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

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2024 Alexander Malyuk

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
DP Differential Privacy

Common Acronyms :

AI Artificial Intelligence
ML Machine Learning
DL Deep Learning
DNN Deep Neural Network
LLM Large Language Model
API Application Programming Interface
GUI Graphical User Interface
SLA Service-Level Agreement
CLI Command-Line Interface
IoT Internet of Things
P2P Peer-to-peer

1. Introduction

In the last several years, the number of smart devices has been rapidly growing and generating enormous amounts of data (BigData). Improvements in connectivity (Cloud Computing & Internet of Things—IoT), connection speeds (5G), and computing power enable this development. 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 to Industry 4.0 and beyond. Examples include agriculture, healthcare, education, and the security sector [8]. Diverse and complex challenges arise from this swiftly evolving landscape.

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.

Governments and organizations have established laws and regulations to prohibit potential abuse of sensitive data. 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. Examples include the European Parliament regulation to protect personal data [55] or the California Consumer Privacy Act (CCPA) [40]. 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 [44]. In FL, instead of collecting the data on a server and training ML models centralized, 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.

Most researchers working in the field of FL focus on enhancing existing FL components, strategies, and algorithms or developing novel ways of doing FL. There is a noticeable scarcity of work that concentrates on the crucial aspects of the initial setup, deployment, and usability of FL. Because FL is a relatively modern technique, it lacks a sophisticated production-grade ecosystem with 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. This is due to the lack of documented steps regarding setup, deployment, management, and execution. Instead of using a shared set of bootstrapping tools to make progress on novel work more efficiently, one needs to set up and manage FL from the ground up. A small set of emerging libraries and frameworks does exist 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 the lack of more advanced techniques to increase productivity that other domains have already been using for several years, such as modern DevOps practices.

1.2. Motivation

Building or contributing to an FL framework or library focusing on the previously mentioned challenges could soften or entirely alleviate those problems. Such a tool should have Docker and Kubernetes as role models and strive 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. These areas range from ML/FL, dependency management, containerization technologies, and orchestration to automation. 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, this envisioned tool could become a significant part of the emerging FL ecosystem. This tool would empower less experienced individuals to participate and contribute to the field of FL. As a result, the entire discipline of FL could improve from these utilized techniques, and more people in more areas could access and benefit from FL.

1.3. Objectives

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

Improve Accessibility
Making FL more accessible by abstracting away and automating complexities enables further 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.

Benefit from Automation
Automating tedious, error-prone, and repetitive manual tasks necessary to perform FL will save time and resources for critical work. Doing more crucial work in less time allows for further advancements in the discipline of FL.

Prioritize Practical FL Application
This tool should focus on being usable in real physical conditions on distributed devices. FL struggles with a gap between research/virtual-simulation and practical application in real production environments. It should be feasible to incorporate this tool into existing workflows.

Embrace Flexibility
Because FL is such a young and active field, it faces constant change. This tool should welcome change in the form of extendability and adaptability. It should be flexible and applicable to a multitude of use cases and scenarios. This tool should be easy to modify to accommodate evolving needs. It should profit from existing technologies to offer a higher level of quality than creating everything from the grounds up.

1.4. Contribution

This thesis proposes a novel solution called FLOps to fulfill the objectives above. It enables individuals to use, develop, and evaluate practical FL. FLOps enriches FL with modern best practices from automation, DevOps/MLOps, and orchestration. The term FLOPS is known as a measurement unit for computer performance (floating point operations per second). **FLOps** means something different and has not been used or applied in the context of FL. However, **MLOps** has been used to describe DevOps techniques for ML. The name FLOps takes inspiration from that. This thesis is intended to be a foundational work to help establish FLOps as a discipline. It is also the name of this thesis' standalone software solution. The work aims to showcase the benefits of utilizing the mentioned techniques and open the doors for future developments for FL. FLOps improves accessibility by enabling users without experience in FL, MLOps, or orchestration to do FL and still benefit from these technologies.

FLOps streamlines FL processes and saves time. To do FL, users simply provide a link to their ML git repository. This repository code needs to satisfy some simple structural prerequisites. It 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 or containerd [14]. This approach eliminates the need for tedious device setup and the struggle to configure heterogeneous dependencies to match the training requirements. FLOps automatically performs FL training based on the user-requested configuration. Users can 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. As a result, FLOps helps users at every step of their FL journey.

Diverse technologies from various disciplines are necessary for FLOps to provide its services. Instead of reimplementing complex features in a subpar way from scratch, FLOps benefits from combining and extending existing solutions and technologies in unique and novel ways. This includes using Anaconda [3] and Buildah [9] to manage dependencies and build images. FLOps utilizes a pioneering FL framework called Flower [19] to execute its FL training loops. The mentioned runtime observability features are available via a mature MLOps tool called MLflow [46]. Because FL pushes model training to client devices, especially edge devices, FLOps uses an orchestrator native to the edge environment. With the help of Oakestra [5], FLOps can deploy and orchestrate its components. FLOps has been implemented as a separate addon for Oakestra. Because they interact via 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. For example, FLOps supports hierarchical FL (HFL), which Flower does not directly support or offer. To the best of our knowledge, FLOps is the first work that combines Flower with MLflow, and allows HFL, and automatically converts ML code into FL-enabled containerized images. In conclusion, FLOps combines these tools in novel ways to guarantee a high level of quality and to achieve its objectives.

Besides the end-user perspective, FLOps aims to be a foundational piece of software that can be easily modified and extended by developers and researchers. FLOps was implemented with the latest best practices and industry standards in mind. Its code strives to be of high quality and great readability. It uses state-of-the-art libraries and frameworks. FLOps includes many development-friendly features. It enforces 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 openly accessible on GitHub [16]. FLOps includes additional base images with optional development flags to speed up the build and execution times. Therefore, 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 ground up [52]. It interacts with Oakestra's and FLOps' APIs. This configurable CLI tool is also capable of visualizing current processes in a human-friendly way in real-time. Additionally, the CLI can trigger evaluation runs and other automated tasks, such as installing necessary dependencies. These additional efforts should enable FLOps to meet custom and future demands.

1.5. Thesis Structure

TODO

2. Background

To properly understand FLOps as a whole and why it combines different techniques, it is necessary to analyze them individually. This analysis includes 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 of each sector and discusses aspects necessary for FLOps in greater detail. The first section explores 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 determine its shortcomings. The following section discusses established best practices from DevOps and MLOps to improve upon these weaknesses. Techniques like automation and CI/CD require infrastructure and resources. Orchestration enables provisioning, management, and deployment of such infrastructure and resources. Its section reviews orchestration technologies and provides a short overview of Oakestra as the chosen platform for FLOps. In the final background section, a couple of existing works resembling FLOps are examined and compared.

2.1. Federated Learning

This section contains necessary background information and context regarding FL. The first subsection covers fundamental FL building blocks and terminologies. The next subsection explains vital supplementary FL concepts. Oakestra orchestrates FLOps. It uses an unconventional three-tiered structure that allows support for geographical clusters [5]. This structure opens up unique opportunities for FL applications. More advanced FL architectures are necessary to benefit from these opportunities. The following subsection discusses these architectures. These three subsections build a solid FL understanding. A great summary and source of information for deeper insights into the field of FL can be found in [43]. It is a 2022 book that captures and discusses the history and progress of FL research and state-of-the-art FL techniques.

Building on top of established FL understanding, the subsequent subsection reviews the research landscape of FL. That subsection showcases active and popular research directions. It points out underexplored aspects and weak points in the field. The penultimate subsection analyzes and compares existing FL frameworks and libraries. The concluding subsection provides an overview of the FL framework FLOps uses.

2.1.1. FL Basics

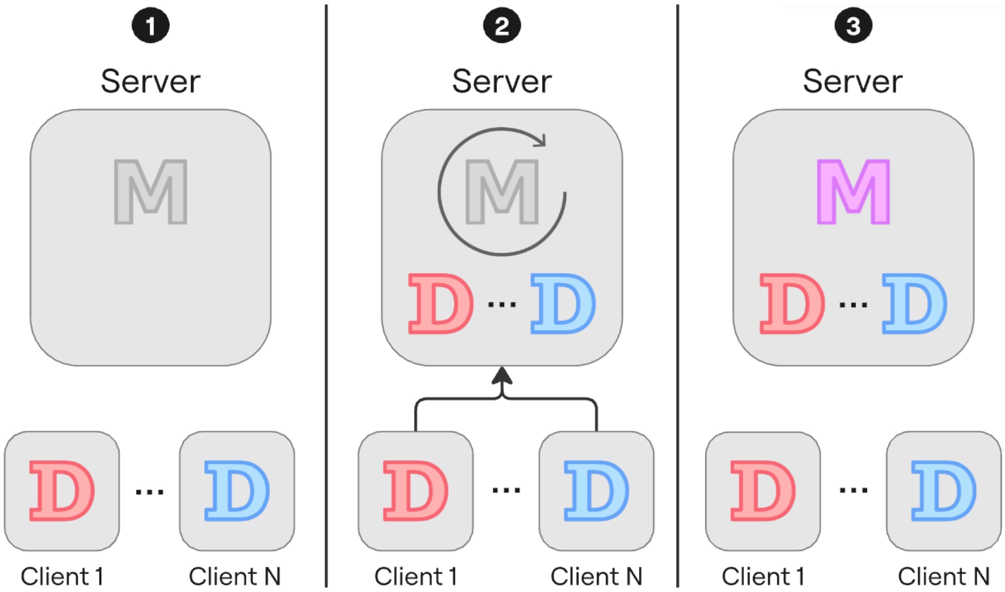

Figure 2.1.: Centralized ML Model Training

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 the clients. (3) depicts the final state after training. (The pink/purple model color symbolizes that different data sources have been used during training.) The client data remains on the server and is exposed to potential exploitation. As discussed in the introductory chapter, this centralized approach often leads to privacy breaches.

FL was introduced to use lucrative sensitive data on client devices for training ML models while keeping that data private. Thus, FL complies with laws and regulations. Many different algorithms and strategies exist for FL. The following example focuses on the widely used base-case/classic FL algorithm FederatedAveraging (FedAvg). It was proposed in the original FL paper [44].

Figure 2.2 shows the basic FL training loop. The number of learners can vary. The first differences are the component names. In FL, the server is frequently called an

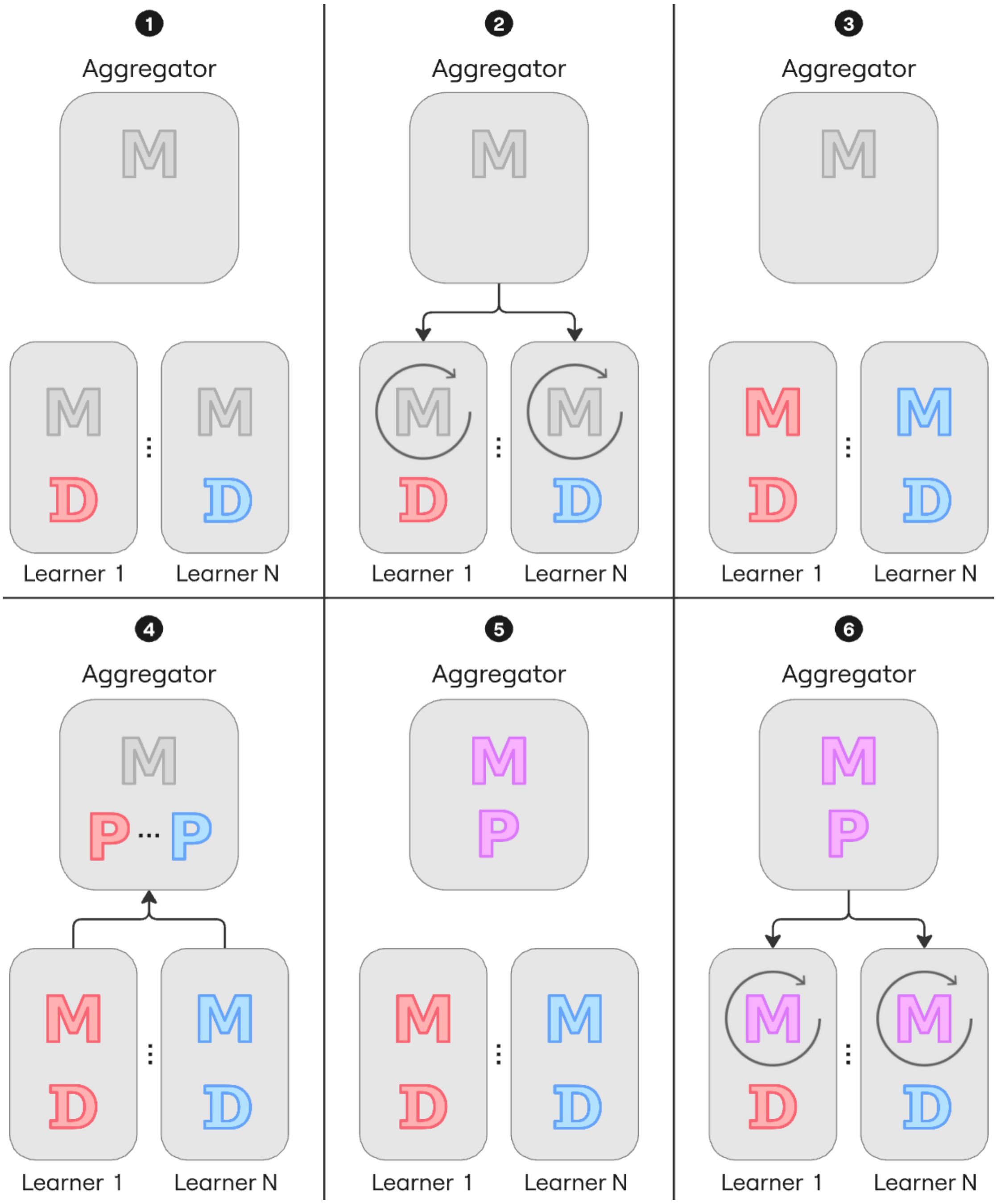

Figure 2.2.: Basic Federated Learning

aggregator, and it coordinates the FL processes. Clients are called **learners**. Using the terms server and clients in FL is still common. This work prefers aggregators and learners because it highlights that these are FL components. This naming choice is also used in FLOps and helps with comprehension. FLOps uses various 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.

As a reminder, one can split up ML models into two parts. One part is (usually) a static lightweight model architecture. It includes layer specification (in DNNs), training configuration, and hyperparameters like learning step sizes, loss, and activation functions. 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. They 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). Model architecture is static in classic ML/FL. Thus, FL components can transmit and share weights and biases instead of the entire trained model. This work calls model relevant data sent between the learners and aggregators (model) **parameters** and depicts it with (P).

The concrete classic FL steps are as follows. 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.) 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. Possible attack vectors still exist. They expose sensitive client information by abusing this parameter-based aggregation process.

In (5), the server aggregates these collected parameters into new global parameters. This aggregation process is also called model fusion [43]. The aggregator applies these global parameters to its model instance. Learners can be heterogeneous and possess varying amounts of data. Therefore, 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 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. By doing this, learners discard their locally trained parameters. The FL training loop could terminate, and the learners or servers could use their global model copy for inference. Otherwise, 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 eventually terminates due to time/resource constraints or a failure to reach a satisfying performance. If not terminated, 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. [37]

2.1.3. FL Architectures

FL comes in two broad structural categories. Cross-silo or enterprise FL gets used in large data centers or multinational companies. Each learner represents a single institution or participating group. There are only around ten to a few dozen learners involved. Cross-silo FL considers the identity of the parties for training and verification. Generally, 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 great pool of learners, a subset typically gets 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, heterogeneous 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 consider bias, availability, resources, and battery life. 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 [5] architecture. Different FL architectures exist to support such large-scale FL environments. The two main challenges for such scenarios are managing a massive number of connections and aggregations and reducing 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

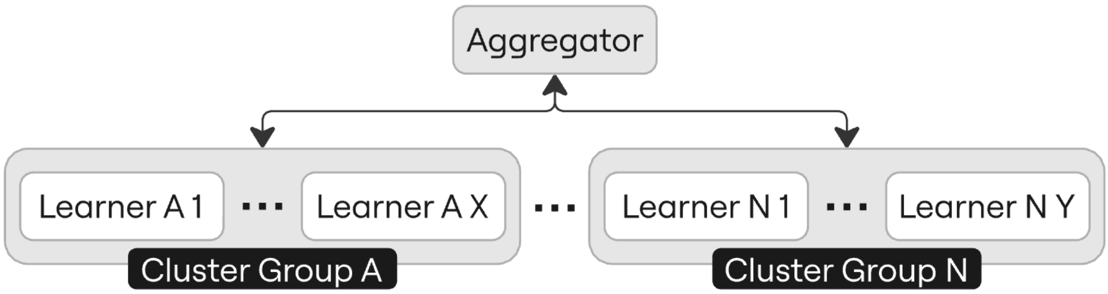

Figure 2.3.: Clustered FL Architecture

Figure 2.3 shows the Clustered FL (CFL) architecture that groups similar learners into clusters. CFL can base clusters 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 suitable clustering strategy and criteria for the concrete use case. If the criteria are very biased, the risk arises that updates from preferred clusters will be heavily favored, resulting in a biased global model with bad generalization. Another task is to properly profile the nodes to match them to the correct cluster. For example, the entire cluster suffers if a slow outlier is present in a cluster. Node properties can vary over time, so cluster membership has to be dynamic. One should not overdo profiling. Otherwise, privacy might get compromised.

The benefits of CFL are its ease of implementation, familiar architecture to classic FL, and flexibility to tune clustering/selection dynamically. One can combine CFL with other architectures. A downside of CFL is that a proper clustering strategy is use-case-dependent and challenging to optimize. CFL does not really solve scalability issues on its own, especially since the clustering overhead becomes critical with larger numbers of nodes.

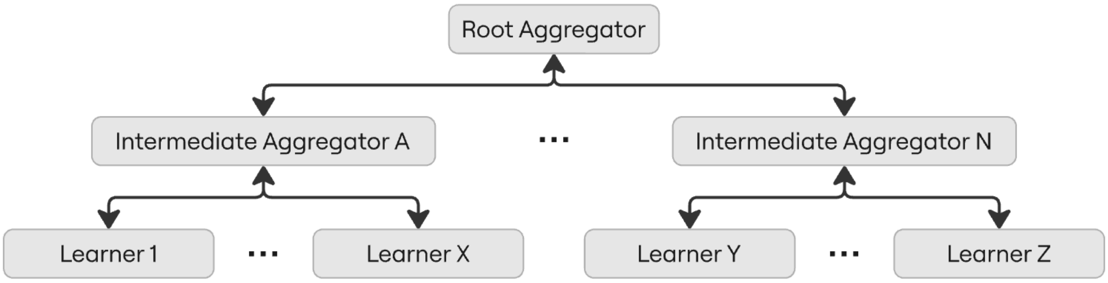

Figure 2.4.: Hierarchical FL Architecture

Hierarchical FL

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 the 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, the intermediate aggregated model can get very biased, and the infrastructure resource and management costs become unjustified for the small number of learners. A management overhead arises with more components, 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 aggregator, including the root, is less likely to face bottleneck issues. One can combine HFL 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 transmitted messages. These messages all need to be secured and encrypted. With more components and nodes, adversaries can take advantage of more possible backdoors.

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 get appended to the blockchain, and the model version is incremented. FLOps does not use this kind of FL, so we keep this part short.

Asynchronous FL

This architecture allows learners to train continuously 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 outliers and timeouts. A new issue of staleness arises when updates are 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 is partially reverted to an older state. Asynchronous FL can be combined with other architectures.

2.1.4. FL Research

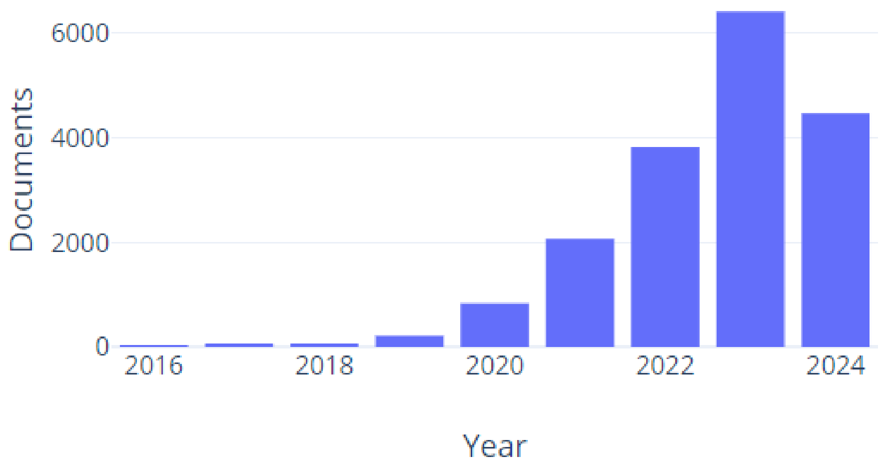

Figure 2.5.: Evolution of FL Publications

Figure 2.5 shows the exponential growth of FL documents since 2016. (This data comes from searching for "federated learning" in article title, abstract, or keywords via Scopus [62].) Note that we based the idea for this graph on [60], and we used a different query with the latest available data.

Before we started working on FLOps, we wanted to find research gaps in the fields of ML at the edge, specifically FL. In total, we have read and examined 47 papers in detail, with 26 papers focusing on FL. Additionally, we consulted several articles, joined and participated in discussion forums, and completed a couple of paid courses [64]. Discussing each paper in detail would heavily bloat this thesis. We present key and meta-findings instead. During our reading, we created and incrementally updated a database in which we noted the specific properties of each paper. These properties include one or multiple categories in which the paper fits in, the initial problems or challenges the authors tried to resolve, their contributions, results, limitations, and envisioned future work. We also noted down what ML or FL frameworks or libraries they used. We based these properties on our subjective analysis instead of extracting them verbatim from the paper.

Table 2.1 depicts a subset of the FL papers we analyzed. It shows the documented contributions, limitations, and future work properties. We explicitly decided to use an abbreviated format instead of verbose sentences to optimize the limited space. One can inspect the remaining FL papers in the appendix A. These tables should provide a good impression of the individual papers we examined. We look for patterns and trends to better understand the research field of FL as a whole. We utilize the documented properties for this.

Figure 2.6 shows the different found categories and their distribution. Most of our papers were focused on performance, trying new concepts, finding best practices, and exploring different FL architectures. Only two papers focused on deployment and orchestration. A similar trend can be seen in figure 2.7. The primary focus is on investigating new concepts or improving existing bottlenecks in terms of performance, scalability, and complexity. We point out that several papers aimed to narrow the gap between industry and research or to make FL easier to use. This ease of use seems to focus on improving already configured and working FL setups.

The main contributions seen in figure 2.8 strengthen this assumption. This chart is dominated by mathematical and conceptual proofs that novel architectures and algorithms work as proposed. Contributions do not seem to focus on improving the initial setup, deployment, and configuration processes. The results achieved mirror these finding. Figure 2.9 shows that these contributions lead to better efficiencies in terms of speed, resource utilization, training results, and handling of heterogeneous data. Note that we based these properties on the results and contributions the authors mentioned themselves and on our conclusions.

Figure 2.10 reflects our perception. If specified, the focal point is on improving privacy and security, further performance optimizations, or adding support for more

ID	Contributions	Limitations & Future Work
[1]	A novel selection and staleness-aware aggregation strategy. Analysis of resource wastage and the impact of stragglers. A smart participation selection based on learner availability.	Privacy or security were not considered. Evaluations are based on classic datasets (MNIST, CIFAR-10), which do not reflect real non-IID data. Only homogeneous resources were assumed. Use of a simple linear regression model for availability prediction. More sophisticated alternatives exist. Factors such as battery level, bandwidth, and user preferences should also be considered for availability prediction.
[38]	A novel cluster-based secure aggregation strategy for diverse nodes. Clustering based on processing score & GPS information/latency leads to better throughput and reduces false-positive dropouts. A new additive sharing-based masking scheme that is robust against dropouts.	All participants are assumed to be honest. Malicious users were not considered. The aggregator might become a bottleneck, which can be resolved via HFL (with cluster heads). Image classification was the only evaluated ML task.
[70]	An FL caching scheme including novel algorithms and architecture. Utilization of an AI training model that considers user history.	A convergence analysis was not provided. For further security and privacy improvements, blockchain-empowered FL should be investigated.
[51]	Analysis of the impact of pre-training ML models for FL initialization compared to the common random approach. Findings show pre-trained model superiority.	It is challenging to get a pre-trained model if the necessary data is not available or private. Using pre-trained models can lead to biases. Only a specific (warm-start) initialization strategy was considered.
[41]	A novel incentive/resource-based allocation schema that utilizes game theory. Learners with more data are more valuable and they can compete for higher participation rewards. Multiple model owners compete for cluster heads with the most data.	The effects of social networks and their impact on worker's cluster selection decisions should be researched. Malicious workers were not considered.
[12]	Synergy of asynchronous and synchronous FL via asynchronous tiers, which is able to handle stragglers.	The tiers all update the server individually. Further improvements are possible via HFL with intermediate cluster heads to do the aggregation. Additional security could be applied at these cluster heads.

Table 2.1.: A Subset of the FL Papers considered for FLOps

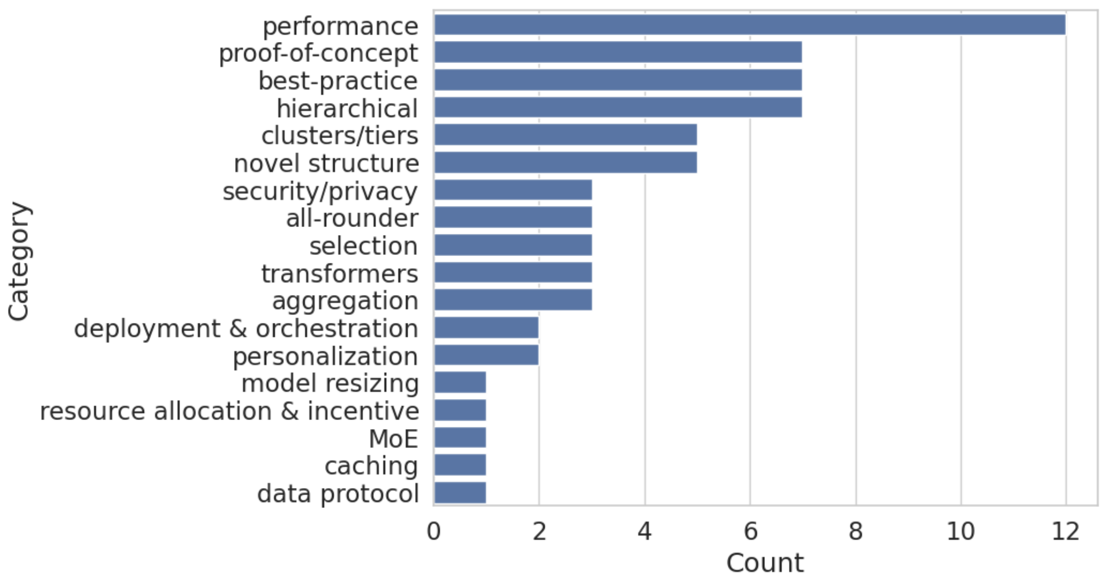

Figure 2.6.: FL Paper Categories

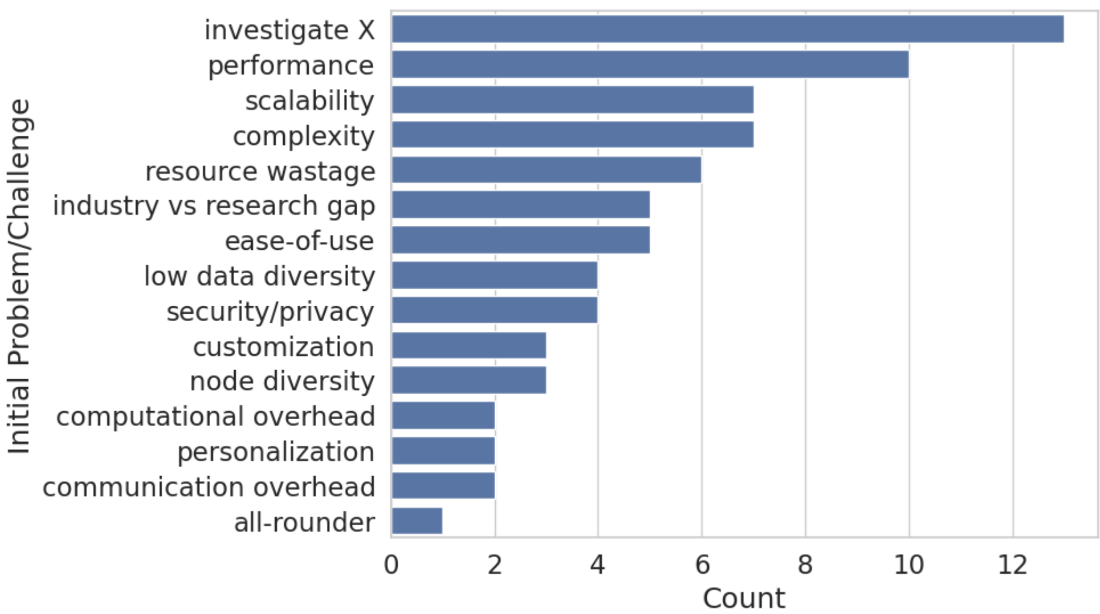

Figure 2.7.: Targeted Problems & Challenges of FL Papers

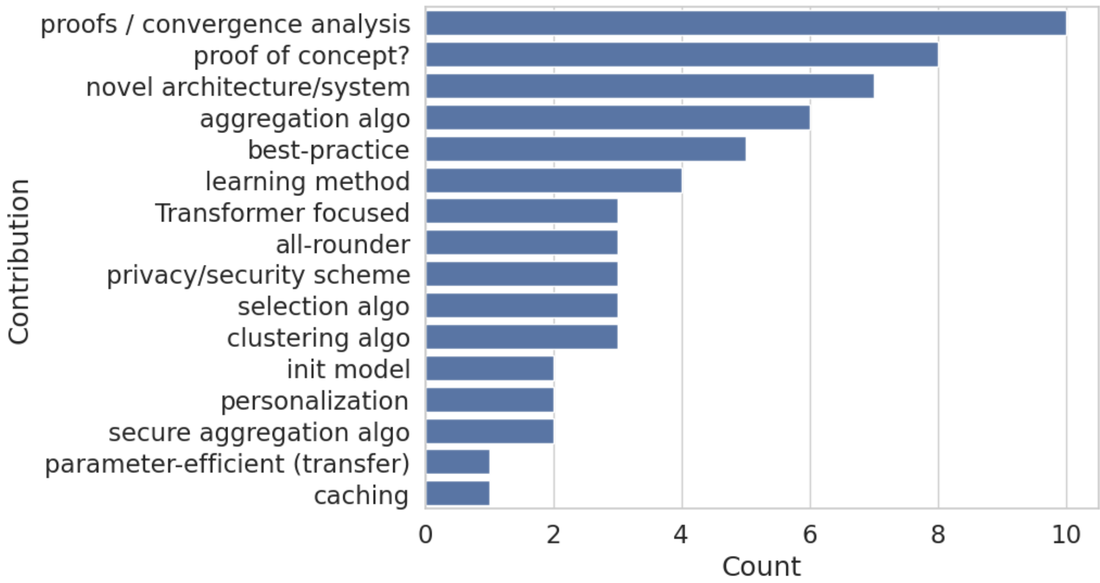

Figure 2.8.: FL Paper Contributions

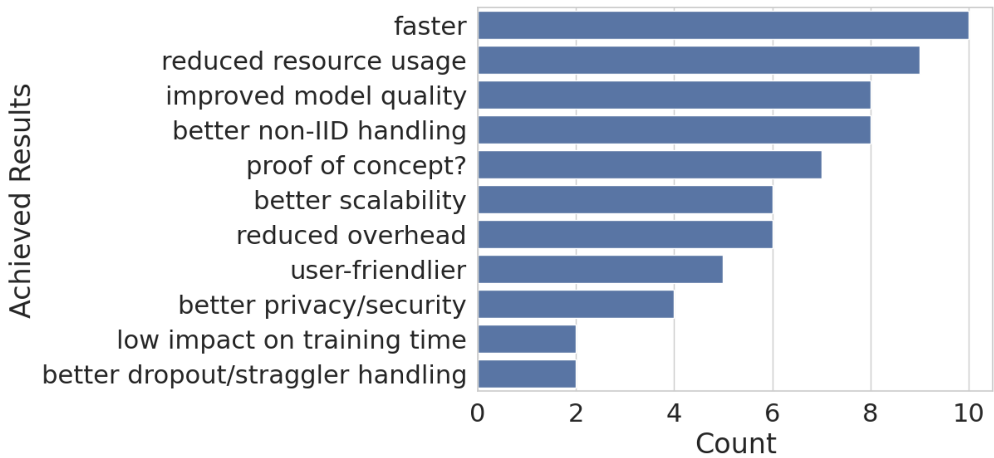

Figure 2.9.: Achieved Results of FL Papers

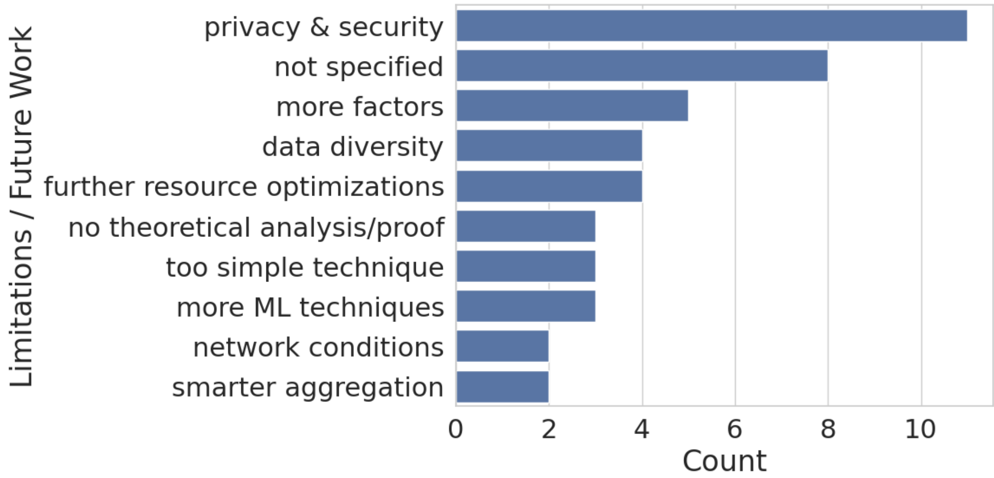

Figure 2.10.: Limitations & Future Work of FL Papers

ML use cases. Even the future focus is not on optimizing accessibility, usability, or the mentioned initial vital steps.

Because we assigned these properties subjectively and our paper sample size is relatively small, we compare our findings so far with the total number of published works about FL. We use the same method to gather the data as for 2.5. Figure 2.11 shows how many works have been published in FL with specific keywords that match our custom categories. The global results paint a similar picture as our samples. The most popular topics in FL are related to privacy/security, performance, or algorithms. Only a tiny portion of FL papers focus on usability, automation, orchestration, or other initial steps.

It seems that researchers assume others to already have working FL environments and motivate their readers to optimize them based on their findings instead of replicating and configuring such an FL setup initially. One can also see these tendencies when inspecting the ML and FL frameworks and libraries the authors mentioned they used in our examined papers. Figure 2.12 shows that most authors did not explicitly state what ML framework or library they used for their work. Many researchers used Pytorch and TensorFlow. Figure 2.13 shows that FL researchers rarely mention what FL frameworks they use for their work. It is much more common for authors to mention what ML framework they used than what FL framework they used.

Possible reasons for this might be that ML as a field is a lot older, more sophisticated, widespread, and established. The same applies to ML frameworks. On the other hand, FL is a very young subfield of ML research. FL frameworks are still in their early stages. Therefore, FL researchers might be using FL frameworks, but due to the framework's

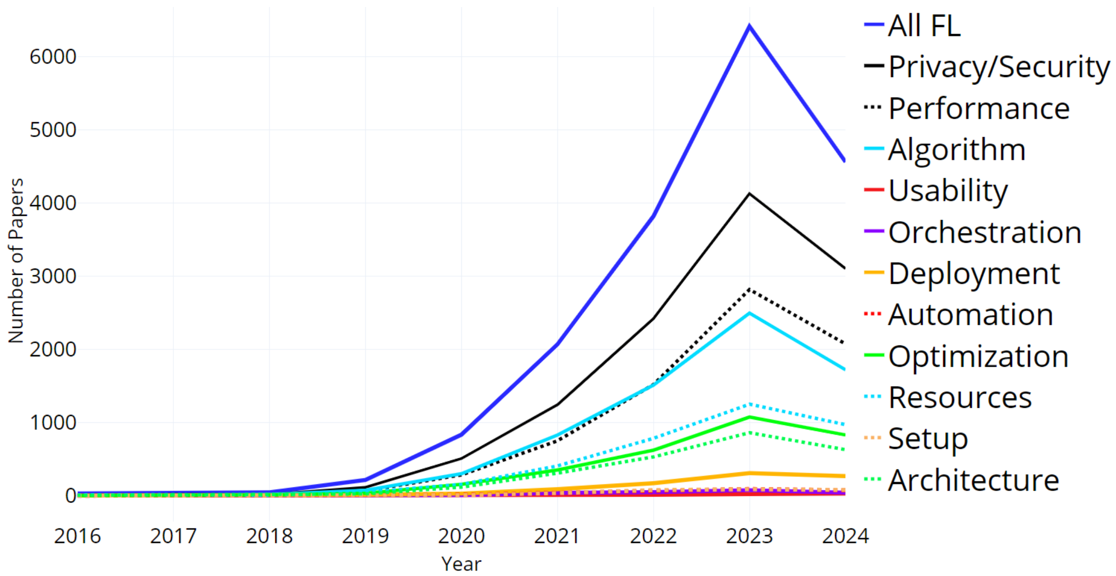

Figure 2.11.: Evolution of FL Publications based on Keywords

immaturity, the researchers might not deem it important to explicitly point out that they used them. Another possible explanation is that FL researchers are experts in FL and can set up and configure FL from the ground up on their own. Either way, this lack of transparency makes reproducing or extending their work challenging, if not infeasible.

These gaps in FL research motivated the creation of FLOps.

2.1.5. FL Frameworks & Libraries

To better comprehend why so many researchers did not specify or use FL frameworks, we examine the current landscape of available FL frameworks. We will keep this discussion short because Saidani already analyzed and evaluated FL frameworks in great detail in his master's thesis [60] from 2023. He examined FL libraries, frameworks, and benchmarks. He found that many FL tools exist for specific niche use cases and architectures. This is contrary to the opinions of his questioned FL practitioners and experts, who expect FL libraries and frameworks to focus on basic FL features, such as communication, aggregator-learner orchestration, security, and data aggregation. Saidani found that many libraries and frameworks, most of which are not production-ready, are still in an experimental research state.

To reduce complexity, he focused on the five most promising open-source frameworks. For a framework to be allegeable, it had to fulfill 2/3 of the following criteria. It needed

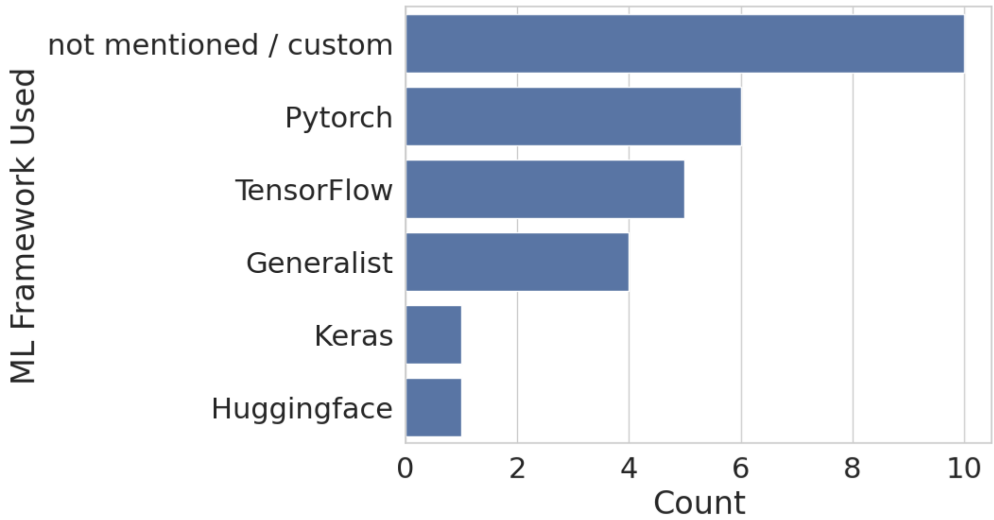

Figure 2.12.: Distribution of mentioned ML Frameworks in FL Papers

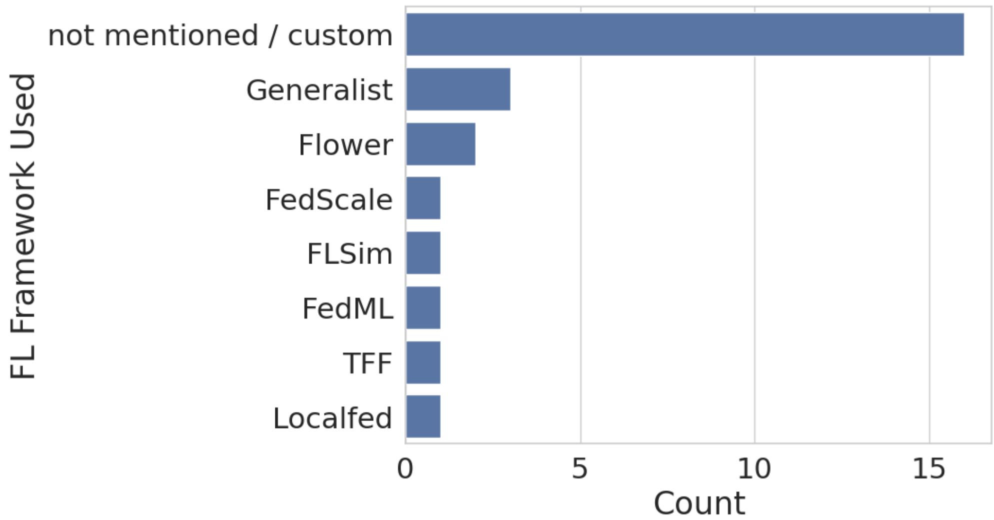

Figure 2.13.: Distribution of mentioned FL Frameworks in FL Papers

more than one thousand starts and 350 forks on GitHub. The interviewed experts had to mention it. The framework had to support all major operation systems. Because FL is rapidly evolving, we updated his findings and expanded upon them by including the last version released, the last commit pushed, and the number of open issues in the repository.

Framework	Version	Release	Stars	Forks	Last Commit	Issues
Pysyft [56]	0.9.0	two weeks ago	9.4k	2k	same day	2
Tensorflow Federated [63]	0.85.0	two days ago	2.3k	578	same day	168
FedML [15]	0.8.9	11 months ago	4.1k	776	3 months ago	118
Flower [22]	1.10.0	3 weeks ago	4.7k	815	same day	284
OpenFL [53]	9.3.4	last month	1.9k	426	same day	256

Table 2.2.: Updated FL Framework Comparison

Table 2.2 shows our updated FL Framework comparison. Note that we took these stats on 16.08.2024. These FL frameworks are in active development. Only FedML has not been updated for several months now.

Saidani's main original contribution was a novel FL benchmarking suite called FMLB (Federated Machine Learning Benchmark). He developed it to evaluate and compare the mentioned FL frameworks efficiently. His previous analysis and summary of existing frameworks were sound and helpful. However, we are critical of his evaluation results, especially the poor performance of Flower surprised us. We tried to replicate his experiments, but his provided code [61] lacks instructions on how to set up this benchmark application.

We simulated the experiments with the latest official flower version of that time, and made sure to stick as close as possible to the same experimental setup and configuration. Our findings show very different results. Flower manages to solve the experiment quickly and efficiently. Our results match the verdicts of other works comparing FL frameworks, such as [58] or [31]. [58] is the latest work that compares FL frameworks that we considered, and its verdict is that Flower even outperforms all its competition.

We decided to use Flower as the FL framework for FLOps.

2.1.6. Flower

This subsection provides an overview of our FL framework of choice, Flower, and highlights important aspects we rely upon. This open-source framework has a corresponding 2022 research paper [6]. Flower's first release (0.10.0) was published in November 2020, and its first major release (1.0.0) was published in 2022 [22]. One major target in Flower's paper was to narrow the gap between research and production, by allowing researchers to run high performance FL simulations and rapidly transition to tangible production environments all via the same tool. Another focal point of the paper was scale and parallelization.

Flower supports all major operating systems, containerization, and ML libraries. It aims to be easily customizable and extendable via a programming language and ML framework agnostic design. Flower strives to offer all popular FL features, such as support for different data types and distributions, pre-implemented popular FL algorithms, support for vertical and horizontal data splitting, traditional ML tasks, like regression or clustering, DNNs, LLMs, and security mechanisms, like secure aggregation. It enables the use of FL via CPUs or GPUs. Flower supports various FL variants, including PFL, edge computing, cross-silo, and cross-device. Flower handles and implements core FL components but does not handle many other aspects, like deployment, orchestration, dependency management, or containerization. Flower offers a mature set of FL simulation techniques. The default communication protocol is gRPC, which can be exchanged.

Users can easily change and add functionality to the framework by interacting with flexible abstractions and interfaces. The heart of Flower is its strategy concept. The aggregator uses this strategy to manage the FL processes. A strategy contains all necessary configurations, such as the FL algorithm to use, the minimum number of learners required for training or evaluation, and more. Users can pick from more than 30 existing strategies [4] or extend from basic strategies and develop their own behavior.

Flower has a lot to offer, but it still has its limits. It does not have native out-of-the-box support for model pruning, advanced security/privacy techniques, CFL, HFL, MLOps, or orchestration. Due to Flowers' flexible design users can implement their custom additions and strategies based on the available basic Flower components and realize many of these features.

On top of that, Flower has a modern, user-friendly, growing ecosystem. A dedicated sub-project called Flower Datasets [18] is part of this ecosystem. This project is still in its infancy (v0.3.0). It allows users to pull HuggingFace [33] datasets easily and split them into FL-optimized data fragments. Users can configure how to split this data up. In that way, Flower Datasets allows to use common non-federated homogeneous/IID datasets to be turned into challenging, federated, non-IID data, ideal for FL research and development. This ecosystem includes a well-structured and rich homepage [23], an extensive set of tutorials, guides, example projects [20], and documentation [24, 19] that ranges from beginner-friendly to advanced. The Flower team has a solid and growing connection to the public and its user base. They have open monthly community events [25], yearly summits [27], a blog [17], a dedicated discussion forum [21], a Slack space [26], and a YouTube channel [28].

It is straightforward to set up Flower and start working with it. Flower is directly available via Python's default package manager pip. One has to define the server/aggregator, strategy, and clients/learners. Users can implement the simplest case with a few dozen lines of Python code. The crucial part is to configure the strategy and clients properly. One needs to create a client class that extends from a Flower client and implement four essential methods that the framework will call during training. These methods include a getter and setter for the model parameters and one method each for training/fitting and evaluating the model.

2.2. Machine Learning Operations

In section 2.1.4, we discovered and discussed the gaps in the FL research field in terms of deployment, automation, orchestration, and usability. To improve upon these aspects and benefit the field, we first need to investigate modern best practices.

Patterns emerged during the history of applying computer science to solve problems and develop solutions. This includes various software engineering techniques and models. Famous examples are the waterfall model or agile development, such as Scrum. They all share the same goal of delivering high-quality, production-ready software. Over the last decades, a plethora of contrasting and intertwined disciplines have emerged that need to cooperate smoothly to develop, deliver, and operate modern software. One can group these tasks into two broad categories: development and operation.

2.2.1. DevOps

Older methods like the waterfall model split up the development and operations tasks and involved individuals. Software was first developed by one team and then operated by another. Due to the massive increase and modern requirements for flexibility and ability for change, developmental and operational tasks now form an interconnected infinite loop. For example, a company develops the first version of a software product in-house. To distribute this software among their clients and make it accessible, they build distributable software artifacts based on their source code. These artifacts might be container images or executable binaries. They publish these artifacts to online registries and roll live services out in the cloud. Users enjoy this product and request further features. The loop starts anew. The new features lead to unexpected bugs. The loop starts again, and so on. A software loop is only as fast as its slowest step.

In today's world, this loop is rarely a linear set of steps but several ones. Such loops are running in parallel at different stages several times per day. This concurrency is especially noticeable in projects that divide software into multiple decoupled parts. For example, in micro-service architectures, one service might be buggy and need fixing, while another is receiving a feature update. These dynamic and strong dependencies require developmental and operational tasks to work tightly together. This coupling also applies to IT professionals that need to cooperate and understand each other's areas well. This combined effort has become its own broad disciple called DevOps.

Due to this synergy, new techniques, tools and professions arose for various tasks, like building, deploying, testing, and monitoring. One core activity in this connected discipline is automation, because repetitive manual labor is an inefficient and expensive bottleneck. Prominent tools include Ansible and Gitlab-CI/CD. DevOps is a very broad discipline without concrete borders, so the activities of building artifacts or container images, orchestration, or knowledge sharing can be considered as part of DevOps. This notion would make Git, Docker, and Kubernetes the primary tools in this field.

An essential concept in DevOps is CI/CD, which stands for continuous integration, continuous delivery, and deployment. CI/CD focused on automating this software loop via custom pipelines. A DevOps pipeline is comparable to an assembly line in a factory. A software product needs to pass several connected stages with multiple steps. These stages can include testing, building, releasing, and deployment.

DevOps as a term was first mentioned around 2009 [39], yet it is still a very active and rapidly evolving field that unfortunately many other disciplines are not taking inspiration from or taking advantage of.

2.2.2. MLOps

MLOps is a young discipline that uses many best practices and techniques from DevOps and applies them to ML. Most DevOps techniques are applicable and beneficial for ML. Further considerations and tooling are required to support specialized ML requirements. ML differs from pure software development because it requires deep knowledge with different focal points, such as math and data science. In addition, training, replicating, or understanding an ML model and its code requires extensive and usually untracked background knowledge. This includes dependencies, environments, used training data, and whether the model is production-ready. Additionally, a model only supports specific input and output values of certain types. These unique requirements distinguish MLOps from conventional DevOps.

Inspecting the processes and challenges of a typical modern enterprise ML workflow demonstrates the need for MLOps. An exemplary company wants to develop a new ML-based feature to satisfy customer needs. Firstly, managers develop ideas for utilizing ML to solve these needs. These ideas get evaluated, accessed, and distilled into formal requirements. ML solutions require data for training and evaluation. The company starts gathering suitable data by scouting for data sources and providers. It collects and stores the found data in a custom data lake. Data engineers can now start preparing this data for training. Data preprocessing includes various steps, such as cleaning the data by removing outliers, wrong data samples, and undefined values. Other steps transform the data to make it more suitable for training. This includes applying normalization and standardization to slim down the feature space to reduce the curse of dimensionality. Other steps involve data analysis and visualization to find insightful patterns and ensure that the available data is sound and useful. These data preprocessing and data acquisition steps are an iterative process. With this data, ML engineers can start designing ML models.

ML model training and deployment are resource- and time-consuming steps. First model iterations are rarely ideal. To reach optimality, models require multiple train and test cycles with different architectures, configurations of layers, and hyperparameters. Just deploying a model is insufficient. Models need to work as intended for expected and unexpected use cases. The model performance can degrade over time. This can occur if the model is allowed to change after the initial training and deployment phase. Performance can also worsen for frozen models if circumstances change, such as the evolution of client needs and requests. Therefore, deployed model instances and their inference serving quality need monitoring. In case of bad performance, the model needs to be retrained or replaced with a better alternative. Such an improved version needs to complete most of the discussed steps again before redeployment. This workflow combines steps from business, management, data/ml/software engineering, and operations. Usually, in larger organizations, each step is handled by a dedicated team of experts who require working closely together. This exemplary workflow demonstrates that ML code and trained model alone cannot provide value in production environments. Enterprise ML requires various supporting disciplines and techniques to be usable, including versioning and infrastructure management. Due to these different iterative steps and stages, ML is a prime target for DevOps techniques.

MLOps is currently heavily underutilized, which slows down progress in ML enterprises. Many trained ML models are not deployed on production systems to provide real value. In 2020, only 14% of trained enterprise ML models were deployed to production in less than a week [2]. Getting an ML model to run on production environments requires entirely different skill sets, which many pure ML professionals, researchers, and hobbyists lack. Many individuals who perform ML lack training and industry experience as software engineers or developers. They might be unaware of DevOps practices or that ML can greatly benefit from them. To bring more awareness to MLOps, Kreuzberger et al. wrote a foundational paper [39] that provides an overview of MLOps and the current state of enterprise ML. They propose the first attempts at definitions and best practices for MLOps, including recommended architectures, tools, and workflows. They conclude that the field of ML is too fixated on academia and developing better ML models instead of optimizing tangible ML in production. Their verdict mirrors and reinforces the findings from section 2.1.4 regarding similar gaps in FL research.

2.2.3. MLflow

MLflow [50] is a one-in-all open-source MLOps platform that enriches and unifies common ML tasks and provides automatic solutions for ML challenges. Its first public version (0.2.0) was released in 2018. Version 1.0.0 came out in 2019. As of this writing, its latest version (2.15.1) was released in August 2024. MLflow's repository [49] accumulated 18.2k stars, 4.1k forks, and 756 contributors. Significant organizations, including Microsoft and Meta, use MLflow. MLflow supports various popular ML tools and frameworks, such as Keras, Pytorch, HuggingFace, and more. Furthermore, it is flexible for custom user extensions to support specialized functionality and tooling. MLflow has a rich and active community and ecosystem. This ecosystem includes detailed documentation [47], code examples [48], and places for discussion and receiving direct support (Slack). A great resource besides the official ones to learn more about MLflow is this online course [29]. MLflow helps users manage their ML workflow loops from conception to re-deployment.

MLflow divides its core features into four interconnected components. These components are rather conceptual groupings of functionalities than concrete isolated interfaces.

Tracking

MLflow can track and log ML experiments to help users record and compare their ML results. An experiment in MLflow is a set of runs. Each run represents a specific execution of a piece of code. A run can record various aspects of that execution, such as code version, metrics, or custom tags. Users can customize what should be tracked and how often. MLflow also offers automatic logging capabilities. Popular targets for tracking include parameters ranging from hyperparameters to custom metaparameters. The utilized code or training data can also be tracked, as well as metrics, such as accuracy and loss. MLflow offers its tracking via various APIs, including Python, Java, or REST. The tracking artifacts get recorded in a centralized place. By default, these artifacts are recorded in a local directory. These tracked records can also be stored and managed by a dedicated local or remote scalable tracking server. That way, users can easily share the results they have tracked with others. An MLflow tracking server comes with its own sophisticated and feature-rich web-based GUI. This GUI allows users to inspect, compare, and manage their recorded findings easily. MLflow tracking handles lightweight parameters, except for input data. It does not track or record trained models (weights and biases).

Models
MLflow can record and store ML models in uniform and popular formats. Popular formats are called "flavors" in MLflow and include pickle formats, python functions, and ML framework-specific solutions. Models can be stored together with exemplary input data, ML code, metadata, and a list of necessary dependencies for replication. MLflow differentiates between storing lightweight parameters, meta-information, and models. Model signatures can also be specified. These signatures are similar to function signatures in programming. They include the expected input and output types. Other tools can utilize such signatures to automatically create the correct Python functions or REST APIs for a model. Due to this standardized representation, many other tools can work with these models. This uniformity also makes deploying these models more efficient. MLflow allows users to deploy models to different environments via various ways, such as local inference servers (REST API), docker containers, and Kubernetes.

Registry
MLflow's model registry is comparable to an interface or API that works with a subset of logged models. It is not a dedicated standalone registry, unlike container image registries. It does not host complete models. This registry enables labeling and versioning for logged models. Labeling includes specific information that tells users if the model is currently in development, review, or production-ready. Not all logged models are part of the model registry. Users can manually or automatically decide if and what models they want to add to the model registry. This process is called registering a model. Every registered model is also a logged model. The benefit of this separation is that models in the registry are carefully selected and managed.

Projects
Projects allow replicating the exact ML environment for development. Unlike the tracked pieces of code from the tracking or model components, MLflow projects contain the entire codebase that was used to train a specific model. Projects aim to uniformly package ML code for reproducibility and distribution. The heart of an MLflow project is its MLproject file. It contains all the necessary information regarding dependencies and environments to guarantee identical conditions. This file can have multiple entry points similar to a Docker file. These entry points can be used for different use cases, including training or evaluation. Other users can quickly start using such projects due to MLflow's project CLI commands. A project's entry point can be called by the project CLI. MLflow can also invoke a project as part of a dynamically built docker container. The image gets built automatically via Docker after running the CLI command. The CLI allows running projects that are local, remote, or stored in a git repository. MLflow projects have a lot of potential, but they are not yet capable of fully handling robust

automatic containerization and dependency management. They work fine if run directly on a host machine that supports Docker. Most orchestrators expect images and deploy containers. It is not yet possible to orchestrate and deploy MLflow projects directly instead of using manually configured images. Issues arise when wrapping an MLflow project into a generic image and then internally calling its CLI to build and run the corresponding image. MLflow uses Docker directly, which is, in most cases, not possible inside a containerized environment. This limitation is represented in the official MLflow examples [45]. In this example all necessary dependencies are explicitly mentioned and installed in a custom Dockerfile that needs to be build manually to run the ML experiments. This emphasizes that MLflow projects cannot be automatically turned into standalone container images yet.

MLflow stores its artifacts in two different data stores. The default does not use any dedicated local or remote storage components. Instead, everything gets stored locally. All lightweight metadata, including metrics, tags, and results, are stored in the backend store. A backend store can be a database, a file server, or a cloud service. Heavy artifacts like trained models are kept in the artifact store. Registered models utilize both stores. Their metadata, such as versions and hyperparameters, are kept in the backend store. Their corresponding trained model is located in the artifact store.

MLflow supports many optional components that can be arranged in various architectures. In the simplest case, everything is stored and located on the local machine, with no need for a dedicated tracking server or data stores. More sophisticated structures support shared and distributed workflows and workloads. The mentioned components can be gradually added or removed. Therefore, MLflow allows flexible and custom solutions. For example, the artifact Store, backend Store, and tracking server can all be deployed on different machines and environments. This separation of concerns enables improved scalability and reduces singular points of failure. The tracking server can function as a proxy and bridge between machines that perform the ML experiments and the data stores. This approach enables centralized security and access control, simplifies client interactions.

MLflow lacks native support for FL. It does not explicitly mention or support FL. However, FL can profit from MLflow due to its modular design, that can be customized and applied to FL specific components and environments. For that reason FLOps is using MLflow.

2.3. Orchestration

Modern orchestration and containerization has been rapidly evolving for the last decade. To avoid issues due to different environments and dependencies containers arose as a lightweight alterative to heavier virtual machines. The most popular containerization software Docker was released in 2013. Multiple individual containers quickly got grouped together to separate concerns and allow decoupled workloads. This became especially useful and popular with the rise of micro service architectures. Handling various containers at the same time is challenging. Techniques such as Docker Compose made this easier, however dynamically scaling and handling failing containers was still difficult. These swarm of containers needed additional management tooling for orchestrating these dynamic processes. The most prominent orchestrator to date, Kubernetes, was released in 2014. Since then many new endevours formed to unify and streamline future developents in the field. Examples include the Open Container Initiate or the Moby Project. Many different alternatives and competitors to Docker and Kubernetes have developed over the years.

2.3.1. ML Containerization & Orchestration

Performing ML in containers is an effective approach. The goal of FLOps is to automate and orchestrate FL workloads on distributed heterogeneous machines. It is crucial to confirm that doing FL/ML via orchestrated containers is viable and that FL processes will not suffer from bottlenecks and problems due to running inside of containers. In 2017, Xu et al. [67] evaluated deep learning (DL) tool performance in docker containers. They analyzed CPU, GPU and I/O utilizations. They found that DL works equally well in containers as running it directly on the host machines.

Containerized ML is widely used today. In 2022, Openja et al. [54] analyzed more than 400 different open-source ML-based projects that use docker containers. These containers are used for various tasks. A popular application of containerization technologies in ML is to streamline and improve deployment efforts. This work demonstrates that nowadays containers are used in all ML-related tasks.

Orchestration efforts are optional for classic ML but essential for FL. As discussed in 2.1 classic centralized ML can be trained on a single powerful machine. FL, especially Cross-Device edge FL, can involve massive dynamic numbers of devices. Managing all these components is challenging. Training an ML model requires more than just the ML code and model. The device performing the training needs to support the necessary environment and dependency requirements. The setup and configuration of devices is a limiting and error-prone task. To avoid potential issues and to allow as many devices to participate as possible FL should use containers.

2.3.2. Oakestra

FLOps is primarily focused on enabling practical FL on real machines. The main target group for cross-device FL are heterogeneous edge devices. Kubernetes is not designed for the edge but for homogeneous, resource-rich cloud environments, whereas Oakestra is designed for the edge and is able to outperform Kubernetes there [5].

Oakestra is a hierarchical, open-source orchestrator for edge computing. It has a lightweight and flexible code base. It features many novel techniques, such as semantic overlay networking for efficient service communication.

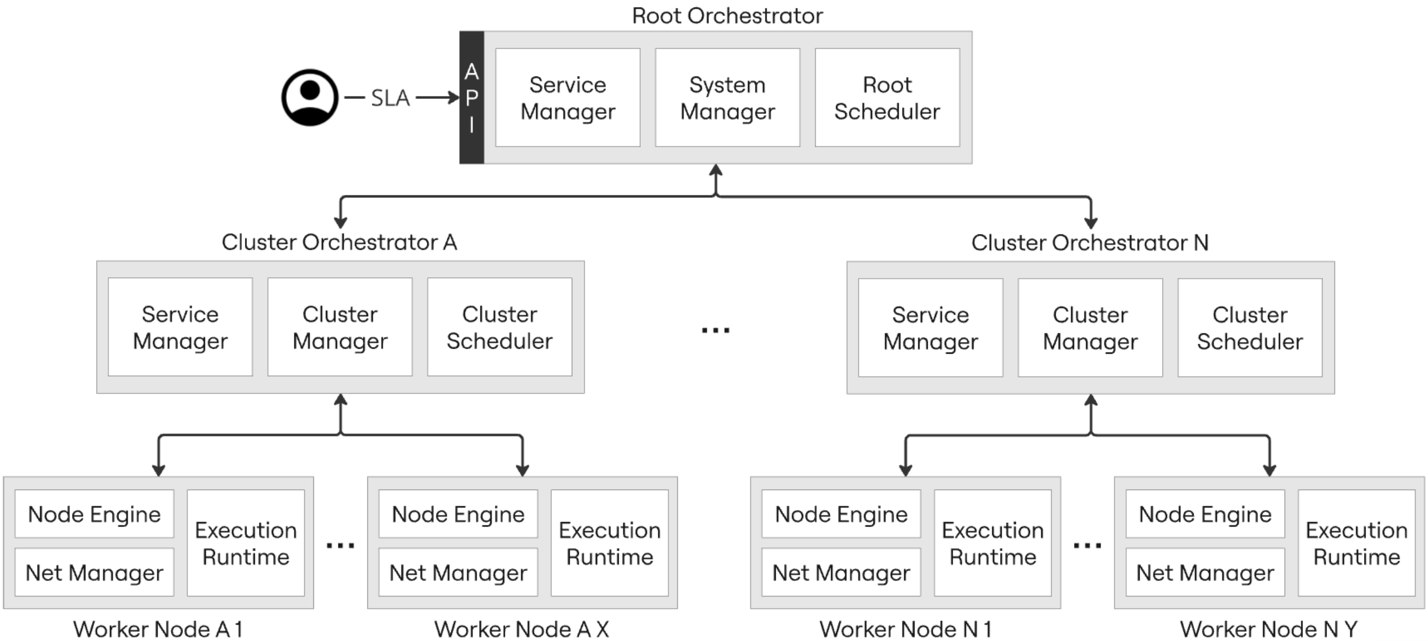

Figure 2.14.: Simplified Oakestra Architecture

Figure 2.14 shows a simplified architecture of Oakestra. This unique federated three-tiered structure allows for scalable delegate task scheduling and execution. Instead of a single control plane as in Kubernetes, Oakestra distributes its control among the root and cluster orchestrators. Oakestra is specialized to handle resource constrained, heterogeneous, devices that are spread across various geographical areas. Different infrastructure providers can have their own isolated cluster and cluster orchestrator. Cluster orchestrators only have access to detailed information about worker from their own cluster. The metrics they share with the root are distilled and no longer contain sensitive individual meta data. This is an ideal environment for FL because this layout supports privacy on an structural level. FLOps uses Oakestra to orchestrate its components.

2.4. Related Work

Only two previous works [11, 59] of those mentioned in 2.1.4 resemble FLOps. Both also noticed the lack in research regarding deploying ML and FL capabilities dynamically via containerization on devices. They use different technologies and offer different functionality than FLOps. They focus on other aspects and do not incorporate MLOps tools, automatic image builds, or automatic deployment of trained model inference servers.

2.4.1. On the feasibility of Federated Learning towards on-demand client deployment at the edge

In 2023, Chahoud et al. [11] proposed a three-layered FL architecture running on Kubernetes. Each following component has a matching image in DockerHub. Newly joining devices can simply pull these images.

Server
The first layer is the server or service provider. The server has various managerial responsibilities. It serves container images to voluntary devices and maintains secure connections to other layers. The server is the aggregator that manages the global model. Together with the mini-servers it determines which nodes should form a cluster. It handles service deployments and client selection after receiving requests from mini-servers.

Various components are part of the server. An oracle engine supports building the base model that will be send out to clients. The oracle determines what type of ML technique is required based on the environment, such as classic ML or DL. An enhanced FL aggregator handles stragglers and missing updates from failed learners. The aggregator uses a threshold to determine if an update should be included or discarded. A Kubeadm environment initializer that turns devices into mini-servers. This decision is based on available devices and the level of client mobility. From there followup setup steps are performed, such as cluster creation and population or container deployment on worker nodes. A communication manager upholds a stable connection between the different layers. A orchestrator manager administers the second layer mini-servers. It can dynamically determine and change which devices should be mini-servers. This task helps to gather better data for FL.

Orchestrators / mini-servers

Mini-servers handle Kubeadm clusters and surveil device movements. They deploy containers and add workers to clusters. Similar to cluster-orchestrators in Oakestra, mini-servers distribute management tasks and workloads among each other and away from the server. A mini-server containers a client profiler component and a client manager. The former gathers worker metrics, and the latter informs the server about changes in the environment. Relevant changes include joining and leaving clients. The client manager also protects against client starvation.

User Devices
User devices are the FL learners. They contain a client profiler that shares the machine metrics with a corresponding mini-server. The profiler also informs the mini-server in case of failures.

Open challenges and future work include more efficient and secure selection algorithms. More sophisticated logic is required to select learners for training to optimize FL results via data heterogeneity. Selecting devices to become mini-servers is a potential security hazard. Currently the authors assume mini-servers to be trustworthy and reliable.

Their evaluation results show that their FL solution is only 10% worse than the centralized alternative.

The similarities between this work and FLOps are the following. Both focus on making FL easy to use and do not focus on optimizing models or algorithms. They enable on-the-fly dynamic deployment and setup of FL components on unprepared devices via containerization technologies. Both provide prepared container images via public registries. This work's mini-servers and "root" server resemble Oakestra's root and cluster orchestrators. Oakestra is a dedicated orchestrator while this work's components are auxiliaries with less features.

This work used a different orchestrator, FL framework (augmented FedML) and image registry. FLOps supports classic and HFL. This work only supports HFL. They used a single hardcoded dataset and ML model for evaluation. This work does not offer different scenarios or utilize dedicated MLOps features and techniques. FLOps allows users to build and train various custom ML code. This works places its focus on 6G and actual real world movement of people, whereas FLOps is more general and feature rich.

2.4.2. Towards Developing a Global Federated Learning Platform for IoT

In 2022, Safri et al. [59] developed a prototype to improve FL on IoT devices. This work enables distributed ML model deployment, federated task orchestration, and monitoring of system state and model performance. They called their approach FedIoT.

Their three-layered architecture resembles the one from [11] and Oakestra. Their root server/orchestrator is called global orchestrator. It acts as an FL aggregator and dynamically configures and deploys local orchestrators via an API. Their local orchestrator is not equivalent to cluster orchestrators or mini-servers. This work focuses on enterprise IoT. IoT devices are usually not capable of handling common ML training due to their limited resources. This work acknowledges this and uses the IoT devices only as data providers but not as learners. Therefore, this work performs classic FL instead of HFL. Local orchestrators are learners in this architecture. They need to be in the proximity of IoT devices to be able to receive their data. Additionally, they provide customizable data preprosessing and evaluation code to be injected via the API.

This work offers additional tooling, such as a custom compressor and monitoring. The compressor is a dedicated component to reduce the size of large files. Monitoring agents are deployed on the local and global orchestrators that measure resources and CO2. A custom GUI presents these metrics.

As future work the authors wanted to add more FL algorithms and add more sophisticated logic to select participants for training based on the monitored metrics.

There are obvious similarities between this work and FLOps/Oakestra. Both want to provide a one-in-all solution to perform FL on tangible devices via containerization and orchestration. They want to automate setup, dependency management, configuration, and metric gathering. Additionally, they want to improve comprehension and observability by providing a GUI.

This work differs compared to FLOps in multiple ways, besides FLOps larger set of features. As already mentioned above, this work only offers classic FL and has a different and less mature architecture than FLOps thanks to Oakestra. This work is very short, thus lacks details and readability. It has no open sourced code to inspect and replicate its implementation. FLOps has this thesis documenting it in great detail and is fully open source. This works tries to implement all components by itself from group up, such as orchestration, monitoring, and FL. FLOps utilizes and combines existing sophisticated solutions to offer higher quality features and performance. For example, this work's GUI is a simple Grafana dashboard, that offers a lot fewer features and is read-only. FLOps utilizes MLflow to provide a sophisticated suite of MLOps tools and functionalities.

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

List of Figures

List of Tables

Bibliography

[1] A. M. Abdelmoniem, A. N. Sahu, M. Canini, and S. A. Fahmy. "REFL: Resource-Efficient Federated Learning." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 215–232. ISBN: 9781450394871. DOI: 10.1145/3552326.3567485.

[2] *Algorithmia 2020 State of Enterprise Machine Learning*. Tech. rep. Accessed: 2024-08-17. 2020.

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

[4] *Available Flower Strategies*. Accessed: 2024-08-16. URL: https://github.com/adap/flower/tree/main/src/py/flwr/server/strategy.

[5] 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.

[6] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, K. H. Li, T. Parcollet, P. P. B. de Gusmão, and N. D. Lane. *Flower: A Friendly Federated Learning Research Framework*. 2022. arXiv: 2007.14390 [cs.LG].

[7] N. S. Bisht and S. Duttagupta. "Deploying a Federated Learning Based AI Solution in a Hierarchical Edge Architecture." In: *2022 IEEE 10th Region 10 Humanitarian Technology Conference (R10-HTC)*. 2022, pp. 247–252. DOI: 10.1109/R10-HTC54060.2022.9929526.

[8] 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.

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

[10] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar. "LEAF: A Benchmark for Federated Settings." In: (2019). arXiv: 1812.01097 [cs.LG].

[11] M. Chahoud, S. Otoum, and A. Mourad. "On the feasibility of Federated Learning towards on-demand client deployment at the edge." In: *Information Processing & Management* 60.1 (2023), p. 103150. ISSN: 0306-4573. DOI: https://doi.org/10.1016/j.ipm.2022.103150.

[12] Z. Chai, Y. Chen, A. Anwar, L. Zhao, Y. Cheng, and H. Rangwala. "FedAT: a high-performance and communication-efficient federated learning system with asynchronous tiers." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476211.

[13] V. Chandrasekaran, S. Banerjee, D. Perino, and N. Kourtellis. "Hierarchical Federated Learning with Privacy." In: (2022). arXiv: 2206.05209 [cs.LG].

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

[15] *FedML Github*. Accessed: 2024-08-16. URL: https://github.com/FedML-AI/FedML.

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

[17] *Flower Blog*. Accessed: 2024-08-12. URL: https://flower.ai/blog/.

[18] *Flower Datasets*. Accessed: 2024-08-16. URL: https://flower.ai/docs/datasets/.

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

[20] *Flower Examples*. Accessed: 2024-08-12. URL: https://github.com/adap/flower/tree/main/examples.

[21] *Flower Forum*. Accessed: 2024-08-12. URL: https://discuss.flower.ai/.

[22] *Flower Github*. Accessed: 2024-08-16. URL: https://github.com/adap/flower.

[23] *Flower Homepage*. Accessed: 2024-08-16. URL: https://flower.ai/.

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

[25] *Flower Monthly*. Accessed: 2024-08-12. URL: https://flower.ai/events/flower-monthly/.

[26] *Flower Slack*. Accessed: 2024-08-12. URL: https://flower.ai/join-slack.

[27] *Flower Summit*. Accessed: 2024-08-12. URL: https://flower.ai/events/flower-ai-summit-2024/.

[28] *Flower Youtube Channel*. Accessed: 2024-08-12. URL: https://www.youtube.com/@flowerlabs.

[29] J. Garg. *MLflow in Action - Master the art of MLOps using MLflow tool*. Accessed: 2024-08-18. URL: `https://www.udemy.com/course/mlflow-course/`.

[30] G. Genovese, G. Singh, C. Campolo, and A. Molinaro. "Enabling Edge-based Federated Learning through MQTT and OMA Lightweight-M2M." In: *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*. 2022, pp. 1–5. DOI: `10.1109/VTC2022-Spring54318.2022.9860964`.

[31] R. Hamsath Mohammed Khan. *A Comprehensive study on Federated Learning frameworks : Assessing Performance, Scalability, and Benchmarking with Deep Learning Model*. Accessed: 2024-08-16. 2023.

[32] A. Hilmkil, S. Callh, M. Barbieri, L. R. Sütfeld, E. L. Zec, and O. Mogren. "Scaling Federated Learning for Fine-Tuning of Large Language Models." In: *Natural Language Processing and Information Systems*. Ed. by E. Métais, F. Meziane, H. Horacek, and E. Kapetanios. Cham: Springer International Publishing, 2021, pp. 15–23. ISBN: 978-3-030-80599-9.

[33] *Hugging Face Homepage*. Accessed: 2024-08-16. URL: `https://huggingface.co/`.

[34] M. Isaksson, E. L. Zec, R. Cöster, D. Gillblad, and Š. Girdzijauskas. "Adaptive Expert Models for Personalization in Federated Learning." In: (2022). arXiv: `2206.07832 [cs.LG]`.

[35] Q. Jia, L. Guo, Y. Fang, and G. Wang. "Efficient Privacy-Preserving Machine Learning in Hierarchical Distributed System." In: *IEEE Transactions on Network Science and Engineering* 6.4 (2019), pp. 599–612. DOI: `10.1109/TNSE.2018.2859420`.

[36] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas. "Model Pruning Enables Efficient Federated Learning on Edge Devices." In: *IEEE Transactions on Neural Networks and Learning Systems* 34.12 (2023), pp. 10374–10386. DOI: `10.1109/TNNLS.2022.3166101`.

[37] 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`.

[38] 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`.

[39] D. Kreuzberger, N. Kühl, and S. Hirschl. "Machine Learning Operations (MLOps): Overview, Definition, and Architecture." In: *IEEE Access* 11 (2023), pp. 31866–31879. DOI: `10.1109/ACCESS.2023.3262138`.

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

[41] W. Y. B. Lim, J. S. Ng, Z. Xiong, J. Jin, Y. Zhang, D. Niyato, C. Leung, and C. Miao. "Decentralized Edge Intelligence: A Dynamic Resource Allocation Framework for Hierarchical Federated Learning." In: *IEEE Transactions on Parallel and Distributed Systems* 33.3 (2022), pp. 536–550. DOI: 10.1109/TPDS.2021.3096076.

[42] Z. Liu, D. Li, J. Fernandez-Marques, S. Laskaridis, Y. Gao, Ł. Dudziak, S. Z. Li, S. X. Hu, and T. Hospedales. "Federated Learning for Inference at Anytime and Anywhere." In: (2022). arXiv: 2212.04084 [cs.LG].

[43] 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.

[44] 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.

[45] *MLflow Docker Example*. Accessed: 2024-08-18. URL: https://github.com/mlflow/mlflow/tree/master/examples/docker.

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

[47] *MLflow Documentation*. Accessed: 2024-08-18. URL: https://mlflow.org/docs/latest/index.html#.

[48] *MLflow Examples*. Accessed: 2024-08-18. URL: https://github.com/mlflow/mlflow/tree/master/examples.

[49] *MLflow GitHub*. Accessed: 2024-08-18. URL: https://github.com/mlflow/mlflow.

[50] *MLflow Homepage*. Accessed: 2024-08-18. URL: https://mlflow.org/.

[51] J. Nguyen, J. Wang, K. Malik, M. Sanjabi, and M. Rabbat. "Where to Begin? On the Impact of Pre-Training and Initialization in Federated Learning." In: (2023). arXiv: 2206.15387 [cs.LG].

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

[53] *OpenFL*. Accessed: 2024-08-16. URL: https://github.com/openfl/openfl.

[54] M. Openja, F. Majidi, F. Khomh, B. Chembakottu, and H. Li. "Studying the Practices of Deploying Machine Learning Projects on Docker." In: *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. EASE '22. Gothenburg, Sweden: Association for Computing Machinery, 2022, pp. 190–200. ISBN: 9781450396134. DOI: 10.1145/3530019.3530039.

[55] 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.

[56] *Pysyft Github*. Accessed: 2024-08-16. URL: https://github.com/OpenMined/PySyft.

[57] L. Qu, Y. Zhou, P. P. Liang, Y. Xia, F. Wang, E. Adeli, L. Fei-Fei, and D. Rubin. "Rethinking Architecture Design for Tackling Data Heterogeneity in Federated Learning." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022, pp. 10061–10071.

[58] P. Riedel, L. Schick, R. von Schwerin, M. Reichert, D. Schaudt, and A. Hafner. "Comparative analysis of open-source federated learning frameworks - a literature-based survey and review." In: *International Journal of Machine Learning and Cybernetics* (June 2024). DOI: 10.1007/s13042-024-02234-z.

[59] H. Safri, M. M. Kandi, Y. Miloudi, C. Bortolaso, D. Trystram, and F. Desprez. "Towards Developing a Global Federated Learning Platform for IoT." In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 2022, pp. 1312–1315. DOI: 10.1109/ICDCS54860.2022.00145.

[60] A. Saidani. "A Systematic Comparison of Federated Machine Learning Libraries." MA thesis. 2023.

[61] A. Saidani. *FMLB Github*. Accessed: 2024-08-16. 2023. URL: https://github.com/sdn98/BFML/tree/master.

[62] *Scopus Homepage*. Accessed: 2024-08-14. URL: https://www.scopus.com/.

[63] *Tensorflow-Federated Github*. Accessed: 2024-08-16. URL: https://github.com/google-parfait/tensorflow-federated.

[64] *Udemy Homepage*. Accessed: 2024-08-14. URL: https://www.udemy.com/.

[65] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor. "Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization." In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 7611–7623.

[66] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. "Adaptive Federated Learning in Resource Constrained Edge Computing Systems." In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), pp. 1205–1221. DOI: 10.1109/JSAC.2019.2904348.

[67] P. Xu, S. Shi, and X. Chu. "Performance Evaluation of Deep Learning Tools in Docker Containers." In: *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. 2017, pp. 395–403. DOI: 10.1109/BIGCOM.2017.32.

[68] Z. Yang, S. Fu, W. Bao, D. Yuan, and A. Y. Zomaya. "Hierarchical Federated Learning with Momentum Acceleration in Multi-Tier Networks." In: (2022). arXiv: 2210.14560 [cs.LG].

[69] C. You, K. Guo, H. H. Yang, and T. Q. S. Quek. "Hierarchical Personalized Federated Learning Over Massive Mobile Edge Computing Networks." In: *IEEE Transactions on Wireless Communications* 22.11 (2023), pp. 8141–8157. DOI: 10.1109/TWC.2023.3260141.

[70] Z. Yu, J. Hu, G. Min, Z. Wang, W. Miao, and S. Li. "Privacy-Preserving Federated Deep Learning for Cooperative Hierarchical Caching in Fog Computing." In: *IEEE Internet of Things Journal* 9.22 (2022), pp. 22246–22255. DOI: 10.1109/JIOT.2021.3081480.

[71] H. Zhang, J. Bosch, and H. H. Olsson. "EdgeFL: A Lightweight Decentralized Federated Learning Framework." In: (2023). arXiv: 2309.02936 [cs.SE].

Appendices

A. Additional FL Research Paper Analysis

The following two tables (A.1, A.2) refer to the omitted FL papers that we examined for FLOps. The main part can be found here (2.1).

Table A.1 shows the first half of the remaining FL papers and table A.2 depicts the second half. When there is no content (-) in the "Limitations & Future Work" column that means that the authors did not mention any explicitly and that we did not notice anything specifically.

ID	Contributions	Limitations Future Work
[34]	Improved an existing PFL algorithm that used clustered models (but discarded all but one in the end). A novel idea to improve performance by using these cluster models as experts in a MoE (Mixture of Experts) setup.	-
[65]	Analysis of drift that occurs due to different learner speeds Novel ideas eliminating that drift.	This work does not consider hierarchical structures, clusters/tiers, or privacy/security.
[35]	Efficiency improvements for privacy-preserving ML techniques for hierarchically distributed structures. Different data partitions and distributions, such as vertical and non-IID, were considered.	Written in 2019. Many other newer papers have investigated HFL security/privacy further.
[10]	A benchmark for federated settings, especially FL, with implementations and datasets.	Outdated benchmark from 2019. When we tried to use it, we encountered many errors and problems, such as broken dependencies, failing example code, and more.
[7]	Proof-of-concept that demonstrates that FL can be deployed and used in hierarchical architectures that fulfill specific industry standards.	The findings and experiments are very basic. Further topics such as diverse network conditions, heterogeneous data, and resources should be investigated.
[68]	Accelerated and improved FL training and the aggregation algorithm via a hierarchical structure and ML momentum.	Security, privacy, and challenging network conditions were not considered.
[32]	Analysis of LLM behavior in FL when using different numbers of learners.	Due to its proof-of-concept nature, this work only features simple experiments that yield few new insights.
[30]	A novel approach to finding and sharing information between FL components and discovering learners. This work uses MQTT with semantic URIs representing the clients' properties, including their resources.	It is a very short paper. The experiments are only simulated. This work's approach was not extensively compared to classic or novel techniques.

Table A.1.: Additional FL Papers considered for FLOps - Part I

ID	Contributions	Limitations Future Work
[13]	Analysis of HFL benefits for security. A novel secure aggregation method and hierarchical DP for HFL.	The number of (online) clients per zone has to be small. Further privacy improvements should be investigated.
[36]	Introduction of distributed adaptive FL model pruning.	Privacy and security were not considered. Further optimizations are possible, primarily focused on GPUs.
[57]	Analysis of the use of transformers in FL compared to other architectures. Findings show that transformers are excellent and should be preferred for FL.	Further investigations are required on how transformers behave with other, latest FL algorithms and privacy/security schemas.
[71]	A scalable edge-only (serverless) FL framework. It utilizes synchronous training and promises rapid integration, prototyping, and deployment.	Planned improvements for this framework include resource optimizations like model compression and quantization and adaptive aggregation strategies based on network conditions, resources, and data diversity. The framework assumes P2P without addressing diverse network conditions. It does not consider security or privacy. The evaluation only checked image classification tasks.
[69]	This paper is likely the first to combine PFL with HFL in a three-tiered structure. It proves mathematically that its approach works and converges. This work includes many interesting insights regarding HPFL.	-
[66]	Analysis of the effects of different global/local update frequencies. A new algorithm to determine global aggregation frequency instead of using the common static one.	Diverse resource usage should be investigated.
[42]	A combination of FL with transfer-learning on Transformers. A parameter efficient (PE) learning method to adapt pre-trained Transformer Foundation Models (FMs) in FL. A novel PE adapter that modulates all pre-trained Transformers layers, enabling flexible early predictions.	-

Table A.2.: Additional FL Papers considered for FLOps - Part II