SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

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
UML Universal Modeling Language

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's regulation to protect personal data, the GDPR (General Data Protection Regulation) [52], or the California Consumer Privacy Act (CCPA) [35]. 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 [39]. 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. Furthermore, the field of FL lacks more advanced techniques to increase productivity that other domains have already been using for several years, such as modern DevOps or MLOps 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 [15]. 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 [10] to manage dependencies and build images. FLOps utilizes a pioneering FL framework called Flower [20] to execute its FL training loops. The mentioned runtime observability features are available via a mature MLOps tool called MLflow [41]. 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 [17]. 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 [48]. 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 [38]. 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 [39].

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
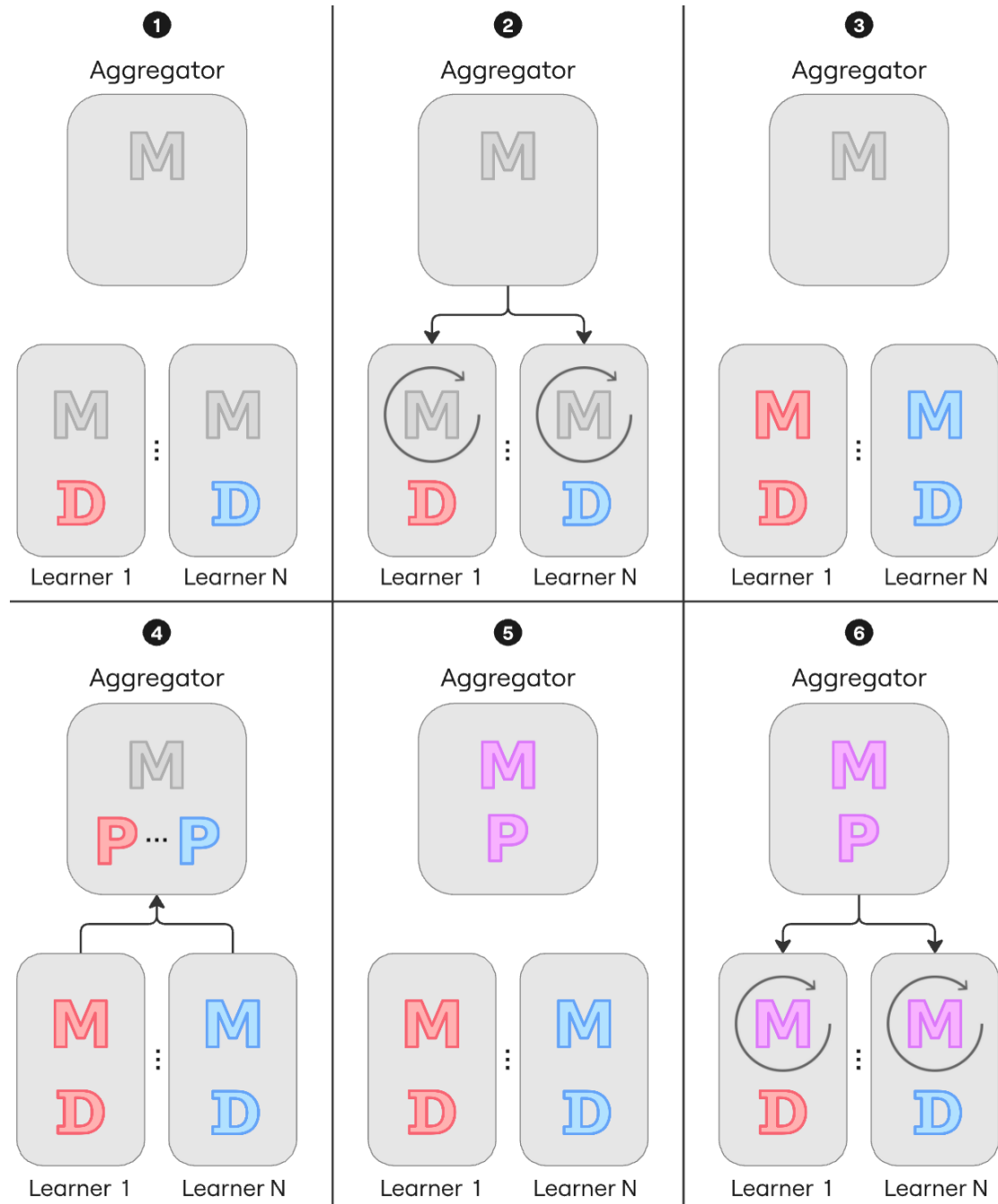
Figure 2.2: Basic Federated Learning

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 [38]. 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

This subsection explores essential supplementary FL concepts to understand the field better.

FL compared to Distributed Learning

At first glance, FL seems similar to Distributed Learning (DL). Both get used for computationally expensive large ML tasks. Computations get distributed among many weaker machines that train individually. This approach increases convergence times and avoids the need for powerful devices. 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 all data before splitting it up among its fixed and predefined clients [38]. 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. This comparison demonstrates that FL and DL have similarities but also crucial differences.

FL Variety

FL is a diverse discipline with various possible applications and use cases. Most FL work focuses on end-user/edge/IoT devices. FL is not exclusive to these environments and can work in conventional cloud environments. As discussed in the first subsection, FL can train DNNs. FL can also apply to classic ML models, such as linear models (logistic regression, classification, and more) or decision trees for explainable classifications. FL also supports horizontal, vertical, and split learning. This work omits discussing these techniques to avoid bloat. More information about these and other methods is available in [38]. Plenty of FL optimizations exist for each ML variant, such as custom algorithms and strategies.

Personalization can help if the global model is too general and does not satisfy a learner's individual needs. 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. A mixture between the global and purely local model can be used for inference. PFL is a deep and growing subfield of FL. Helpful resources to learn more about PFL include [38, 65, 30].

FL Security & Privacy

The field of FL highly focuses on security and privacy because protecting those was the key motivation for FL's creation. 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.

A variety of FL adversaries and threats exist. Adversarial insiders, such as malicious aggregators or learners, are part of the FL process. Adversarial outsiders try to interfere from beyond the FL system. One threat example is manipulation, where insiders try to distort the model to their advantage. Attackers tinker with FL components they 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- Each learner 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, this learner's update overshadows other participants and influences the global model the most. As a result, even very scarce, well-timed attacks in FL can have devastating impact. [38]

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. 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. [38]

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. [38]

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 [33].

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. Due to this great pool of learners, only a subset typically trains per round. The identities of the participating learners are usually unimportant. Due to the nature of these devices and their environments, cross-device FL needs to manage special challenges. Challenges include 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. Note that from now on, when we mention FL in this work, we mean cross-device FL.

Different FL architectures exist to support large-scale FL environments. FLOps wants to benefit from the unique three-tiered Oakestra [5] architecture. Such a scenario has two main challenges. The first challenge is managing a massive number of connections and aggregations. The second one is 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. The following is an overview of prominent FL architectures.
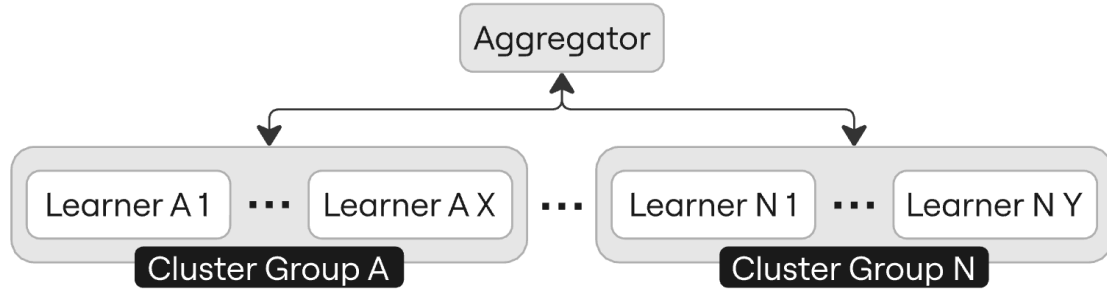
Clustered FL

Figure 2.3: Clustered FL Architecture

Figure 2.3 shows the Clustered FL (CFL) architecture that groups similar learners into clusters. CFL can form clusters based on local data distribution, training latency, available hardware, or geographical location. The issue of the singular aggregator as a bottleneck persists. The main challenge for CFL is choosing a suitable clustering strategy and criteria for the concrete use case. If the criteria are biased, updates from preferred clusters might be heavily favored, resulting in a biased global model with bad generalization. Another task properly profiling the nodes to match them to the correct cluster. 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. Too intrusive profiling can lead to compromised privacy. The benefits of CFL are its ease of implementation, familiar architecture to classic FL, and flexibility to tune clustering/selection dynamically. CFL can be combined 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. Its clustering overhead becomes critical with larger numbers of nodes. More information about CFL is available in [33, 13, 38, 36].

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. 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
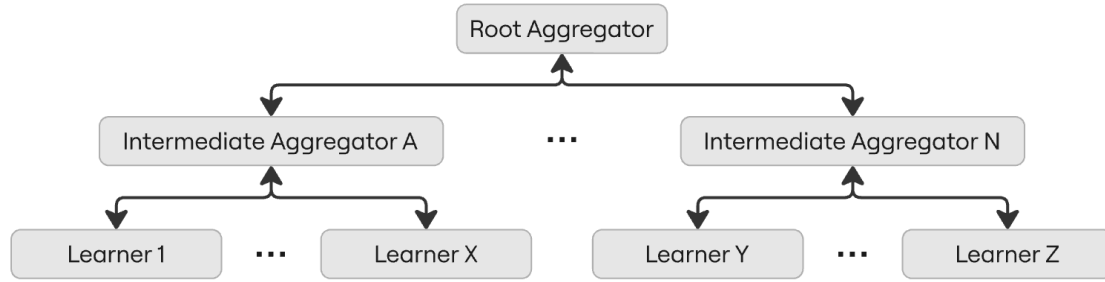
Figure 2.4: Hierarchical FL Architecture

the root aggregator. The root combines the intermediate parameters into global ones and sends them downstream for further FL rounds. HFL's structure requires changing the underlying FL architecture.

The proper design and implementation, and 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. The intermediate aggregated model can be biased if too few learners are assigned. Thus, the infrastructure resources 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. These findings show that special FL architectures, such as HFL, require careful consideration and correct implementation.

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. HFL provides a powerful way to improve scalability for FL if done right. Additional information and interesting uses of HLF are available in [7, 64, 14, 36, 38, 65].

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. [38]

Asynchronous FL

Asynchronous FL allows learners to train continuously and freely push their updates to the aggregator. 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. [38]

2.1.4 FL Research
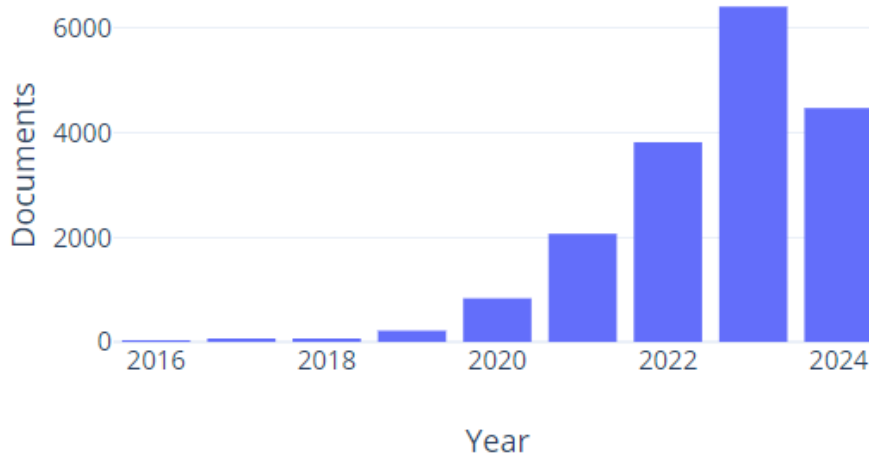
Figure 2.5: Evolution of FL Publications

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 [59].) The idea for this graph is based on [57]. Graph 2.5 uses a different query with the latest available data.

Before creating FLOps, we looked for research gaps in the fields of ML at the edge, specifically FL. 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. Discussing each paper in detail would heavily bloat this thesis. This subsection presents key and meta-findings instead. While working through the material, we created and incrementally updated a database in which we noted specific properties of each paper. These properties include one or multiple categories in which the paper fits in. Additional properties include the initial problems or challenges the authors tried to resolve, their contributions, results, limitations, and envisioned future work. We also noted what ML or FL frameworks or libraries they claimed to use.

Tables 2.1, 2.2, and 2.3 depict our analyzed FL papers. They present the documented contributions, limitations, and future work properties. 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. These tables provide a good impression of the examined FL papers. Patterns and trends can be extracted from these papers based on the documented properties.

Patterns and trends help to better understand the research field of FL as a whole. Figure 2.6 shows the different categories and their distribution. Most examined papers focused on performance, trying new concepts, finding best practices, and exploring different FL architectures. Only two papers focused on deployment and orchestration.

Figure 2.7 reveals a similar trend. The primary focus is on investigating new concepts or improving existing performance, scalability, and complexity bottlenecks. Several papers have aimed to narrow the gap between industry and research or 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. Mathematical and conceptual proofs dominate this chart. They prove that novel architectures and algorithms work as proposed. Contributions do not seem to focus on improving the initial setup, deployment, and configuration processes.

The achieved results mirror previous findings. Figure 2.9 shows that these contributions lead to more efficient FL. Improved aspects include speed, resource utilization, training results, and handling of heterogeneous data.

Figure 2.10 reflects this perception. If specified, the focal point is on improving privacy and security, further performance optimizations, or adding support for more ML use cases. Even the future focus is not on optimizing accessibility, usability, or the mentioned initial vital steps. These documented properties might be biased, and the inspected sample size of papers is relatively small. To improve confidence in these findings, we compare them with the total number of published works about FL.

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.
[33]	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.
[66]	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.
[47]	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.
[36]	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.
[13]	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: FL Papers considered for FLOps - Part I

ID	Contributions	Limitations Future Work
[30]	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.	-
[61]	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.
[31]	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.
[11]	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.
[64]	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.
[28]	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.
[26]	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 2.2: FL Papers considered for FLOps - Part II

ID	Contributions	Limitations Future Work
[14]	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.
[32]	Introduction of distributed adaptive FL model pruning.	Privacy and security were not considered. Further optimizations are possible, primarily focused on GPUs.
[54]	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.
[67]	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.
[65]	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.	-
[62]	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.
[37]	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.	-

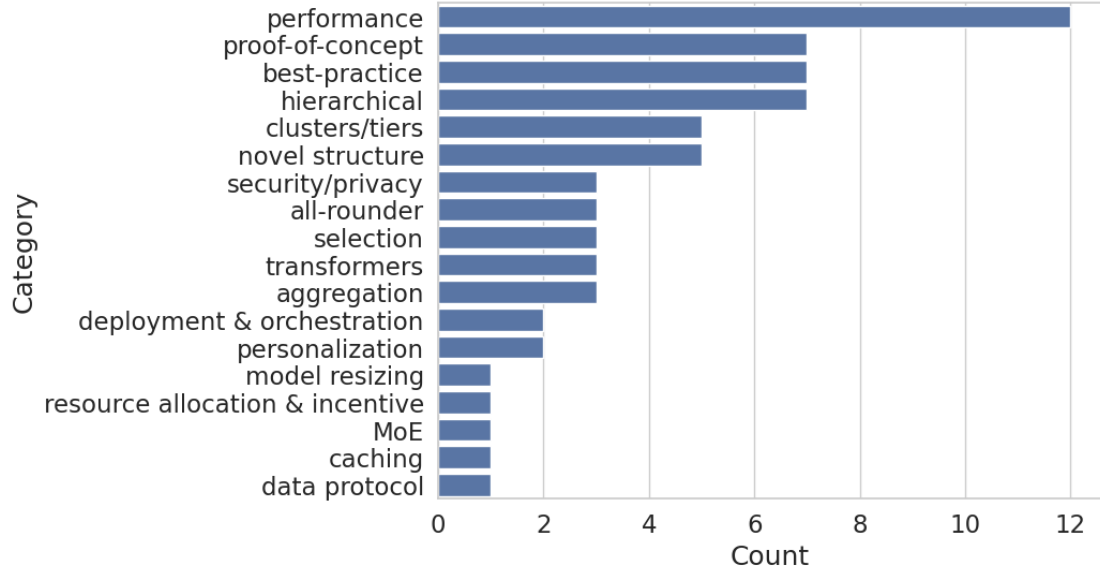Table 2.3: FL Papers considered for FLOps - Part III

Figure 2.6: FL Paper Categories

Figure 2.7: Targeted Problems & Challenges of FL Papers
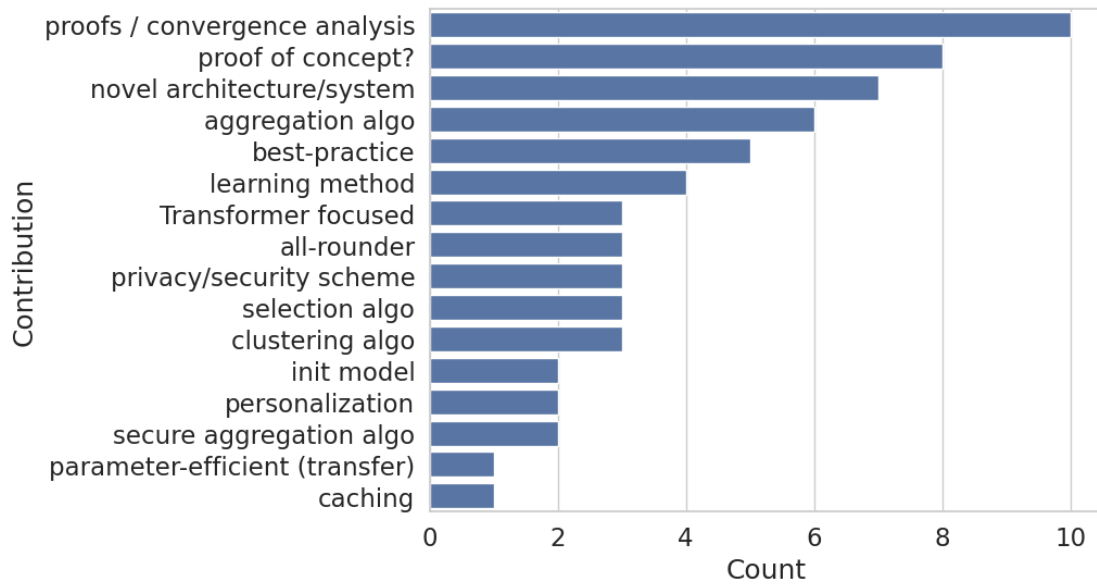
Figure 2.8: FL Paper Contributions
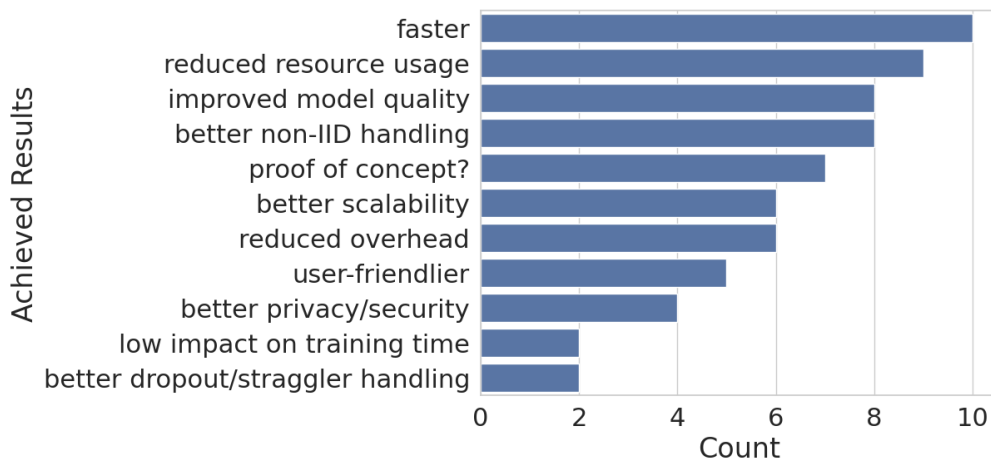
Figure 2.9: Achieved Results of FL Papers
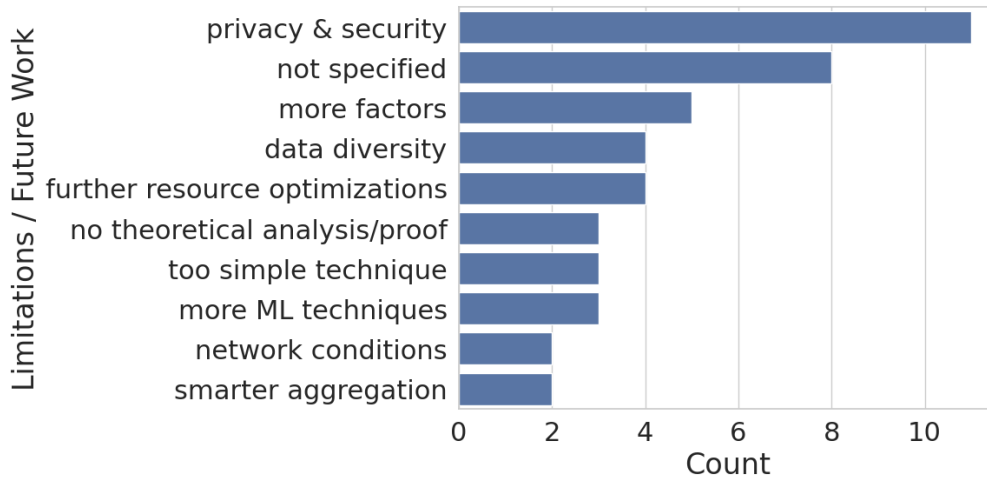
Figure 2.10: Limitations & Future Work of FL Papers
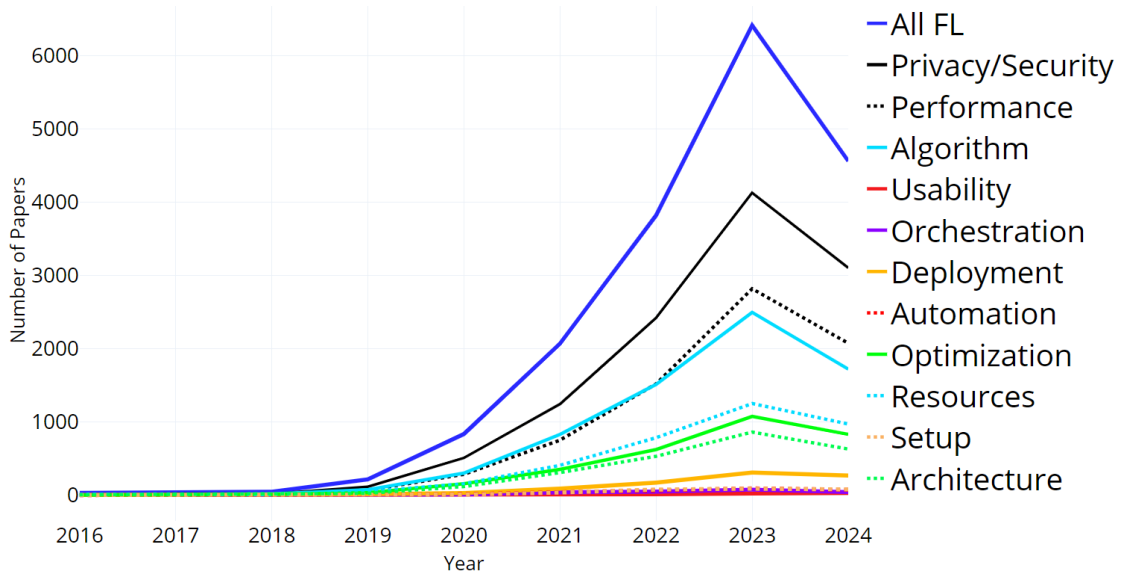
Figure 2.11: Evolution of FL Publications based on Keywords

Figure 2.11 depicts how many works have been published in FL with specific keywords that match our custom categories. We applied the same method to gather the data as for 2.5. 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. Furthermore, they seem to motivate their readers to optimize these setups based on their findings instead of replicating and configuring such an FL setup initially. These tendencies are visible when inspecting the ML and FL frameworks and libraries the authors mentioned they used. The following figure is again based on 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.
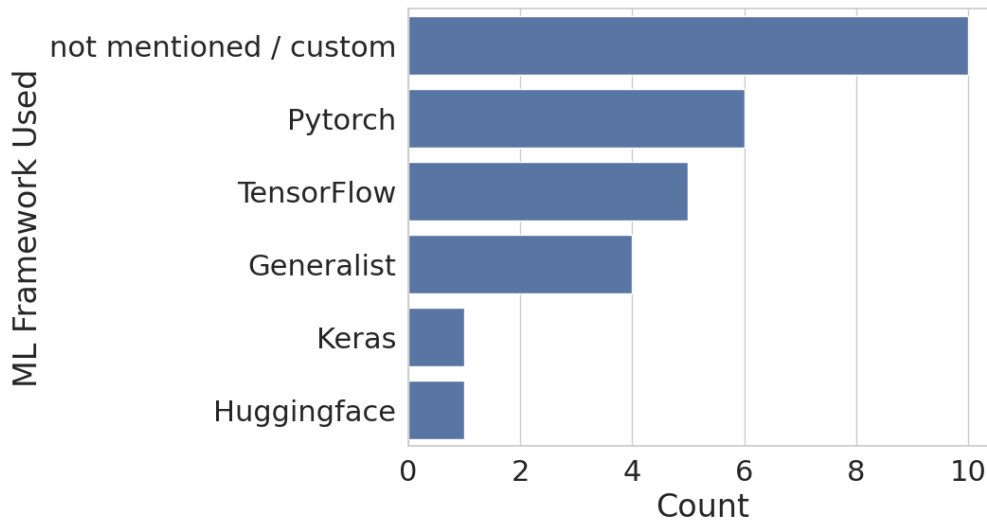
Figure 2.12: Distribution of mentioned ML Frameworks in FL Papers

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. FL researchers might be using FL frameworks. However, due to the framework's 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. 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.
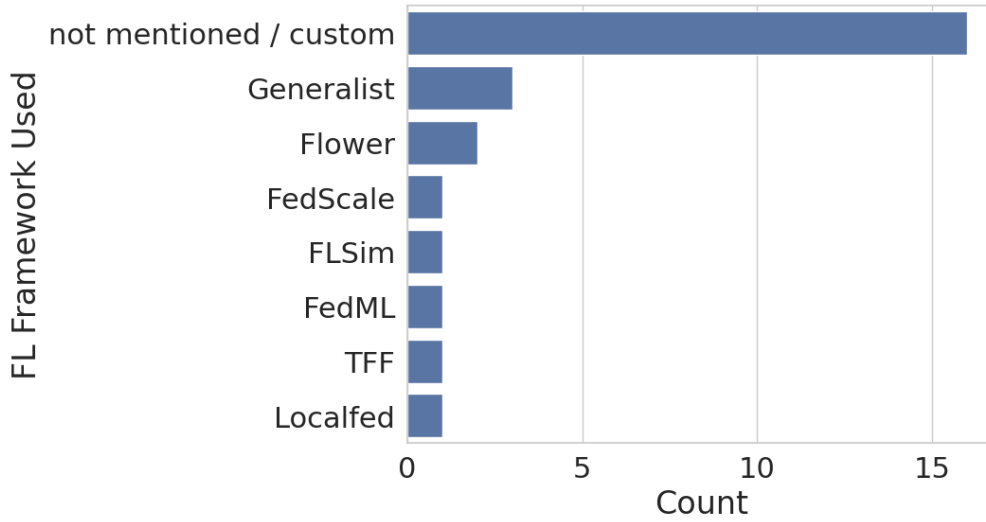
Figure 2.13: Distribution of mentioned FL Frameworks in FL Papers

2.1.5 FL Frameworks & Libraries

This subsection examines the current landscape of available FL frameworks. The goal is to better comprehend why most researchers did not specify or use FL frameworks in the previous findings. This discussion will be brief because Saidani already analyzed and evaluated FL frameworks in great detail in his master's thesis [57] from 2023. He examined FL libraries, frameworks, and benchmarks. Many FL tools exist for specific niche use cases and architectures [57]. This finding contradicts the opinions of his questioned FL practitioners and experts. They expected FL libraries and frameworks to focus on basic FL features, such as communication, aggregator-learner orchestration, security, and data aggregation. Many libraries and frameworks, most of which were not production-ready, were still in an experimental research state [57].

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 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. We included 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 [53]	0.9.0	two weeks ago	9.4k	2k	same day	2
Tensorflow Federated [60]	0.85.0	two days ago	2.3k	578	same day	168
FedML [16]	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 [50]	9.3.4	last month	1.9k	426	same day	256

Table 2.4: Updated FL Framework Comparison

Table 2.4 shows the updated FL Framework comparison (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). Its goal was 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. We tried to replicate his experiments, but his provided code [58] lacks instructions on how to set up this benchmark application. We simulated the experiments with the latest official flower version at the time and made sure to stick as closely 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 [55] or [27]. [55] is the latest work that compares FL frameworks that we considered, and its verdict is that Flower outperforms all its competition. FLOps uses Flower as its FL framework of choice.

2.1.6 Flower

Flower [6] is a research-backed open-source FL framework. One major target in Flower's paper was to narrow the gap between research and production. It allows researchers to run high-performance FL simulations and rapidly transition to tangible production environments using the same tool. Another focal point of its paper was scale and parallelization.

Flower is a sophisticated, feature-rich FL framework. 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]. 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. Further features include

pre-implemented popular FL algorithms and vertical and horizontal data splitting support. Flower supports traditional ML tasks, like regression or clustering, DNNs, LLMs, and security mechanisms, such as secure aggregation. It enables the use of FL via CPUs or GPUs. The default communication protocol is gRPC, which can be replaced. Flower supports various FL variants, including PFL, edge computing, cross-silo, and cross-device. It 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.

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 and the minimum number of learners required for training or evaluation. Users can pick from more than 30 existing strategies [4] or extend from basic strategies and develop their own behavior. 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 features and strategies based on the available basic Flower components.

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. In conclusion, Flower provides a rapid and easy onboarding experience.

Flower has a modern, user-friendly, growing ecosystem. A dedicated sub-project called Flower Datasets [19] is part of this ecosystem. This project is still in its infancy (v0.3.0). It allows users to pull HuggingFace [29] datasets easily and split them into FL-optimized data fragments. Users can configure how to split this data up. Flower Datasets can turn standard non-federated homogeneous/IID datasets into challenging, federated, non-IID data, which is ideal for FL research and development. This ecosystem includes a well-structured and rich homepage [23], an extensive set of tutorials, guides, example projects [21], and documentation [24, 20] 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, yearly summits, a blog, a discussion forum, a Slack space, and a YouTube channel. Our experience working with Flower and engaging with the community and developers has been positive.

2.2 Machine Learning Operations

Investigating modern best practices is necessary to improve and benefit the field of FL. 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, many contrasting and interconnected disciplines have emerged. They must cooperate smoothly to develop, deliver, and operate modern software. These tasks can be grouped into two broad categories: development and operation.

2.2.1 DevOps

The history of software development shows an evolution from static and isolated to dynamic and intertwined workflows. 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. There was a massive increase in modern requirements for flexibility and the ability to 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. They build distributable software artifacts based on their source code for distribution among clients. 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, software development loops are rarely linear sets of steps. 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 receives a feature update. These dynamic and strong dependencies require developmental and operational tasks to work tightly together. This coupling also applies to IT professionals who must cooperate and understand each other's areas well. This combined effort has become its own broad disciple called DevOps.

The synergy between development and operations created new techniques, tools, and professions. This combination includes various tasks such as building, deploying, testing, and monitoring. Automation is one core activity in this connected discipline 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. Building artifacts or container images, orchestration, or knowledge sharing can all be considered part of DevOps. This notion makes 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 [34]. This field is still very active and rapidly evolving. Unfortunately, many other disciplines are not taking inspiration from or taking advantage of DevOps.

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 [34] 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 [45] 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 [44] accumu-

lated 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 [42], code examples [43], 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 [25]. MLflow helps users manage their ML workflow loops from conception to re-deployment.

MLflow divides its core features into multiple different interconnected components. These components are rather conceptual groupings of functionalities than concrete isolated interfaces. The following represents a major selection of critical components.

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, and 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 [40]. In this example, all necessary dependencies are explicitly mentioned

and installed in a custom Dockerfile that needs to be built 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, which 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 techniques have been rapidly evolving for the last decade. To avoid issues due to different environments and dependencies, containers arose as a lightweight alternative 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. These container groups 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 dynamic swarms

of containers needed additional management tooling for orchestration. Kubernetes, the most prominent orchestrator to date, was released in 2014. Since then many new endevours formed to unify and streamline future developents in the field. Examples include the Open Container Initiative [49] or the Moby Project [46]. 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. FLOps aims to automate and orchestrate FL workloads on distributed heterogeneous machines. It is crucial to confirm that doing FL/ML via orchestrated containers is viable. In addition, FL processes should not suffer from bottlenecks and problems due to running inside containers. In 2017, Xu et al. [63] 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. [51] analyzed more than 400 different open-source ML-based projects that use docker containers. These projects used containers for various tasks. A popular application of containerization technologies in ML is to streamline and improve deployment efforts. This work demonstrates that containers are used in all ML-related tasks nowadays.

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 primarily focuses on enabling practical FL on real machines. The main target group for cross-device FL is 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 outperforms 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.
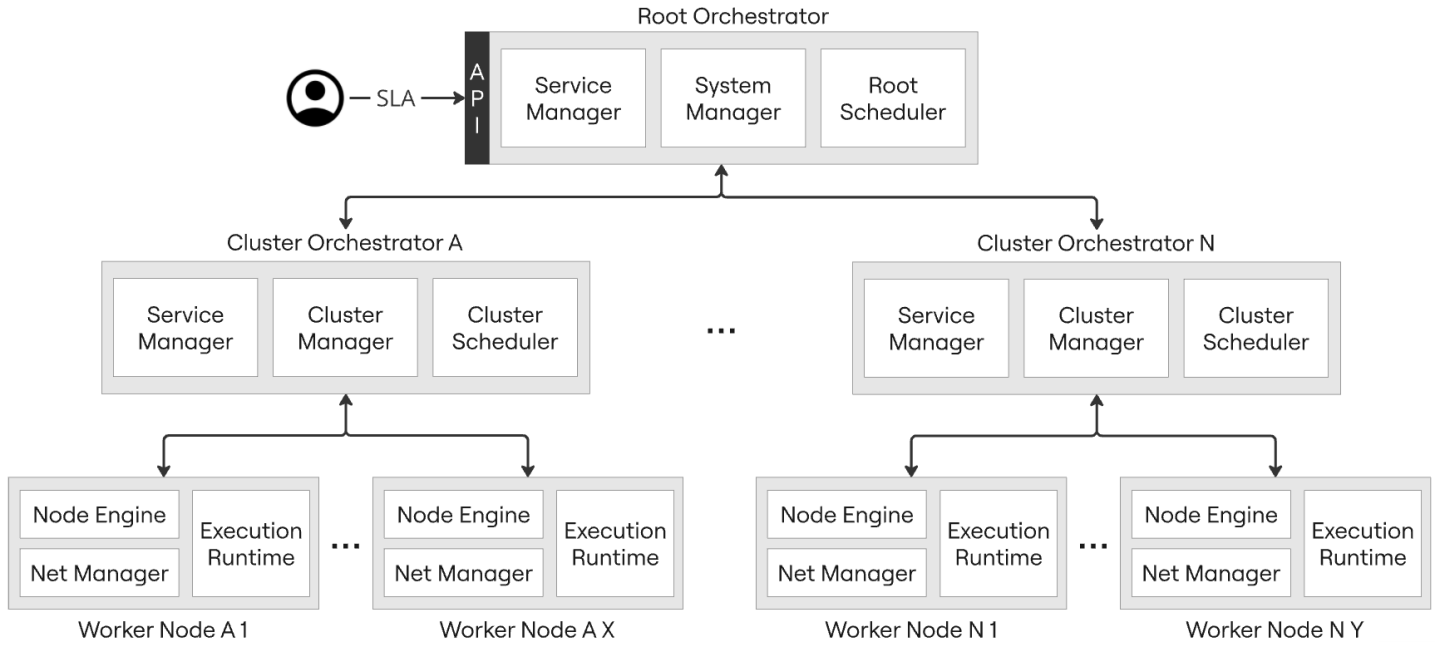
Figure 2.14: Simplified Oakestra Architecture

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 specializes in handling 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 can only access detailed information about workers from their own cluster. The metrics they share with the root are distilled and no longer contain sensitive individual metadata. This is an ideal environment for FL because this layout supports privacy on a structural level. FLOps uses Oakestra to orchestrate its components.

2.4 Related Work

Only two previous works [12, 56] mentioned in 2.1.4 resemble FLOps. Both also noticed the lack of research regarding dynamically deploying ML and FL capabilities via containers. They use different technologies and offer different functionality compared to 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. [12] 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 sent out to clients. The oracle determines the required type of ML technique, such as classic ML or DL, based on the environment. 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. Additional 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. An 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 themselves 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 environmental changes. 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 their work and FLOps are as follows: 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. Their work's mini-servers and "root" server resemble Oakestra's root and cluster orchestrators. Oakestra is a dedicated orchestrator, while their work's components are auxiliaries with fewer features.

Their work used a different orchestrator, FL framework (augmented FedML), and image registry. FLOps supports classic and hierarchical FL. Their work only supports HFL. They used a single hardcoded dataset and ML model for evaluation. Their work does not offer different scenarios or utilize dedicated MLOps features and techniques. FLOps allows users to build and train various custom ML code. Their work focuses on 6G and the 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. [56] developed a prototype to improve FL on IoT devices. This work enables distributed ML model deployment, federated task orchestration, and system state and model performance monitoring. They called their approach FedIoT. This three-layered architecture resembles the one from [12] 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. Their work focuses on enterprise IoT. IoT devices are usually incapable of handling common ML training due to their limited resources. Their work acknowledges this and uses the IoT devices only as data providers but not as learners. Therefore, their work performs classic FL instead of HFL. Local orchestrators are learners in their architecture. They need to be in proximity to IoT devices to receive their data. Additionally, they provide customizable data preprocessing and evaluation code to be injected via the API.

Their work offers additional tooling, such as a custom compressor and monitoring.

The compressor is a dedicated component that reduces the size of large files. Monitoring agents are deployed on the local and global orchestrators that measure resources and CO_2. A custom GUI presents these metrics.

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

There are obvious similarities between their 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.

Their work differs from FLOps in multiple ways besides FLOps' larger set of features. As mentioned above, their work only offers classic FL and has a different and less mature architecture than FLOps, thanks to Oakestra. Their paper is very short and 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. Safri et al. implemented most components by themselves from the ground up, such as orchestration and FL. FLOps utilizes and combines existing sophisticated solutions to offer higher-quality features and performance. For example, their 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 graphical suite of MLOps tools and functionalities.

3 Requirements Engineering & System Design

This chapter performs requirements engineering and presents system models. These models illustrate how the system works and fulfills the found requirements.

Usually, the requirements engineering and system design parts are split into different chapters. The requirements part defines and formalizes the reasons, needs, and problems of a system. It focuses on the application domain, which represents the proposed system and its surrounding environment. Its goal is to explain the new system's workflows, processes, and structural relationships without delving into concrete ways of realizing these goals. It is crucial to understand the basic behavior, reasoning, and environment of the system before working out how to realize these goals in a tangible manner. Conventionally, this part also includes system models. System models consist of object models that describe the system's structure, functional models that show its available functionalities, and dynamic models that depict how components interact. The system design explores an existing or proposed solution for these requirements. It is the first and most abstract part of the solution domain. Besides the system design, the solution domain covers how specific objects are designed, considering data structures and algorithms, down to concrete implementation and testing of the codebase. [9]

We combine both into a single chapter because of the following reasons. FLOps is already an implemented system. There is no need to discuss and illustrate significantly simplified possible solutions for the system first and then do the same more concretely for the existing system later. This would lead to models and explanations with many commonalities and redundancies and bloat this work. The existing system satisfies the same requirements and is still explained in a simplified way via abstractions. For very detailed implementation the next chapter and the codebase is available [17]. The system design part directly follows and relies on the requirements engineering part [9]. Many aspects are interconnected and can hardly be assigned to one another, especially when discussing an existing system. For example the system design concerns itself how the system handles concurrency, software control, and global resources [9]. Usually, these aspects are described via dynamic models, which directly correspond and depend on concepts discussed in the requirements analysis [9]. Therefore, both use models to describe how the system works and satisfies the requirements. This chapter covers both

aspects where the initial sections cover the application domain and the last ones the solution domain. Intermediate sections feature aspects of both. As a result this chapter enables flexible discussions logically leading through both domains.

3.1 Requirements Elicitation & Specification

It uses the problem statement 1.1, objectives 1.3, and weaknesses found in the FL field 2.1.4 as input to elicit requirements. These requirements are specified, analyzed, and concretized. This chapter starts by analyzing and deriving functional and nonfunctional requirements. It utilizes scenarios, use cases, and (UML) models to explain the proposed system's functionalities, dynamics, and structure. The contributions section 1.4 provides an overview of FLOps's functionalities and features and how they realize the discussed objectives. This chapter follows the Requirements Analysis Document Template by Brügge et al. [9].

3.1.1 Functional Requirements

Functional Requirements (FR) describe mandatory functional relationships between the proposed system and its surroundings. These requirements only focus on concrete functionalities, user interactions, and environmental conditions. They ignore implementation details and nonfunctional conditions, such as performance. FRs capture what a proposed system must achieve instead of how it achieves it. [9]

FR-1 **Federated Learning**

    FR-1.1 **Enable individuals to use, develop, and evaluate practical FL**: FLOps is a platform where individuals can utilize FL regardless of their level of expertise. Target groups include inexperienced and expert users, developers, and researchers.

    FR-1.2 **Automate FL management & processes**: FLOps automatically handles all necessary duties to perform FL for the user. These duties include providing, creating, (un)deploying, and removing FL components, such as learners and aggregator(s). FLOps starts and stops the training and evaluation processes.

    FR-1.3 **Support various flexible FL scenarios**: Besides classic FL, FLOps supports (clustered) HFL. FLOps is ML library/framework agnostic, allowing different ML techniques, such as DNNs and classic ML.

FR-2 **Provide flexible configuration**: FLOps supports different FL project configurations. For example, users can specify and request different resource requirements,

such as the number of training rounds, FL algorithms, and the minimum number of learners.

FR-3 **Handle FL augmentation and containerization**: FLOps automatically converts user ML code into FL-capable container images that include all necessary dependencies to do FL. It stores these images internally and deploys them for training.

FR-4 **Provide a GUI for monitoring, evaluation, and result management**: FLOps provides a sophisticated GUI for monitoring, comparing, storing, exporting, sharing, and organizing training runs, metrics, and trained models. Users can access this GUI at any time. They can follow live training progress or inspect their previous results.

FR-5 **Provide trained model access to users**: FLOps enables users to access their trained models. FLOps can build container images that serve the trained models. Users can pull these images to use their trained models as they wish. Users can see and access their models via the GUI.

FR-6 **Enable inference serving**: FLOps can automatically build and deploy inference servers based on trained models. Users can send inference requests to their trained models directly after training on the same platform.

3.1.2 Nonfunctional Requirements

Nonfunctional Requirements (NFR) define how the proposed system should work. NFRs include constraints that the system must fulfill. The following NFRs are based on the established FURPS+ categories, as seen in [BD09]. These NFRs mainly represent groups of requirements instead of intricate individual requirements to reduce bloat.

NFR-1 **Usability**: FLOps should automate and streamline FL, MLOps, and orchestration processes. Thus, it should allow users without specific experience in these domains to perform FL and benefit from these technologies. FLOps should accelerate FL development and evaluation. Therefore, it should save time for FL experts by automating away redundant manual tasks.

NFR-1.1 **Effortless FL Participation**: Users should be able to participate and initiate FL projects by simply providing a link to their (non-FL) ML code.

NFR-1.2 **Prepared Reusable Components**: FLOps should provide ready-made, extendable, multi-platform components to automate development and evaluation workflows.

NFR-1.3 **GUI**: FLOps' GUI should follow established conventions for usability. Upholding these conventions is essential to ensure intuitive information visualization and easy navigation.

NFR-1.4 **CLI**: Its CLI should provide a clear and complete set of commands to interact with the system. Each key user functionality should have a corresponding command, such as creation, inspection, and termination. The CLI's monitoring capabilities should be comprehensive and close to real-time (less than 5 seconds delay) to provide users with timely information (Performance Requirement). The CLI should provide users with thorough, supportive instructions and help messages to improve onboarding and general use. It should be possible to start and stop FLOps FL projects with a single request each.

NFR-2 **Supportability**:

NFR-2.1 **Maintainability**: FLOps should make extending and modifying it straightforward and comfortable to ensure long-time developers and occasional contributors can work on it efficiently. The target group should include FL researchers who might have little experience in writing high-quality code.

NFR-2.1.1 **Codebase**: Its codebase should follow industry best practices for readability and maintainability. Its system design should be modular and extensible, allowing for easy updates and additions of new features or integrations with other technologies. Therefore, FLOps should uphold high-quality code standards and use state-of-the-art libraries and frameworks.

NFR-2.1.2 **Quality Enforcement**: FLOps should enforce formatters, linters, and automatic CI pipelines to verify high code quality.

NFR-2.2 **Portability**:

NFR-2.2.1 **ARM & AMD Support**: FLOps should support AMD devices, which are primarily used for development, and ARM devices, which are mainly used in edge devices.

NFR-2.2.2 **Generic Interfaces**: To reduce vendor-lock-in and hardcode, FLOps should prioritize generic interfaces. Therefore, It should be able to integrate with existing or future tools and frameworks.

NFR-3 **Performance**:

NFR-3.1 **Scalability**: FLOps should handle dynamic workload increases without significant degradation in performance. It should be able to scale across

different hardware and network conditions. FLOps should be able to support a large number of client devices and training rounds. It should be able to manage and orchestrate multiple FL tasks simultaneously.

NFR-3.2 **Availability**: FLOps should provide reliable FL training by implementing robust error handling and recovery mechanisms, ensuring that the system can recover gracefully from failures during training or deployment. FLOps should ensure high availability and fault tolerance during FL training and orchestration.

NFR-3.2.1 **Error Handling**: FLOps should handle errors gracefully and provide meaningful error messages. Gracefully means that errors are caught and resolved if possible. Fatal error should allow the rest of the system to continue working. Exceptions and error messages should be concise. They should inform users about the concrete error and its context in a couple of sentences instead of a wall of text or call stacks.

NFR-3.3 **Optimized Image Building**: The system should create containerized images quickly and efficiently. These image-building processes should not burden the control plane or user resources to avoid bottlenecks. They should adhere to best practices for speed and lightness. These images should not contain unnecessary content, such as dependencies, code, or layers, which would bloat their size.

NFR-4 **Security**: The system should ensure secure communication between client devices and the central server. It should protect sensitive data during FL training and deployment. Otherwise, the entire premise of FL gets broken, and legal issues arise.

NFR-5 **Constraints**:

NFR-5.1 **Packaging**: FLOps' components should be able to run on different environments. The automatically built container images should be compatible with dominant technologies like Docker or containerd to maximize coverage and compliance. The built container images should support multiple platforms, at least AMD and ARM.

NFR-5.2 **Implementation**: FLOps should not implement all its features and components from the ground up. This is necessary to avoid the risk of subpar quality and optimize development and maintenance time and resources. It should use and benefit from existing state-of-the-art solutions and technologies.

3.2 System Models

After eliciting and specifying requirements, this section presents scenarios and analysis models representing how the system satisfies these requirements. These models aim to improve the comprehension of FLOps on the application level instead of the solution one. The models do not depict concrete implementation. Instead, they show a simplified conceptual representation of FLOps' architecture and workflows. This includes involved components and their relationships. These are not identical to the actual implementation. The goal is to improve comprehension of the system instead of showing overwhelmingly verbose intricate details that might change in future updates.

System models aim to build what is called an analysis model. The analysis model has three distinct parts The scenarios and use cases form the functional model. Class and object models make the analysis object model. State machines and sequence diagrams create the dynamic model of the system. [9]

3.2.1 Scenarios

The goal of scenarios is to enhance the understanding of the proposed system by looking at a concrete set of common use cases [BD09]. These scenarios are also written in natural language to further increase the comprehension of the planed end user experience while working with the new system. Scenarios can also be used to build models. We differentiate between two kinds of scenarios. Demo scenarios that illustrate the achieved use cases of the new system. Visionary scenarios portrait an almost utopian experience that will not be realized by this thesis, but lay the foundation for future work.

Demo Scenario A: FLOps workflow for a FL newcomer

TODO rephrase this so it is maybe more university than highschool? - SC prof that is not specifialized in ML or FL

John is a computer science teacher at a highschool. He is new to FL, he has some experience in ML but no in MLOps, DevOps or automation. During the corona pandemic he had the change to try out different digital solutions and approaches to motivate and teach his students. He wants to figure out if there are patterns and correlations between specific approaches and the number and quality of completed homework. Because doing this analysis by hand is too tedious and time consuming he wants to use ML. Specific local device metrics such as how long students have programs open that are relevant for homework and at what time students make the biggest progress or how long each exercise took for students to complete are of his

interest. He creates a ML code repository where he defines the model architecture and how it should be trained. He verifies if his model training approach works by providind mock data.

He is well aware that he is neither legaly nor morally allowed to access this private student data. However, these patterns and correlations might make a significant difference for his students success. He read online that FL can overcome such issues. He wants to set this up by instructing his students to allow their devices to gather these metrics and to train local models based on this data. Therefore, John can get his global model and find patterns without breaking the privacy and trust of his students.

The students he works with have diverse setups for online homework. Some have might gaming PCs, other old or borrowed laptops, some savy kids use very unique devices. So he needs to to enable FL on different devices in a easy and non breaking way for his students to replicate at home or during CS class.

He want to use established solutions instead of creating a custom solution from scratch. He notices that there are several FL frameworks but no easy platform to do FL, until he discovers FLOps.

He installs docker on his students devices, starts a local python script to scrape the metrics he is interested in. Now student machines collect the correct data locally. He setsup a oakestra cluster at school and shows his students how to turn their devices into worker nodes.

After a week of collecting data he starts up his FLOps management components with a single command via the CLI. He sends a SLA request to his FLOps manager via the CLI. He assumes that only halve of his students have working connection to the orchestrator. He configures his SLA to have a minimum of 14 learners and a maximum of 28 learners. He wants to do a test run so he keeps everything else as is. Thus he will create a FLOps project that will perform classic FL for 3 FL rounds. Each round will trigger his ML model that will train 10 rounds locally. It total each learner will train for 30 rounds. As part of this SLA he provides a link to his github repository where he stored his ML code. He needed to adjust his code slightly to satisfy FLOps' requirements.

He sends of his requests and observes via the project observer that a new project has been started. He goes to the tracking server url shows in the observer. There he sees a modern GUI showing the first round of training. This first round uses 21 learners.

After 30 minutes the FL training is completed and he can see in the GUI that his model has now a 65% accuracy. John wants to have a higher accuracy. He starts another FLOps project that now performs 6 rounds instead of 3. He notices that the second time the training starts sooner because the relevant components have already been build and reused from the integrated image registry. After the training has completed john sees that the new model reached 87% accuracy. He wants to test it out straight away.

This time he added the optional post training steps in the SLA to automatically wrap the trained model as an inference server and deploy it in the orchestrator. He finds this inference server and sends a request to it to check if things work out as planned. The inference server returns his requests with the correct prediction and presents the first interesting pattern to him. Now John can continue finding new patterns and correlations to help his students to learn and him to teach better while preserving privacy.

Demo Scenario B

Visionary Scenario

TODO

3.2.2 Use Case Model

UML use case diagrams visualize the use of all significant available functions of a system from the user's perspective. They are based on the functional requirements. This means that every function, whether directly triggered by the user or an internal system function, that leads to observable changes and results for the user is depicted. Use case diagrams aim to showcase available and consequential functionality as seen by the user in a compact representation. [9]

Figure 3.1 shows the Use Case diagram for FLOps. The white use cases represent the functionalities the external users can directly trigger. The grey use cases are internal system actions that are directly visible to users or lead to visible results. They get triggered as a result of user actions. For example, the user knows that FLOps is performing FL training by inspecting different provided outlets, such as the GUI. FLOps tracks the training progress and results. These logged artifacts become incrementally visible to the user who inspects the GUI. Thus, the user knows that FLOps is currently performing FL training and logging. Use cases inside the GUI boundary are directly accessible via the GUI. The same applies to the API boundary. Other tasks are executed and accessible via FLOps combined with its orchestrator. Use cases that involve developing or modifying FLOps itself are not explicitly portrayed. The depicted User actor represents end users of varying FL expertise (FR-1.1). This actor includes FL developers and researchers. The core use case is starting an FL project which also includes configuring that project (FR-2). This activity starts a chain of events, such as building an FL-enabled container image (FR-3), creating and deploying the learners and aggregator(s), and performing the FL training (FR-1). During training, FLOps tracks the model and system metrics, which the user can monitor and evaluate in the GUI (FR-4). After training, the model can be containerized and deployed as an inference

Figure 3.1: FLOps UML Use Case Diagram

server (FR-5, FR-6). The user can access this trained model (FR-5) and request services from its inference server (FR-6).

3.2.3 FLOps Overview

From this section onwards, we introduce the structures and concepts of FLOps that satisfy the gathered requirements, scenarios, and use cases. Depicting and covering all FLOps aspects at once would require a steep learning curve, complicated images, and verbose explanations. Therefore, we avoid large singular models and explanations but describe and depict FLOps piece by piece, from coarse to fine-grained. This subsection shows an abstracted overview of FLOps' big picture and core elements. These simplified concepts are discussed in detail in the following sections throughout this thesis.
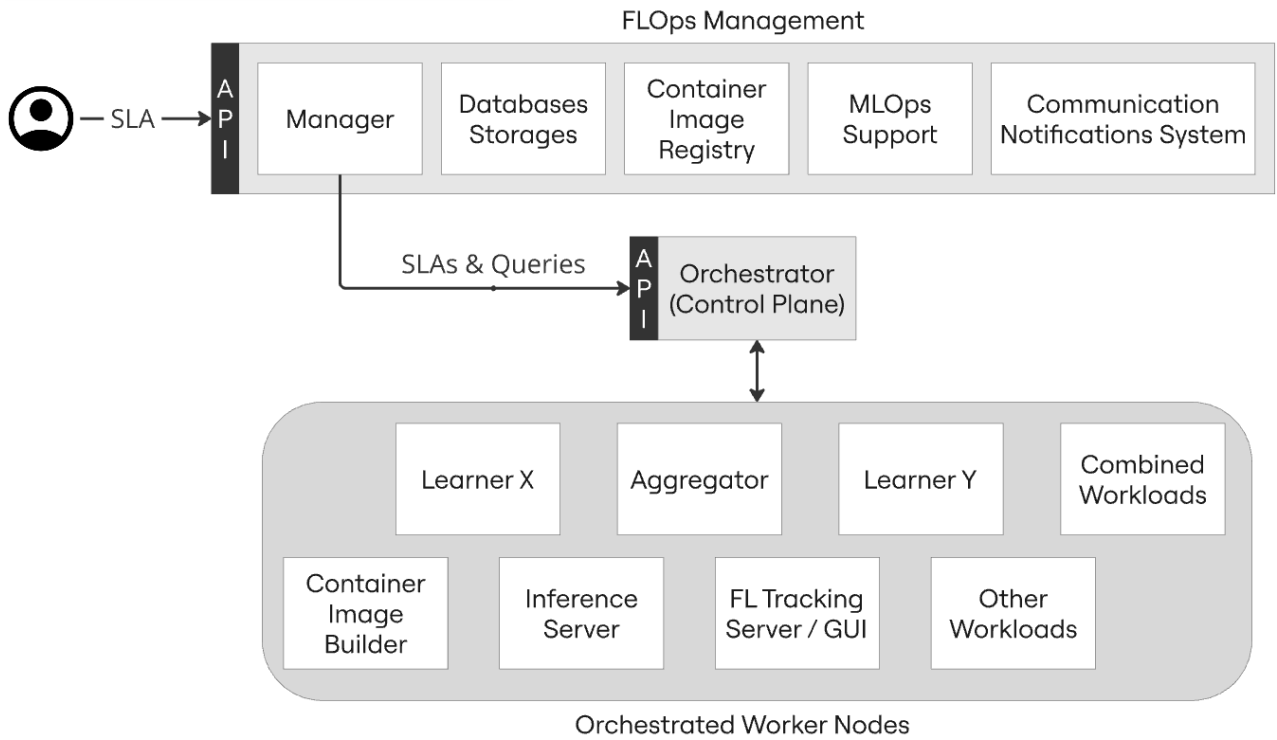
Figure 3.2: FLOps Structural Overview

Figure 3.2 provides a simplified overview of FLOps' structure. Note that the management components and deployed services on orchestrated workers are interconnected. These connections are not explicitly visualized to avoid clutter. The services shown in the rounded rectangle depict an arbitrary number of worker nodes. Multiple services

can run on the same worker, or every worker might only have a single service deployed. The FLOps system works via the interactions and relationships between the FLOps Management, the orchestrator, and the worker nodes. The FLOps management is a composition of components (containers). Its goals and responsibilities are to manage FLOps processes and store FLOps artifacts. The management components coordinate automatic processes and events. They store container images and training results such as metrics and trained models. These managerial components do not perform the FL training. They delegate and distribute computation to orchestrated worker nodes. The FLOps manager uses the orchestrator to create, (un)deploy, and remove different components. The manager spreads computationally heavy image builds and FL training across the worker nodes. The GUI and inference servers also run on worker nodes.
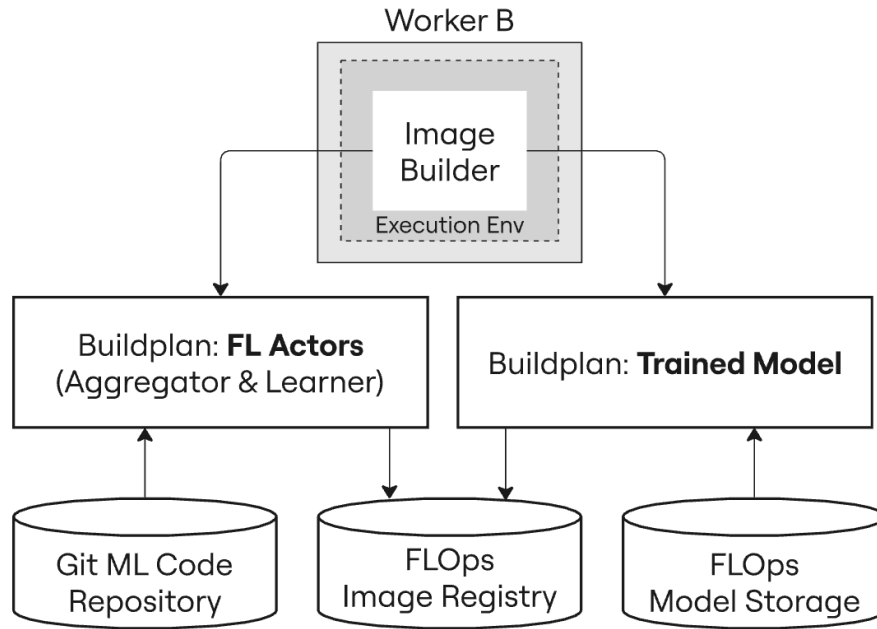
Figure 3.3: Simplified FLOps Image Builder Processes

Figure 3.3 shows a simplified overview of FLOps' image builder processes. The container images get built on worker nodes. Worker B stands for an arbitrary worker node capable of building images, i.e., the worker has enough resources and privileges. The build process occurs inside a container that requires special considerations. Remember that the user only provides ML code, not FL code. FLOps' image builder clones the user ML code, augments it to support FL, handles specific dependency issues, and builds multi-platform container images. This builder can build FL actors, i.e., the aggregator

and learner images, as well as an inference server for the trained model. These images get pushed to the FLOps image registry. When the learners, aggregators, or inference servers are needed, their corresponding images are pulled from that registry onto an orchestrated worker node and executed. Image builder services only exist when they are needed. The FLOps manager removes them after building images for a concrete FLOps project.
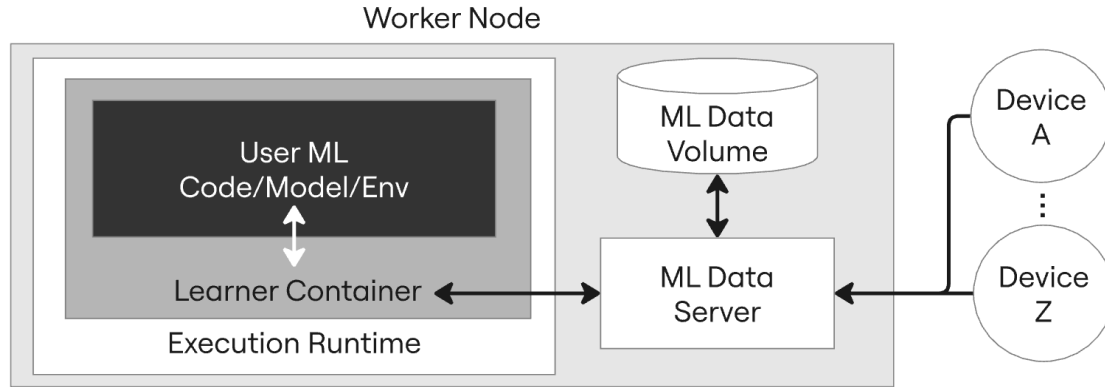

Figure 3.4: Simplified FLOps Local Data Management

Figure 3.4 shows a simplified overview of how FLOps manages local training data. FLOps targets practical, real FL applications. Thus, it does not expect users to provide data as part of their ML repositories. Instead, users need to coordinate with real data providers on the orchestrated worker nodes. The figure shows a deployed learner container on a worker node. The learner container itself has no data. FLOps cooperates with the orchestrator and deploys an ML Data Server before training on user-specified worker nodes. This data server is reachable by nearby devices via an API. Devices can send their data to this data server. The data server will store this data on the local machine. During FL training, the augmented learner container will fetch the local data via the data server. The augmented learner FL code forwards this local data to the user ML code for preprocessing and training. FLOps aims to support resource-restricted edge and IoT devices. They are usually not capable of handling demanding ML training. Letting them send their data to nearby edge/fog gateway devices capable of such tasks is possible. This idea is similar to the approach in [56].

3.2.4 Analysis Object Models

This subsection depicts FLOps' main components and their relationships in more detail. The following models are derived from and created to resolve the elicited requirements.

They focus on the user perspective. It is common to use generalized and abstract UML class diagrams to depict the system's main components, properties, and functions [9].
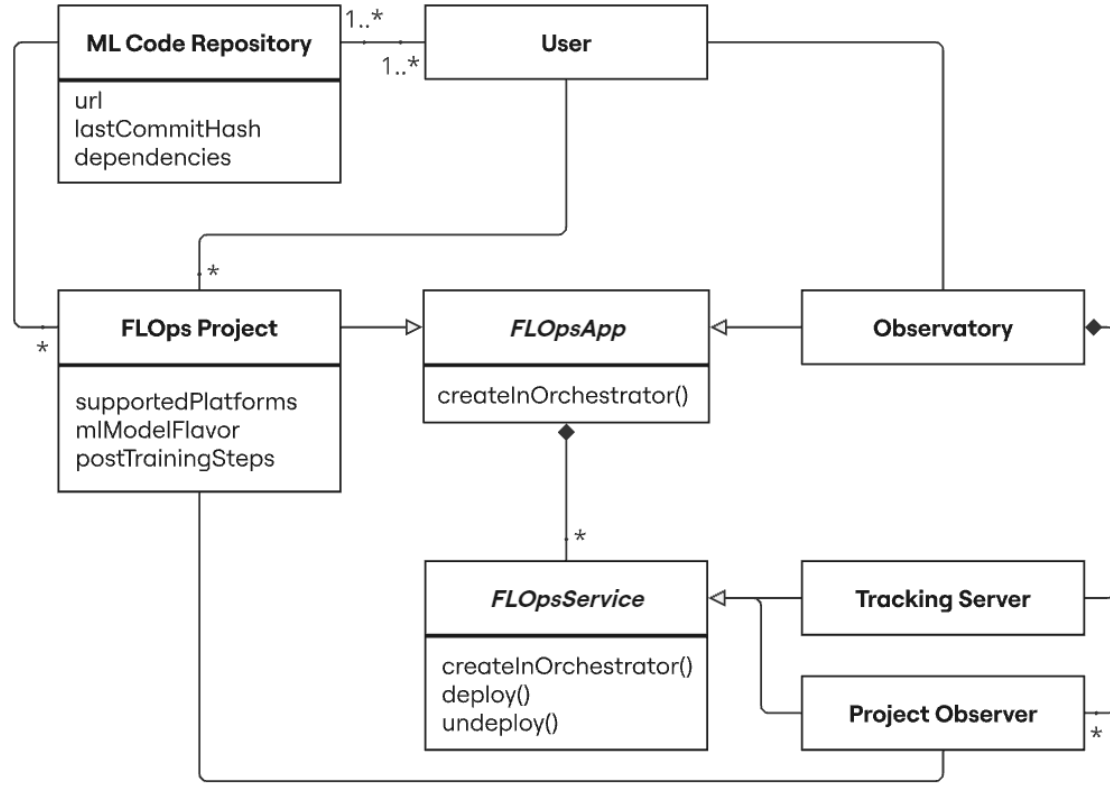

Figure 3.5: FLOps Core UML Analysis Object Model

Figure 3.5 shows the core FLOps UML Analysis Object Model. The main workflow is represented and grouped via a FLOps Project. Such a project links all necessary FL and ML/DevOps components to power one FL user request. A project contains information about the user who requested it, the target platforms that should be supported (e.g., ARM/AMD), and what steps FLOps should perform after training. If no steps are specified, the FLOps project counts as completed after training. Available post-training steps include building a containerized image for the trained model and deploying an inference server to serve the trained model. The ML Model Flavor indicator tells FLOps what ML framework to expect and work with. Examples include Keras, Sklearn, or Pytorch. Each project is associated with exactly one ML Code Repository. This repository can be owned by the user or be a public one. Thus, multiple users can reuse the same repository, and each user can create multiple FLOps projects per repository. These properties are based on the SLA from the user request.

FLOps uses the concepts of Applications and Services to manage dependent components and concepts. Each app can have multiple services. Services are bound to parent apps and cannot exist on their own. The orchestrator creates and realizes apps and services as usable components. Applications themselves are collectors of information and metadata. They do not run or contain any executable code, images, or similar. Services are the computational components that can be deployed and un-deployed. This split is based on Oakestra's applications and services. The two main FLOps app types are project-based apps and customer-facing ones.

The Observatory app is a customer-facing app. There is exactly one observatory app for each user, whereas users can have multiple projects. The observatory hosts the tracking server and project observer services. The tracking server service tracks the projects and individual FL experiments. It hosts the GUI. (It utilizes the MLFlow tracking server mentioned in 2.2.3.) When users request/start a new project, the observatory is created with all its components if it does not already exist. Users can request access to the GUI/tracking server independently from a project. A Project Observer service gathers and displays information or updates regarding the project status for the user. The project observer informs the user of any issues during the project's live time, such as dependency issues during the containerized image builds. There is one Project Observer per project to improve readability and comprehension.
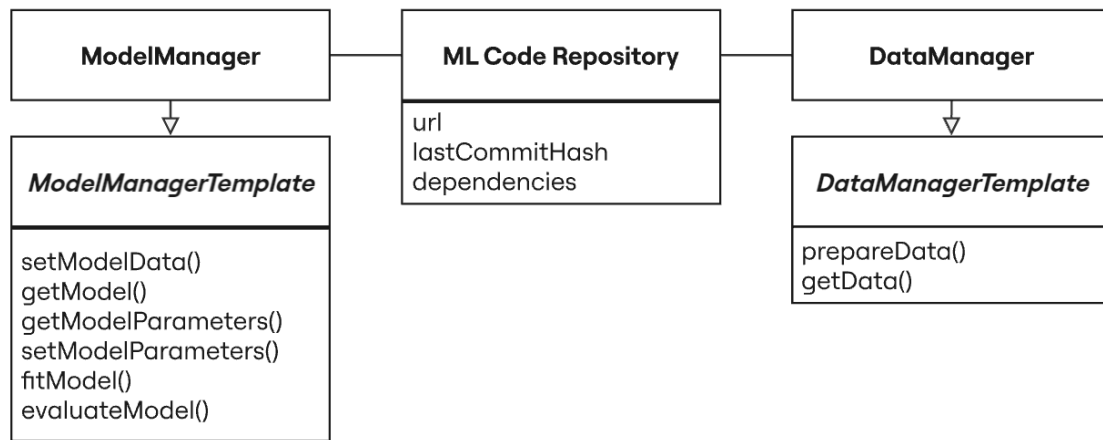

Figure 3.6: FLOps ML Code Repository UML Analysis Object Model

Figure 3.6 shows additional details of the ML Code Repositories from the core model. Users can provide a link to ML code repositories for FLOps to augment and train. The repository must fulfill the following structural requirements for this to be possible and straightforward. The repository needs a dedicated file that lists all necessary dependencies to train its model. Theoretically, it should be possible to extract these

requirements dynamically by inspecting the code. However, this is a complex and error-prone endeavor. To avoid these issues, users should provide the dependencies they used for training. We recommend running the training locally on some exemplary or mock data and recording the dependencies via MLflow's auto-logging functionality. This is an easy and viable approach to getting a suitable dependency file. Note that this does not guarantee compatibility because MLflow's dependency logging can be erroneous. Before providing the dependency file to FLOps, we recommend ensuring the dependencies are sufficient and compatible.

For FLOps to augment and utilize the ML code properly, FLOps excepts the repository to implement a model manager and data manager. The model manager is the interface that accesses the model and its data and parameters. It further trains and evaluates the model. It calls its linked data manager to prepare the data and retrieve it once it is ready. The data itself should not be part of the repository. The prepare data method will call a FLOps method that will be added during FL augmentation. The user has to define in prepareData how to pre-process the retrieved data for individual training. Both managers have an abstract parent class that users can import during implementation for guidelines. These templates are available as part of the FLOps Utils pip package [18].

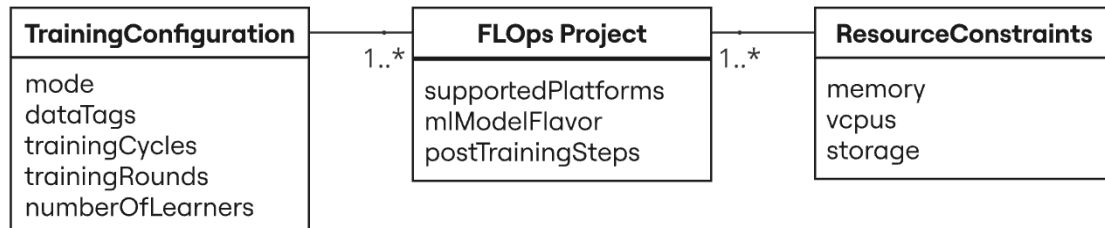| **TrainingConfiguration**	**FLOps Project**	**ResourceConstraints**
mode dataTags trainingCycles trainingRounds numberOfLearners	supportedPlatforms mlModelFlavor postTrainingSteps	memory vcpus storage

Figure 3.7: FLOps Project UML Analysis Object Model

Figure 3.7 shows further details about the contents of an FLOps project. Users can customize their projects via the SLA that is part of their API requests. One possible customization is to specify resource constraints such as memory or storage. Users can customize the FL training by changing the project's training configuration. The same ML repository can be trained differently depending on these configurations. This configuration includes a mode that tells FLOps to perform different types of FL if applicable. Currently, FLOps supports classic and (clustered) HFL. The project will only use training data that matches the provided data tags. The training rounds configure the number of training and evaluation rounds that each learner performs. Only HFL uses training cycles. The training rounds mean the number of training rounds performed on each learner per cycle. A training cycle is the number of training

rounds between the root and cluster aggregators, which resemble aggregators and learners in classic FL. For example, if the user requests three cycles and five rounds, the learners will train five rounds per cycle for three cycles. Each learner will train for 15 rounds during the entire project runtime. The depicted attributes are only a subset of currently available and possible configurations.
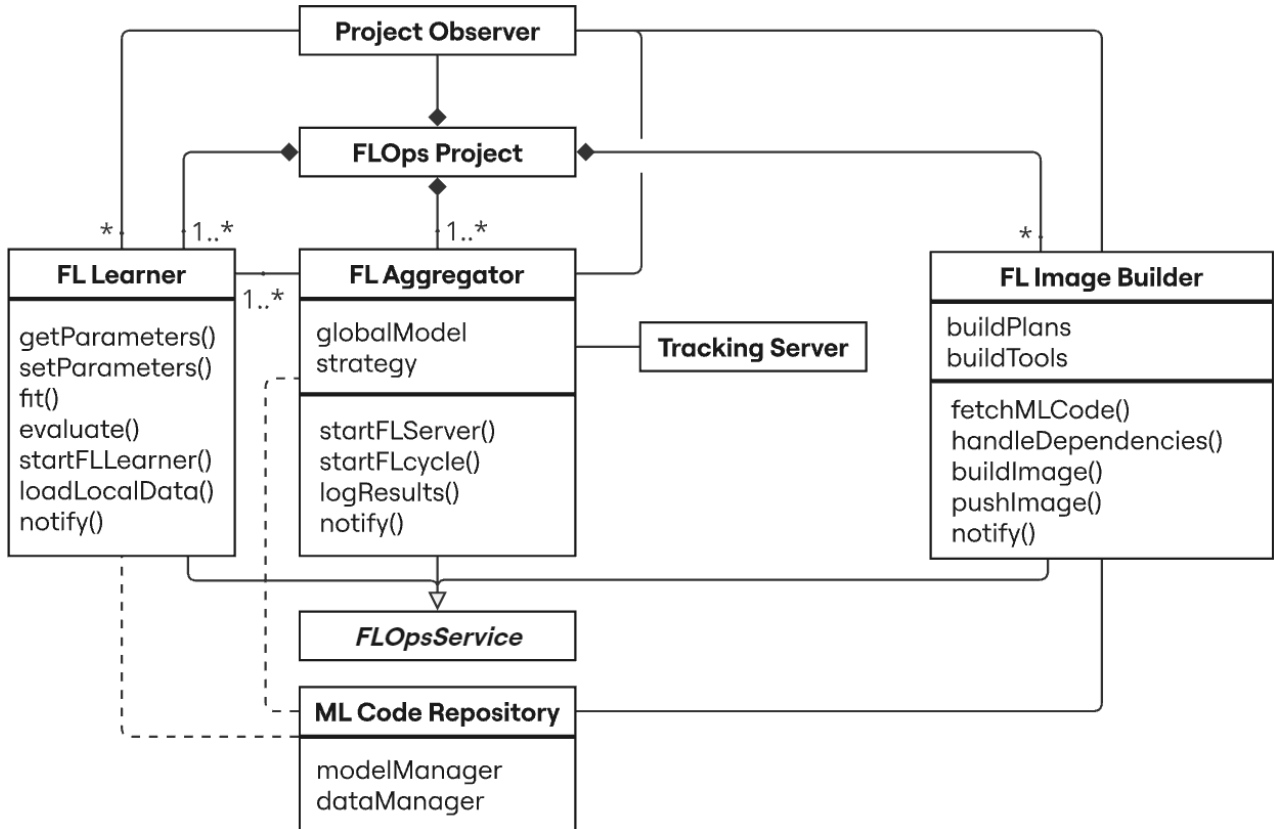

Figure 3.8: FLOps Project Services UML Analysis Object Model

The core figure 3.5 only alluded that a project consists of several services and depicted only the project observer. Figure 3.8 expands upon this and shows important project services and their relationships. There are three main project services. The FL Image Builder is a service that builds containerized images. It can build the FL augmented images for the learner, aggregator, and inference server of the trained model. Different buildPlans enable this distinction. The builder clones the ML repository, handles and checks the provided dependencies, builds the images, and pushes them to an image registry. During and after the builder operation, the service notifies other components,

including the project observer, about its progress, current state, and potential errors.

The FL Aggregator manages the FL training loop and holds the global model and strategy for training. It starts its internal FL server so learners can register for training. The aggregator starts and terminates learning rounds and cycles. It logs results like metrics or the final trained model via the tracking server. Similarly to the builder, it notifies other components during runtime about its progress and errors. The aggregator and learners utilize the code provided in the user's ML code repositories. They have direct access to the model and data managers. The image builder injects both of them.

The FL learners are project services that perform the FL training on local data. They fetch locally stored data, connect to the aggregator, and perform FL activities such as training. The learner uses the code found in the model and data managers and wraps itself around their implemented interface methods. As a result, users do not need to implement the FL (boilerplate) code themselves. Therefore, a learner's getParameters method uses the getParameters method described in the user's ML repository with additional logic around it. Learners also notify other components about their progress or failures.
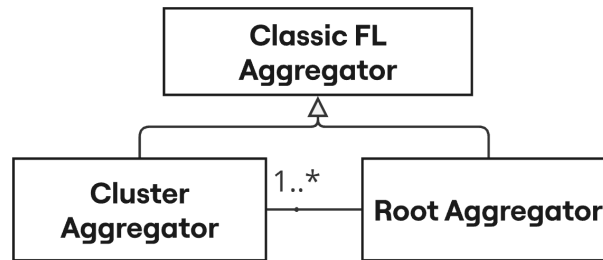

Figure 3.9: FLOps Aggregator Types UML Analysis Object Model

Figure 3.9 shows the simplified relation between different FLOps aggregator types. Because FLOps supports classic and hierarchical FL, it must support different aggregator types. For conventional FL, FLOps uses the classic aggregators. For HFL, FLOps creates a single root aggregator and one cluster aggregator per cluster, which are available in the orchestrator. The root orchestrator sees cluster orchestrators as plain learners. A cluster aggregator is a hybrid between an aggregator and a learner.

3.2.5 Dynamic Models

Dynamic models illustrate the dynamic behavior of the system and the interactions between its components. Different dynamic model types include UML activity, state chart, communication, and sequence diagrams. Because of the large number of components in FLOps, this section focuses on sequence diagrams to cover a large set of

significant interactions.

This subsection depicts the necessary interactions for a typical FLOps project. We use many abstractions and depict the absolute base case for an FLOps project to reduce complexities and improve readability. The base case aims to showcase a project workflow from the ground up. That means that no code, data, or images are present yet. This base case terminates after successful training. It has no post-training steps, such as building and deploying an inference server. This simple base case is already complex enough to require multiple UML sequence diagrams. For example, FLOps Management consists of multiple components. For simplicity's sake, these components are grouped into one. The same applies to multiple worker nodes and edge devices.
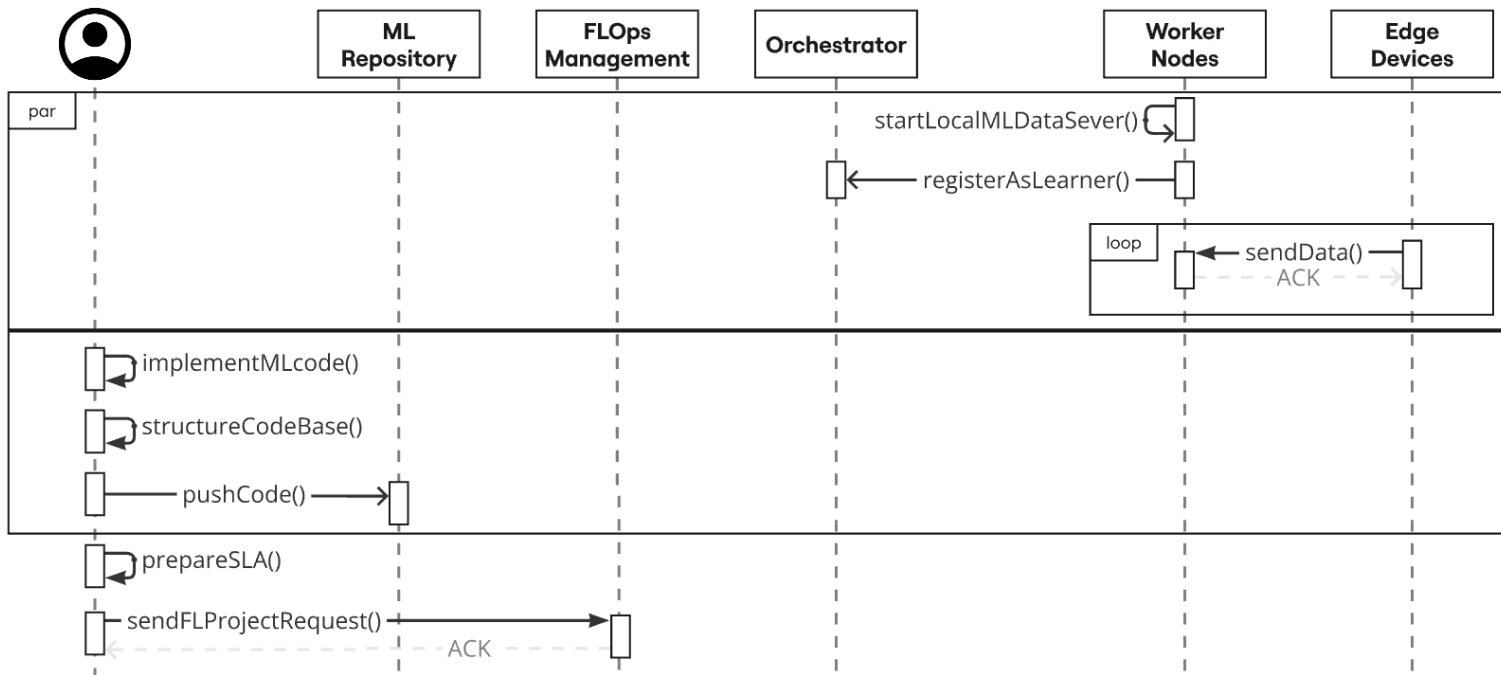
Figure 3.10: FLOps Preparation - UML Sequence Diagram

Figure 3.10 is a UML sequence diagram depicting the initial steps before a FLOps project can start. Two different sets of sequences can occur independently or in parallel with each other. Before FLOps can run properly, FL worker nodes need to register with the orchestrator and stay available. Only a subset of worker nodes are intended to be capable of performing ML training. Worker nodes that should participate as learners must inform the orchestrator accordingly and start the provided ML Data Server locally. This server accumulates training data locally for later training. This process has to occur before training begins. Otherwise, not enough or no data will be available for

training. Edge devices or other nearby data sources should send their data to these designated learner nodes. The infrastructure provider has to ensure that these worker nodes are appropriately protected and trustworthy. Once these steps are completed, FLOps should have access to available data-rich orchestrated worker nodes.

The second set of tasks that should occur before using FLOps is for users to prepare their ML code. They should implement their ML code and structure it as required for FLOps. This code must be available via an accessible git-based repository. If the user is satisfied with his codebase, he can prepare his SLA. (We discuss the SLA in detail in the implementation chapter.) After both sequences have been completed, the user can send his SLA to the FLOps Management API and request the start of a new FLOps project.

Figure 3.11: FLOps Project Start - UML Sequence Diagram

Figure 3.11 shows the main sequence of steps from starting a new FLOps project to deploying an image builder service. The light blue activity bars mean that FLOps creates a new object inside the manager context, independent of the orchestrator and deployed components. Management objects hold stateful information and are not run or used for computational workloads. This split is necessary for the FLOps management to keep track of ongoing processes, retain memory, and handle unique custom requirements independent of and unavailable in the orchestrator. Each colored or white rectangle on a lifeline inside the central UML sequence diagram corresponds to a specific functionality. Once this functionality terminates, the bar stops as well. Usually, sequence diagrams show concrete instances of classes as actors. We use abstract actors to optimize the page space and allow further abstractions and simplifications. For example, the lifeline of an observatory app inside the orchestrator could be a separate actor. However, this would lead to redundancy and verbosity between the orchestrator and observatory actors. Instead, these modified graphs show the lifeline of FLOps components on the right side. The color and dotted lines help to link those concepts together. Light rose orchestrator actions symbolize a service being appended to an existing application.

The FLOps Management registered the new project request and extracted the SLA. Firstly, the FLOps management creates a new observatory for the user inside the FLOps management context. The management requests to create a corresponding app inside the orchestrator. The same applies to the actual FLOps project. Once these two parent applications exist, the management will create the first service. The management requests the creation of the project observer as a service inside the observatory application. The management then requests to deploy an instance of this service on a worker node to the orchestrator. Now, the user can access this project observer service at any time to observe the progress of his project.

The FLOps management checks if its image registry already contains images that match the requested ML repository. For this, the management contacts the ML repository to check its latest state. In this example, the registry is empty. Therefore, new images are required. The management creates a new image builder service and deploys it similarly to the project observer.

Figure 3.12 shows the interactions necessary to augment user ML code into FL-enabled containerized images. Once the builder service is deployed, it executes the build plan for actor images. FL actors are the aggregator(s) and learners. The builder service notifies the management about a successful start of this process. It proceeds by cloning the ML repository and checks if it complies with FLOps requirements. The builder also checks if the dependencies are sound. Once these build prerequisites are met, the builder continues to build the aggregator and learner images. (Concrete details about this build process are available in the implementation chapter.)
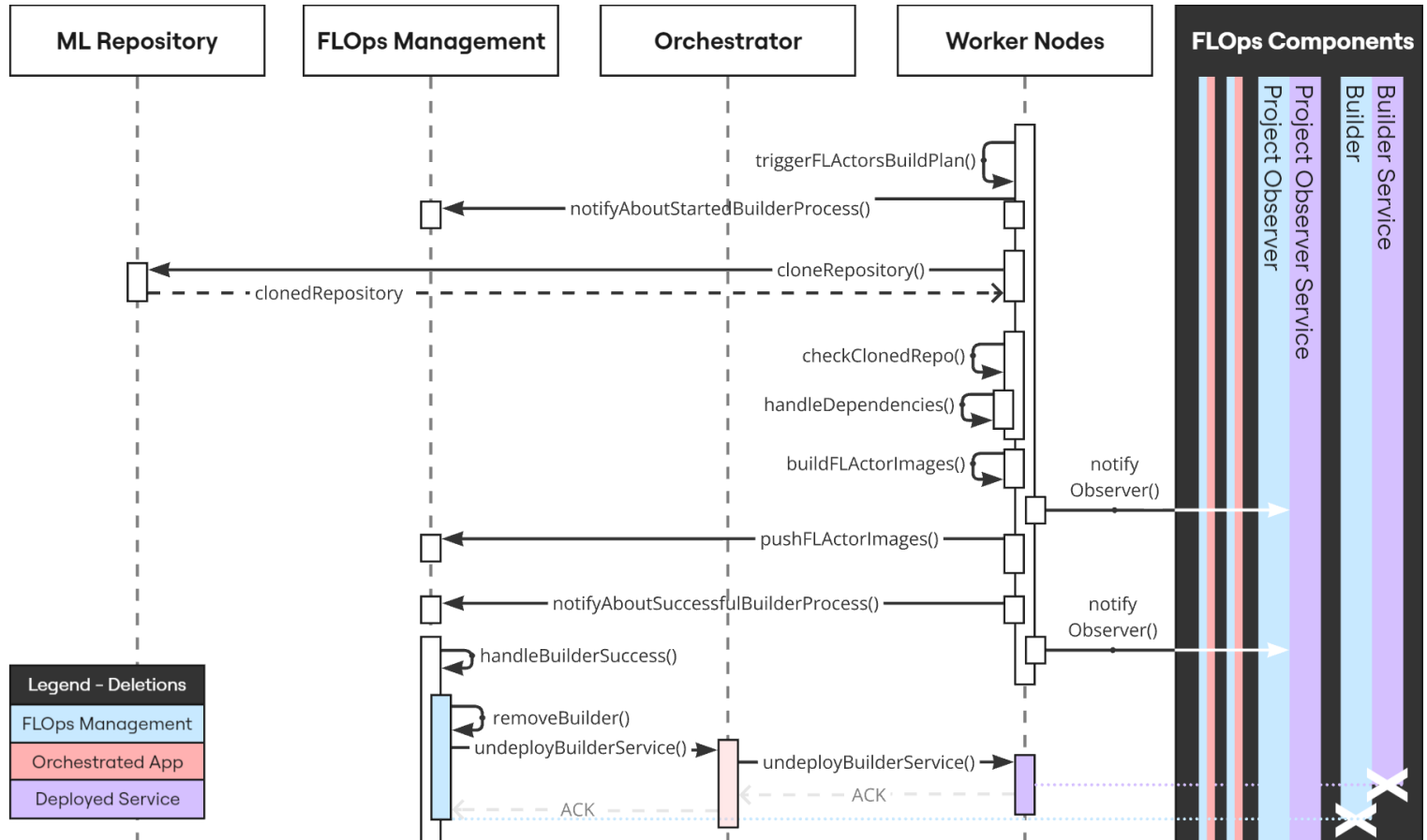
Figure 3.12: FLOps Image Builder Processes - UML Sequence Diagram

Afterward, the builder notifies the project observer to inform the user about the successful build. Note that the sequence model action lengths do not correspond to their actual duration. Short rectangles visualize the build and push actions in the diagram, even though these two actions are by far the most time-consuming. We use this design to ensure more compact diagrams. In a similar fashion, the observatory and project lifelines are reduced to lines.

As the last build process step, the builder pushes these built images to the image registry hosted by the FLOps management. After the successful push, the builder notifies the project observer and FLOps management of its successful completion. The FLOps management catches this message and removes the builder from its own context and undeploys it from the orchestrator and worker. Now that the FL actor images are ready, the training can begin.

Figure 3.13 shows the necessary interactions that realize FL training under FLOps. Note that the right side omits the previously depicted detailed FLOps component lifelines. We collapse these details for this diagram because the depicted components and their lifetimes show similar behavior as the previous diagrams. Arrows that point to the right mean that specific FLOps components are targeted. They are not explicitly depicted to optimize readability and reduce verbosity.

After building the required images, the FLOps management will start handling the FL training processes. Firstly, it notifies the project observer that FL training will start shortly. Secondly, it creates and deploys the tracking server that provides the GUI. Users can access this GUI by following the link shown in the project observer or directly accessing the deployed tracking service, similarly to the project observer. Note that there are several differences between the project observer and the GUI. The project observer is a minimalistic way to inform the user about the current state and potential errors during the lifetime of an FLOps project. The GUI is a standalone application that focuses on tracking the training results.

The FLOps management now creates and deploys the FL aggregator and learners. It uses the previously built images for this. Once the aggregator image is pulled and executed, it starts the FL server processes and notifies its watchers about its success. The aggregator waits for learners to connect before starting the training. In the meantime, the FL learner images were pulled and started. Each learner starts its FL client activities, such as registering with its specified aggregator.

Now that all FL actors are ready, the aggregator starts the first training round and triggers the learners. The learners train their models with the local data from their worker node. Once completed, the learners will push their model parameters to the aggregator. The aggregator fuses them into a new global model and returns the new global parameters to the learners. The learners apply those to their local model to be ready to begin the next FL training round. After each FL training round, the aggregator logs metrics, such as accuracy and loss, via the tracking server service. The FLOps management stores the logged results.

After the last FL training round, the aggregator notifies its observers and logs the final trained model via the tracking service. The aggregator only tracks the model once to avoid wasting bandwidth or storage. Afterward, the aggregator and learner activities terminate. Similarly to the builder's un-deployment process, the FLOps management registers the successful message and removes the FL actors. With this, the core FLOps project is concluded.

Similarly, FLOps realizes more complex configurations, modes, or post-training steps. For the post-training steps, the builder gets deployed once more. This time, it runs the trained model build plan and pulls the model from the FLOps management. It pushes the built image back to the management image registry. The inference service

gets deployed similarly to the FL actors using the built-trained model image.
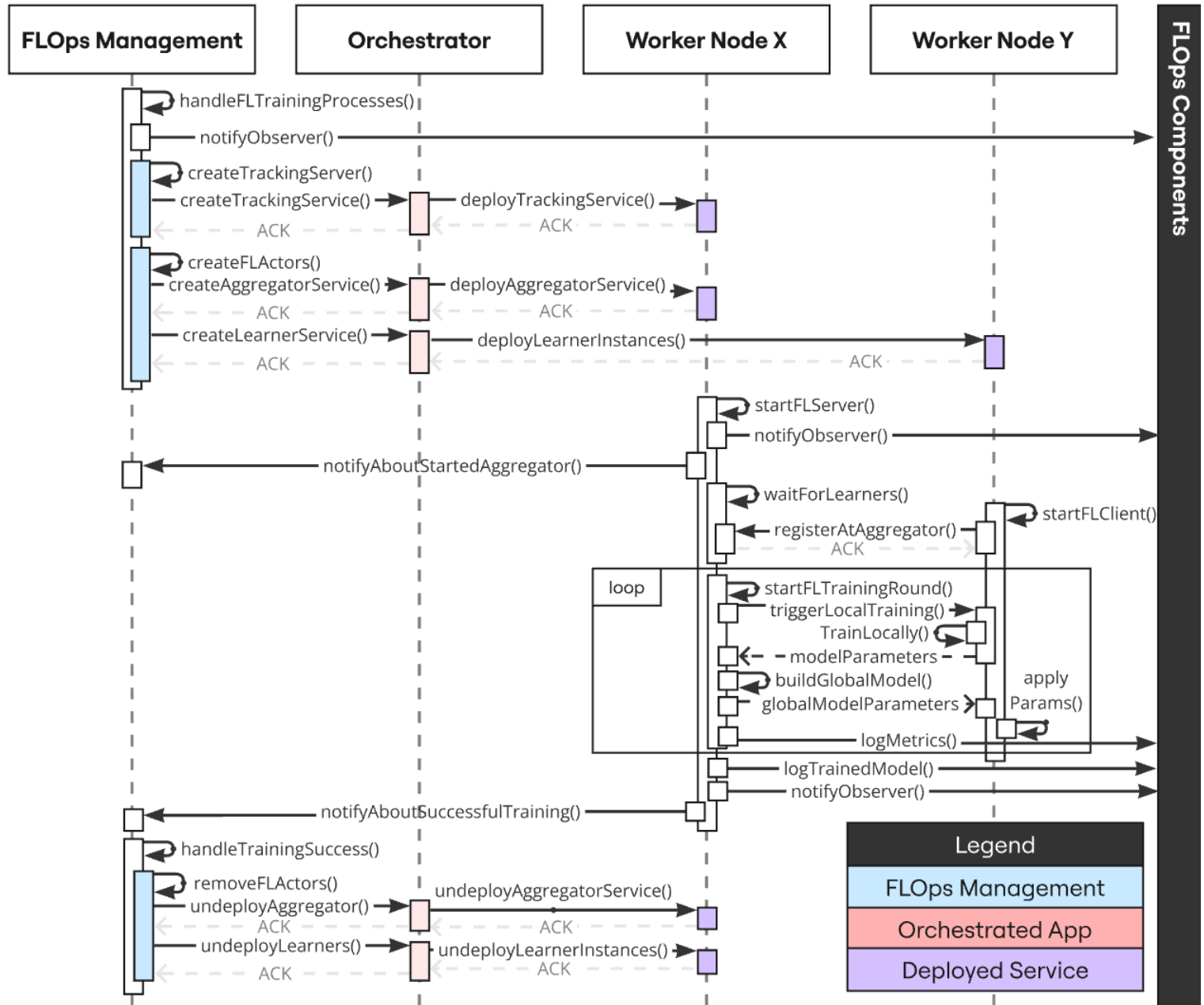
3.2.6 Component Diagram

3.2.7 Deployment Diagram

Figure 3.13: FLOps FL Training Processes - UML Sequence Diagram

4 Implementation Details

4.1 API & SLAs

4.2 Image Building

4.2.1 Dependency Management

4.2.2 ML Frameworks and their Sizes

4.2.3 Image Builders

4.2.4 Multi Platform

4.2.5 FLOps Image Builder Details

4.3 Local Data Management

4.4 MLOps via MLflow

4.4.1 MLOps Components & Architecture

4.4.2 GUI

4.5 Clustered HFL

4.6 CLI

5 Evaluation

5.1 Rationale

5.1.1 Chosen Experiments

5.2 Experimental Setup

5.2.1 Monolith

5.2.2 Multi-Cluster

5.2.3 Evaluation Procedure

5.3 Results

5.3.1 Basics

5.3.2 Image Builder

5.3.3 Different ML Frameworks/Libraries & Datasets

5.3.4 Multi-cluster & HFL

6 Conclusion

6.1 Limitations & Future Work

6.1.1 Federated Learning via FLOps

6.1.2 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] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136061257.

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

[11] 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`].

[12] 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`.

[13] 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`.

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

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

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

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

[18] *FLOps Utils Pip Package*. Accessed: 2024-08-26. URL: `https://pypi.org/project/flops-utils/0.5.2/`.

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

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

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

[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] 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/`.

[26]   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.

[27]   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.

[28]   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.

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

[30]   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].

[31]   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.

[32]   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.

[33]   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.

[34]   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.

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

[36]   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.

[37] 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].

[38] 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.

[39] 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.

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

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

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

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

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

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

[46] *Moby Project Homepage*. Accessed: 2024-08-20. URL: https://mobyproject.org/.

[47] 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].

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

[49] *Open Container Initiative Homepage*. Accessed: 2024-08-20. URL: https://opencontainers.org/.

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

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

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

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

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

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

[61] 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.

[62] 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.

[63] 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.

[64] 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].

[65] 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.

[66] 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.

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

Appendices