know, the term FLOps, besides being a measurement unit for computer performance (Floating point operations per second), has not been used or applied to FL unlike MLOps has been used to describe DevOps techniques for ML. The goal of this work is to showcase the benefits of utilizing the mentioned techniques and open the doors for future developments for FL.

Besides the end-user perspective, FLOps is intended to be a foundational piece of software that can be easily modified and extended for developers and researchers. We put a lot of effort into writing high quality code, using state of the art libraries and frameworks. FLOps includes many development-friendly features. We enforce proper styling and typing via formatters and linters, including CI. Ready-made extendable multi-platform images and services automate development and evaluation workflows. These images, as well as the entire code, are made available on GitHub [7]. We also added base images with optional development flags to speed up the build and execution times of FLOps so that developers can verify and check their changes more rapidly.

On top of that we also implemented a new CLI tool for Oakestra and FLOps from the grounds up [15]. It is used to interact with Oakestra's and FLOps APIs. Besides that this configurable CLI tool also is capable of visualizing current processes in a human friendly way in real time as well as trigger evaluation runs and other automated tasks like installing necessary dependencies.

1.5 Thesis Structure

TODO

2 Background

As mentioned in the contributions section, FLOps combines and uses a large set of technologies from different disciplines. To properly understand FLOps as a whole and why it combines these techniques, it is necessary to analyze them individually. This enables us to form a common understanding, including critical background knowledge of their benefits and downsides. Only afterward does it make sense to discuss how FLOps merges them to create something new.

This background chapter provides a general overview for each sector and discusses aspects that are necessary for FLOps in greater detail. We start with exploring the field of federated learning. FL is the core task at hand that FLOps aims to optimize. A thorough understanding of this discipline is required to figure out where it has shortcomings. To improve upon these weaknesses, we study the established set of best practices from DevOps and MLOps. Techniques like automation and CI/CD require infrastructure and resources. Orchestration allows us to provision, manage, and deploy such infrastructure and resources. We review the field of orchestration technologies and provide a short overview of Oakestra [2] as the chosen platform. In the final background section, we take a look at and compare a couple of existing pieces of work that resemble FLOps.

2.1 Federated Learning

This section starts with the fundamental building blocks and terminologies of FL, followed by a section showcasing vital supplementary FL concepts. FLOps is orchestrated via Oakestra [2], which uses an unconventional three-tiered structure that allows support for geographical clusters. We have the opportunity to benefit from this unique composition and apply FL to it. To do so, we investigate more advanced concepts in FL, focusing on different FL architectures.

With this solid FL understanding, we review the research landscape of FL. We look at active and popular research directions and point out under-explored aspects, and weak points. We briefly look at FL in industry. Followed by a detailed comparison of existing FL frameworks and libraries. We conclude the section on FL by providing an overview of Flower [8], our FL framework of choice.

We base the majority of the first three subsections (FL basics, supplementary FL concepts, and FL architectures) on the 2022 book 'Federated Learning - A Comprehensive Overview of Methods and Applications' [12]. It captures and discusses the history and progress of FL research and state-of-the-art FL techniques (up to 2022).

2.1.1 FL Basics

Note that we assume the reader to be familiar with basic machine learning concepts.

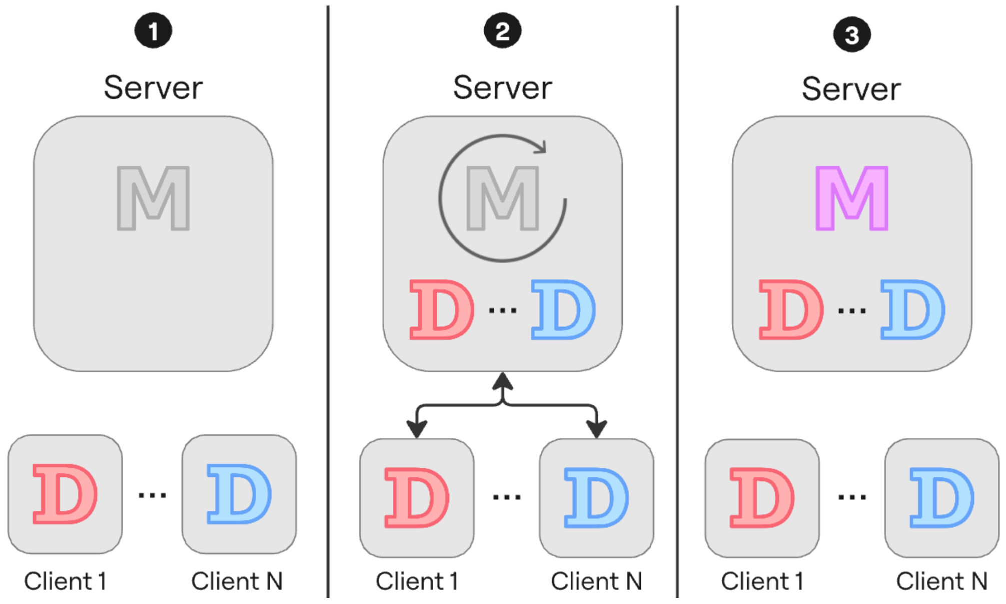


Figure 2.1: Centralized ML Model Training

Figure 2.1 depicts the classic centralized ML model training process. Starting from (1), where clients have their data (D) and the server hosts the untrained (gray) ML model (M). In (2), the clients send their data to the server. The server can now train the model using data from both clients. (3) depicts the final state after training. (The pink/purple model color represents that both data sources, red and blue, have been used during training.) The client data remains on the server and is exposed to potential exploitation.

As discussed in the introductory chapter, the centralized approach often leads to privacy breaches. FL was introduced to use this lucrative sensitive data on client devices for training ML models while keeping that data private and complying with laws and regulations. Many different algorithms and strategies exist for FL. We focus on the widely used base-case/classic FL algorithm FederatedAveraging (FedAvg) proposed as part of the original FL paper [13].

Figure 2.2 shows the basic FL training loop. Note that the number of learners can vary. This and the following figures mainly represent such groups only via two members, to optimize page space. The first differences are the component names. In FL, the server is frequently referred to as an **aggregator** and coordinates the FL processes. Clients are called **learners**. Note that using the terms server and clients in FL is still common. We prefer aggregators and learners because it highlights that these are FL components. This naming choice is also used in FLOps and helps with comprehension because FLOps uses a manifold of components, including non-FL servers and clients. Another difference is that all components must know and possess the ML model locally. They also need to set up their environment for training properly.

Initially, at (1), all models are untrained. At (2), the aggregator starts the first FL training cycle by telling the learners to start their local training. The local training rounds (epochs) are completed at (3). (The 'M's are now colored.) As a reminder, one can split up ML models into two parts. One part is (usually) a static lightweight model architecture that includes layer specification (in DNNs), training configuration, hyper-parameters like learning step sizes, and what loss type or activation function to use. Model weights and biases are the dynamic components of an ML model. A model without them is not useful because weights and biases are what get trained and allow the model to fulfill its intended use, such as prediction, inference, or generation tasks. These weights and biases are the major contributors to a trained model's overall size (space utilization). Because the model architecture is static in classic ML/FL, one can transmit the weights and biases between aggregators and learners instead of the entire trained model. We call everything that gets sent between the learners and aggregators (model) **parameters** and depict it with (P).

In (4), the learners have extracted their model parameters and sent them to the aggregator. The aggregator now has access to these parameters but not the sensitive data used to train them. That is how FL can profit from sensitive data while maintaining its privacy. Note that there are still attack vectors that allow exposing sensitive client information by abusing this parameter-based aggregation process. We briefly discuss this and other FL security aspects later on.

In (5), the server aggregates these collected parameters into new global parameters, which the aggregator applies to its model instance. This aggregation process is also called model fusion. Because learners can be heterogeneous and possess varying amounts of data, some learner updates might be more impactful than others. To respect this circumstance, learners typically also send the number of data samples, they used for training, to the aggregator. That way, the aggregator can prioritize its received updates proportionally. Otherwise, in classic FL aggregation, the mean of the parameters is used for the global model. The result is a **global model** that was trained

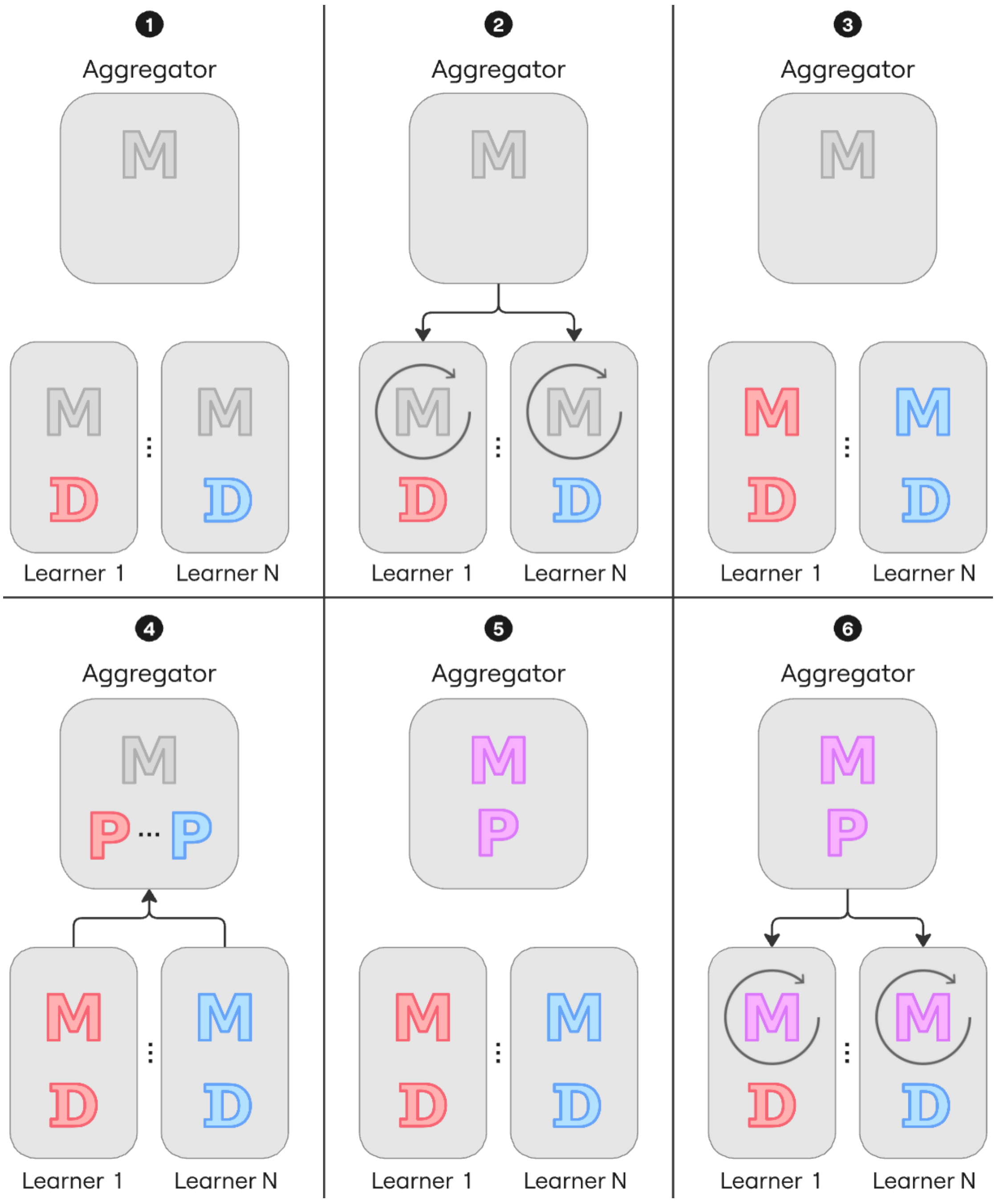


Figure 2.2: Basic Federated Learning

for one FL cycle.

In (6), the aggregator sends its global parameters back to the learners. The learners apply these parameters to their local model instance to make it identical to the aggregator's global model, thus discarding their locally trained parameters. Now, the FL training loop could terminate, and the learners or servers could use their global model copy for inference, or as depicted in (6), another FL training cycle begins. There can be arbitrarily many FL cycles, similar to conventional training rounds in classic ML. FL training will have to stop, due to time/resource constraints or reaching a satisfying performance. Otherwise, the accuracy and loss will worsen due to overfitting, assuming the available training data is finite and unchanging.

2.1.2 Supplementary FL Concepts

In this subsection, we explore essential supplementary FL concepts to get a better understanding of the field.

FL compared to Distributed Learning

At first glance, FL seems similar to Distributed Learning (DL). Both get used for computationally expensive large ML tasks. To increase convergence times and avoid needing one mighty machine, the computations get distributed among many weaker machines that train individually. Afterward, a global model gets aggregated at the server.

Regarding their differences, the quantity and distribution of training data can be very diverse in FL and might remain unknown throughout training. FL only uses the data that the learners offer. DL starts with full centralized access and control to the entirety of data, before splitting it up among its fixed and predefined clients. Thus, DL does not support the privacy concerns because it has total oversight and control of all data and how to split it up. In FL, the data might be IID or non-IID. Different learners can have varying amounts of data. The number of learners in FL can be very dynamic. Some devices might only join for a few training rounds or crash/fail/disconnect during training.

FL Variety

Most FL work is focused on end-user/edge/IoT devices. FL is not exclusive to these environments and can be used in conventional cloud environments.

As discussed in the first subsection, FL can train DNNs. One can also apply FL for classic ML models, such as linear models (logistic regression, classification, and more) or decision trees for explainable classifications. Plentiful FL optimizations, such as custom algorithms and strategies, exist for each mentioned ML variant.

FL can also support horizontal, vertical, and split learning. Horizontal learning can be helpful in scenarios where the available data features are the same but originate from different sources. One use case for horizontal learning is working with patient data from different hospitals that record the same features, such as age and ailment. Vertical learning is practical when different data samples have different feature spaces. In the hospital example, this would mean asking different doctors/experts about the same patients. The patient reports would be about the same individuals but include varying features, such as cardiological metrics or neurological metrics. We omit to discuss split learning due to its complexity that would bloat this thesis.

In case the global model is too general and does not satisfy a learner's individual needs, one can employ personalization. Different personalized FL (PFL) approaches exist. Some take the final trained global model and further train it on local data (fine-tuning). Other techniques train two local models concurrently. The first model gets shared and updated with the global parameters. The second one stays isolated and only gets influenced by local data. For inference a mixture between the global and purely local model can be used. PFL is a deep and growing subfield of FL.

FL Security & Privacy

Secure FL should use secure and authenticated communication channels to prevent messages from being intercepted, read, or impersonated by a man-in-the-middle adversary. To help with that, one should ensure that learners and aggregators are the only actors with access to those messages and can decipher them. There are two kinds of adversaries in FL. Insiders are part of the FL process, such as malicious aggregators or learners. Outsiders try to interfere from beyond the FL system.

A variety of FL threats exist. One example is manipulation, where insiders try to distort the model to their advantage by tinkering with FL components that the attacker can access. The attack goals include polluting the global model to misclassify (Backdoor). If the attack is untargeted (Byzantine), injecting random noise or flipping labels can degrade the model's performance. It is difficult to detect malicious activity because FL can support dynamic or even unknown numbers of learners that can use vastly different non-IID data. It can be unclear if the learner is innocent and simply has access to unusual data or if the learner is adversarial. Another example is if there are no safeguards in place during aggregation. A malicious learner can claim to have used an overwhelming amount of training samples, thus overshadowing other participants and influencing the global model the most. As a result, even very scarce, well-timed attacks in FL can have devastating impact.

Another threat comes from (model) inference, where insiders or outsiders try to extract sensitive information about the used training data. In classic FL, privacy leakage can only occur via inference. Inference attacks try to deduce private information from artifacts that the FL process produces. A large body of ML research exists that focuses on analyzing and protecting against such attacks. There are different subtypes of inference attacks. One example is the membership attack, which tries to find if specific samples were used for training. Another attack is called 'extraction attack', which tries to obtain all training samples. The challenge here is that attackers have easy access to the final model. Malicious insiders can even attack intermediate models. Model inversion attacks are different attack variants in which adversaries query the trained model in peculiar ways to reverse engineer data samples. If the attacker is repeatedly successful, it is possible to deduce the original dataset. Other attacks require malicious aggregators that can trace back the update parameters that the learner provided before aggregating the global parameters.

Fortunately, there exists a growing array of defenses against those threats. It is crucial to pick and combine these defenses wisely based on the use case and environment. One major technique is differential privacy (DP). DP is a complex mathematical framework that is formally proven to work. One can use DP as noise for the dataset or (inference) query. The downside is that DP might reduce the model accuracy significantly.

Secure aggregation is a prominent protection against model inversion attacks. It securely combines individual model parameters into global ones before sending them to the aggregator, which makes re-engineering and backtracking much harder. [9]

2.1.3 FL Architectures

FL comes in two broad structural categories. Cross-silo or enterprise FL gets used for example in large data-centers or multinational companies. Each learner represent a single large institution or participating group. There are only around ten to a few dozen learners involved. The identity of the parties are considered for training and verification. In general, every individual local update from every learner at every training round is significant. Fallouts and failures of individual learners are serious.

Cross-device FL can include hundreds or millions of devices, primarily edge/IoT devices. One can say that cross-device is the opposite of cross-silo. Due to this large pool of learners, typically one a subset of them get used per training round. The identities of the participating learners are usually unimportant and get ignored. Due to the nature of these devices and their environments, cross-device FL needs to manage challenges, such as non-IID data, heterogenous device-hardware, different network conditions, learner outages, or stragglers. Various techniques exist to navigate these challenging conditions, including specialized algorithms for aggregation or learner selection. These strategies can take bias, availability, resources, and battery-life into account. FLOps focuses on cross-device FL. From now on, when we mention FL, we mean cross-device FL.

As discussed FLOps wants to benefit from the unique three-tiered Oakestra [2] architecture. Different FL architectures exist to support such large-scale FL environments. The two main challenges for such scenarios are how o manage huge number of connections and aggregations, and how to reduce the negative impact of straggling learner updates. The problem with using a single aggregator, as seen in 2.2, is that this single aggregator becomes a communication bottleneck. Additionally, per-round training latency is limited by the slowest participating learner, thus stragglers turn into another bottleneck.

We discuss four main architectures for large-scale FL.

Clustered FL

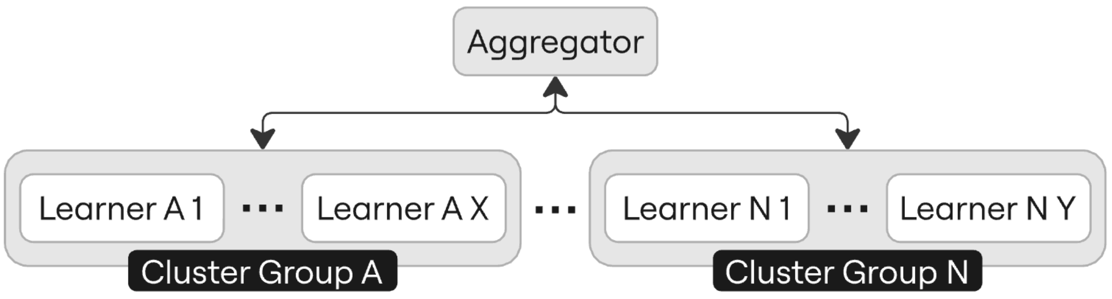


Figure 2.3: Clustered FL Architecture

Figure 2.3 shows the Clustered FL (CFL) architecture, where similar learners are grouped together. This clustering can be based on local data distribution, training latency, available hardware, or geographical location. The issue of the singular aggregator as a bottleneck persists.

The main challenge for CFL is choosing a fitting clustering criteria and strategy for the concrete use case. If the criteria is very biased the risk arises to heavily favor updates from preferred clusters, thus resulting in a biased global model with bad generalization. Another task is to properly profile the nodes so that they are matched to the correct cluster. For example if a slow outlier is present in a cluster the entire cluster suffers. Node properties can very over time, so cluster membership has to be dynamic. One should not overdue profiling otherwise privacy might get in danger.

The benefits of CFL are its ease of implementation, familiar architecture to classic FL, flexibility to tune clustering/selection dynamically, and that CFL can be combined with other architectures.

The downsides of CFL are that a proper clustering strategy is use-case dependent and challenging to optimize. CFL does not really solve scalability issues on its own, especially that with larger numbers of nodes the clustering overhead becomes critical.

Hierarchical FL

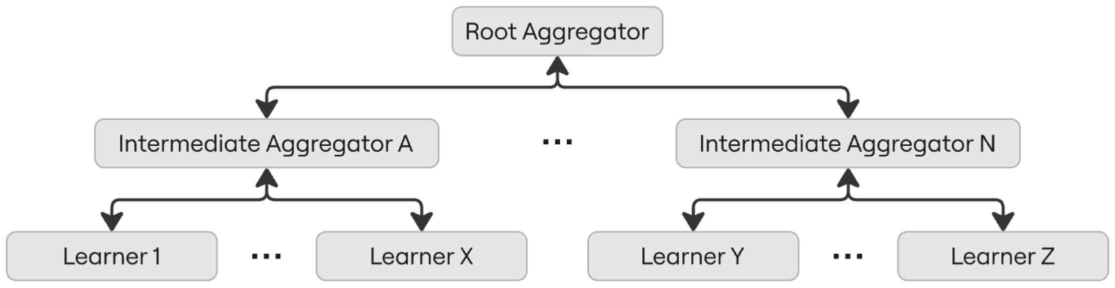


Figure 2.4: Hierarchical FL Architecture

Figure 2.4 depicts the hierarchical FL (HFL) architecture. In HFL the root aggregator delegates and distributes the aggregation task to intermediate aggregators. Note that HFL can have multiple layers of intermediate aggregators. Each intermediate aggregator and its connected learners resemble an instance of classic FL. After aggregating an intermediate model the intermediate aggregators send their parameters upstream to the root aggregator. The root combines the intermediate parameters into global ones and sends them downstream for further FL rounds. This structure requires significant modifications to the underlying FL architecture. The proper design and implementation as well as assignment of learners to aggregators determine the success of one's FL setup. For example if too many learners are attached to a given aggregator, that aggregator becomes a bottleneck. If too few learners are assigned that the intermediate aggregated model can get very biased and the infrastructure resource and management costs become unjustified for the little number of learners. With more components a management overhead arises, including handling fault-tolerance, monitoring, synchronizing, and balancing. Bad synchronization can amplify straggler problems. Balancing refers to combining and harmonizing intermediate parameters to get a good global model.

The benefits of HFL are its dynamic scalability and load balancing. One can easily add or remove intermediate aggregators and their connected learners. Due to this distribution of load and aggregation each individual aggregator including the root, is less likely to face bottleneck issues.

HFL can be combined with CFL, where each intermediate aggregator is responsible for one or multiple clusters.

The downsides of HFL are communication and management overheads. More components lead to more messages being transmitted. These messages all need to be secured and encrypted. With more components and nodes more possible backdoors exist for adversaries to take advantage of.

Decentralized FL

Decentralized FL does not require a central aggregator, instead it operates on a peer-to-peer basis via a blockchain. That way the centralized communication bottleneck gets resolved. The blockchain represents the global model. Learners train in parallel. Each locally trained update gets a version. Based on this version random clients are chosen for aggregation. The results gets appended to the blockchain, and the model version incremented. FLOps does not use this kind of FL so we keep this part short.

Asynchronous FL

This architecture allows learners to train all the time and push their updates to the aggregator once they are finished. This method eliminates stragglers and dropout problems, because a training round does not need to wait or handle for any outliers and timeouts. The new issue of staleness arises, where updates get merged into the global model that took a very long time to complete. Such an update used a now outdated version of the global model. As a result the global model gets partially reverted to an older state. Asynchronous FL can be combined with other architectures.

2.1.4 FL Research

2.1.5 FL in Industry

2.1.6 FL Frameworks & Libraries

2.1.7 Flower

2.2 Machine Learning Operations

2.2.1 DevOps

2.2.2 MLOps

2.2.3 MLflow

2.3 Orchestration

2.3.1 Fundamentals

2.3.2 ML Containerization & Orchestration

2.3.3 Oakestra

2.4 Related Work

3 Requirements Analysis

3.1 Overview

3.2 Proposed System

3.2.1 Functional Requirements

3.2.2 Nonfunctional Requirements

3.3 System Models

3.3.1 Scenarios

3.3.2 Use Case Model

3.3.3 Analysis Object Model

3.3.4 Dynamic Model

4 System Design

5 Object Design

6 Evaluation

6.1 Rationale

6.1.1 Chosen Experiments

6.2 Experimental Setup

6.2.1 Monolith

6.2.2 Multi-Cluster

6.2.3 Evaluation Procedure

6.3 Results

6.3.1 Basics

6.3.2 Image Builder

6.3.3 Different ML Frameworks/Libraries & Datasets

6.3.4 Multi-cluster & HFL

7 Conclusion

7.1 Limitations & Future Work

7.1.1 Federated Learning via FLOps

7.1.2 Complementary Components & Integrations

8 DELME tum example

8.1 Section

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}, \ac{TUM}` $\Rightarrow$ **TUM!** (**TUM!**), **TUM!**

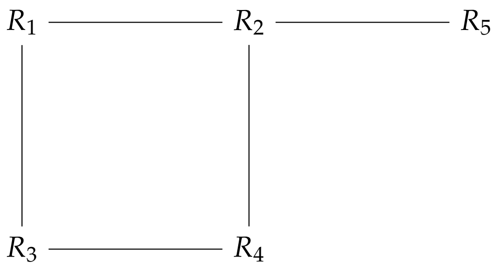
For more details, see the documentation of the `acronym` package[1].

8.1.1 Subsection

See Table 8.1, Figure 8.1, Figure 8.2, Figure 8.3.

Table 8.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3



$$\begin{tikzcd} R_1 \arrow[r, dash] \arrow[d, dash] & R_2 \arrow[r, dash] \arrow[d, dash] & R_5 \\ R_3 \arrow[r, dash] & R_4 & \end{tikzcd}$$

Figure 8.1: An example for a simple drawing.

[1] `https://ctan.org/pkg/acronym`

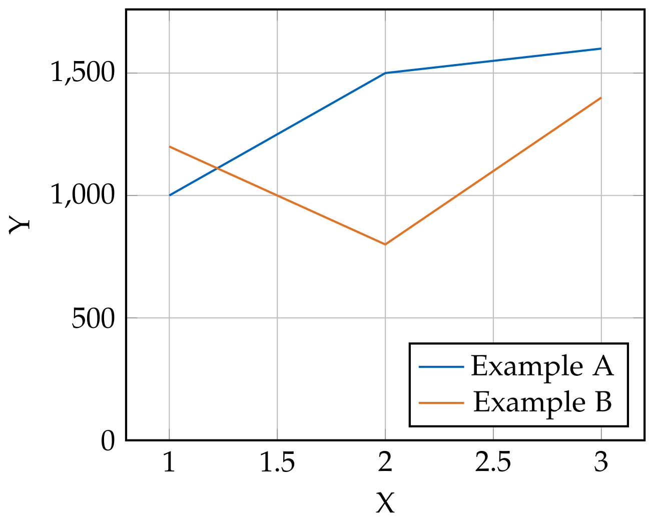


Figure 8.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 8.3: An example for a source code listing.

List of Figures

List of Tables

Bibliography

[1] *Anaconda Documentation*. Accessed: 2024-08-12. URL: https://docs.anaconda.com/.

[2] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9.

[3] A. Bourechak, O. Zedadra, M. N. Kouahla, A. Guerrieri, H. Seridi, and G. Fortino. "At the Confluence of Artificial Intelligence and Edge Computing in IoT-Based Applications: A Review and New Perspectives." In: *Sensors* 23.3 (2023). ISSN: 1424-8220. DOI: 10.3390/s23031639.

[4] *Buildah Homepage*. Accessed: 2024-08-12. URL: https://buildah.io/.

[5] *containerd Documentation*. Accessed: 2024-08-12. URL: https://containerd.io/docs/.

[6] *Docker Documentation*. Accessed: 2024-08-12. URL: https://docs.docker.com/.

[7] *FLOps Code Repository*. Accessed: 2024-08-12. URL: https://github.com/oakestra/addon-FLOps.

[8] *Flower Documentation*. Accessed: 2024-08-12. URL: https://flower.ai/docs/.

[9] J. Kim, G. Park, M. Kim, and S. Park. "Cluster-Based Secure Aggregation for Federated Learning." In: *Electronics* 12.4 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12040870.

[10] *Kubernetes Documentation*. Accessed: 2024-08-12. URL: https://kubernetes.io/docs/home/.

[11] C. Legislature. *California Consumer Privacy Act (CCPA)*. Online; accessed August 11, 2024. 2018.

[12] H. Ludwig and N. Baracaldo, eds. *Federated Learning - A Comprehensive Overview of Methods and Applications*. Springer, 2022. ISBN: 978-3-030-96896-0. DOI: 10.1007/978-3-030-96896-0.

[13] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas. “Communication-Efficient Learning of Deep Networks from Decentralized Data.” In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by A. Singh and J. Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 2017, pp. 1273–1282.

[14] *MLflow Documentation*. Accessed: 2024-08-12. URL: `https://www.mlflow.org/docs/latest/index.html#`.

[15] *Oakestra & FLOps CLI Code Repository*. Accessed: 2024-08-12. URL: `https://github.com/oakestra/oakestra-cli`.

[16] T. E. Parliament and Council. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. Online; accessed August 11, 2024. 2016.