SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

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

Alexander Malyuk

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

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

TODO

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

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

Munich, 15.09.2024 Alexander Malyuk

Acknowledgments

Abstract

Kurzfassung

Contents

Abbreviations

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

Specific Acronyms :

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

Common Acronyms :

AI Artificial Intelligence
ML Machine Learning
DL Deep Learning
DNN Deep Neural Network
LLM Large Language Model
API Application Programming Interface
GUI Graphical User Interface
SLA Service-Level Agreement
CLI Command-Line Interface
IoT Internet of Things
P2P Peer-to-peer
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 [13]. 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) [81], or the California Consumer Privacy Act (CCPA) [57]. 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 [64]. 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 [25]. 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 [4] and Buildah [16] to manage dependencies and build images. FLOps utilizes a pioneering FL framework called Flower [37] to execute its FL training loops. The mentioned runtime observability features are available via a mature MLOps tool called MLflow [68]. 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 [10], 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 [33]. 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 [75]. 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

The following background chapter discusses vital concepts required to understand FLOps as a whole and in detail. It covers fundamental and detailed aspects of FL, such as the basic process, various architectures, frameworks, and tools. It analyses the research field of FL in great detail. The background chapter continues to showcase essential FLOps parts, including ML operations and orchestration. The concluding background section highlights related work.

Chapter three performs requirements engineering and showcases significant parts of FLOps' system design. Firstly, it elicits and specifies functional and nonfunctional requirements. Secondly, it discusses system models that depict the FLOps system and satisfy these requirements. This chapter aims to provide a comprehensive understanding of the system through its requirements and simplified architecture and processes without delving into underlying technicalities.

The fourth chapter analyses concrete implementation details. It starts by explaining how users can request workloads via FLOps' API and SLAs. It elaborates on the image-building processes in FLOps. It delves into the rationale, challenges, and internal solutions for building suitable containerized images. Afterward, this chapter explains how local data is managed on the learning worker nodes. Next, it showcases the

architecture of its MLOps components and how its GUI works. The penultimate section of this chapter discusses how FLOps realizes clustered hierarchical FL. The last section discusses the accompanying new CLI and how it makes working with FLOps and its orchestrator easier.

The evaluation chapter assesses the soundness and performance of FLOps. It compares different setups, experimental configurations, and their results.

The final chapter draws conclusions about the current FLOps project and implementation. It discloses FLOps' limitations and alludes to possible future work to improve the system further.

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 [10]. 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 [62]. 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

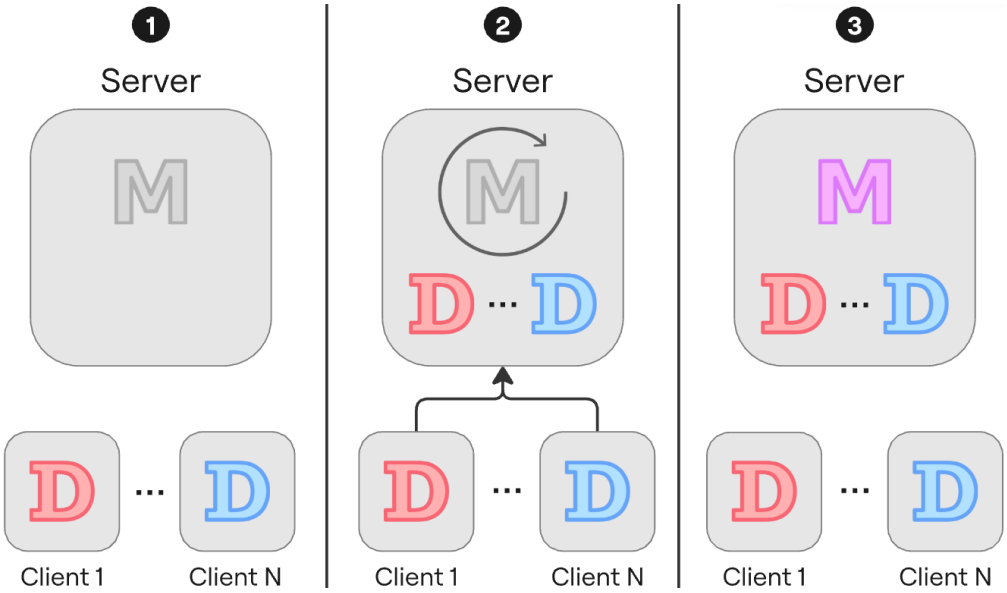

Figure 2.1: Centralized ML Model Training

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 [64].

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

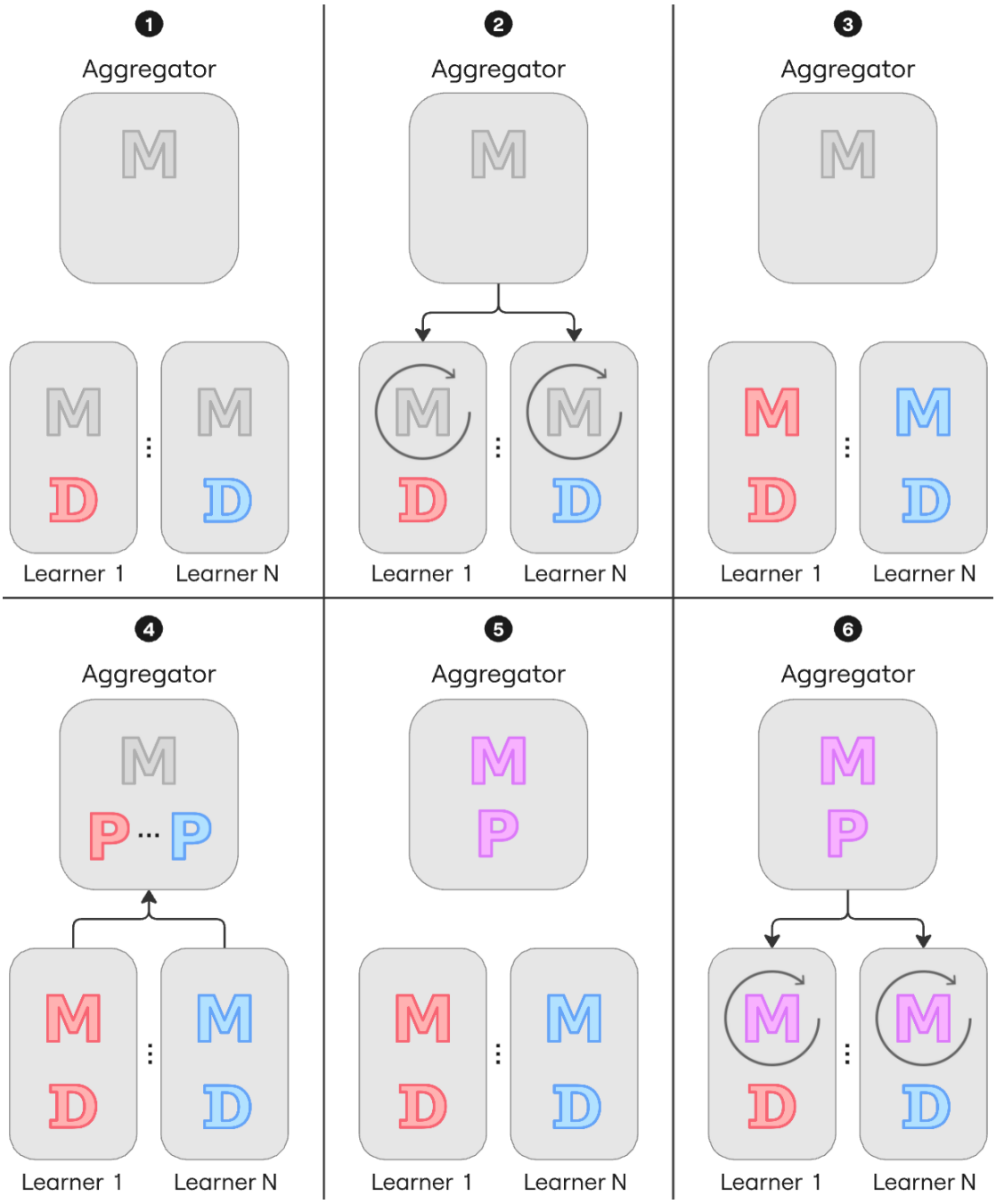

Figure 2.2: Basic Federated Learning

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 [62]. 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 [62]. 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 [62]. 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 [62, 106, 51].

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. [62]

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. [62]

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. [62]

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 [54].

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 [10] 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

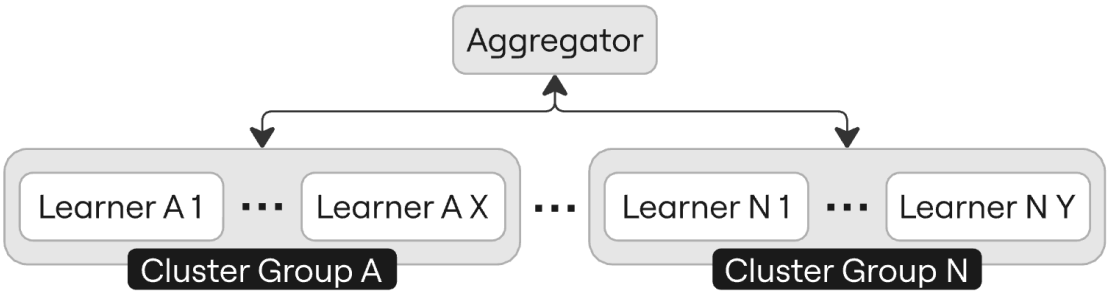

Figure 2.3: Clustered FL Architecture

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 [54, 19, 62, 59].

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

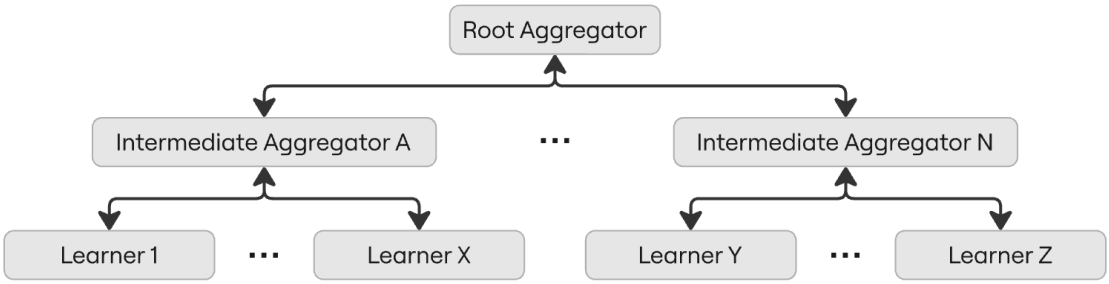

Figure 2.4: Hierarchical FL Architecture

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 [12, 105, 20, 59, 62, 106].

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. [62]

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. [62]

2.1.4 FL Research

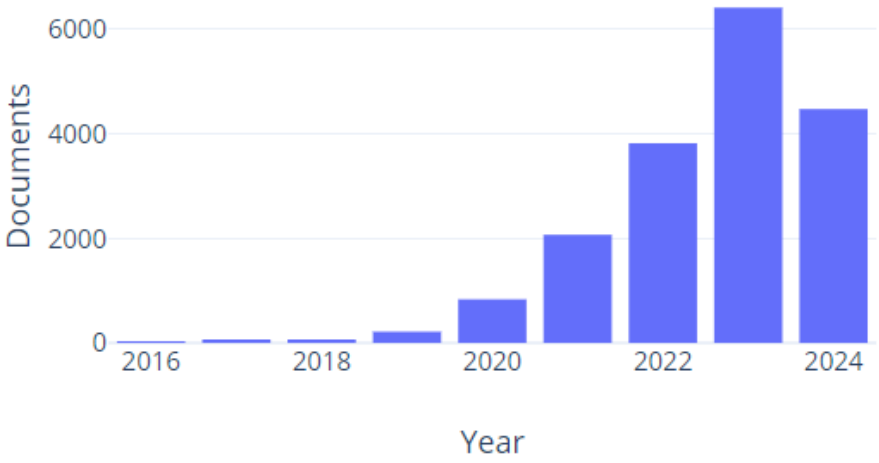

Figure 2.5: Evolution of FL Publications

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 [96].) The idea for this graph is based on [94]. 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.
[54]	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.
[107]	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.
[74]	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.
[59]	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.
[19]	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
[51]	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.	-
[102]	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.
[52]	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.
[17]	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.
[12]	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.
[105]	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.
[47]	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.
[44]	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
[20]	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.
[53]	Introduction of distributed adaptive FL model pruning.	Privacy and security were not considered. Further optimizations are possible, primarily focused on GPUs.
[88]	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.
[108]	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.
[106]	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.	-
[103]	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.
[61]	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

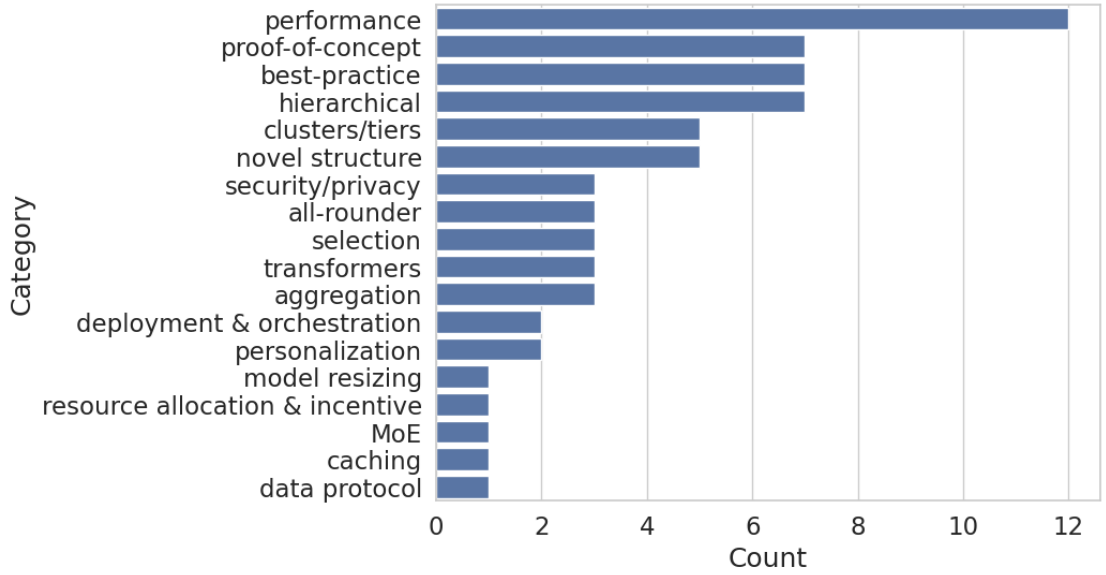

Figure 2.6: FL Paper Categories

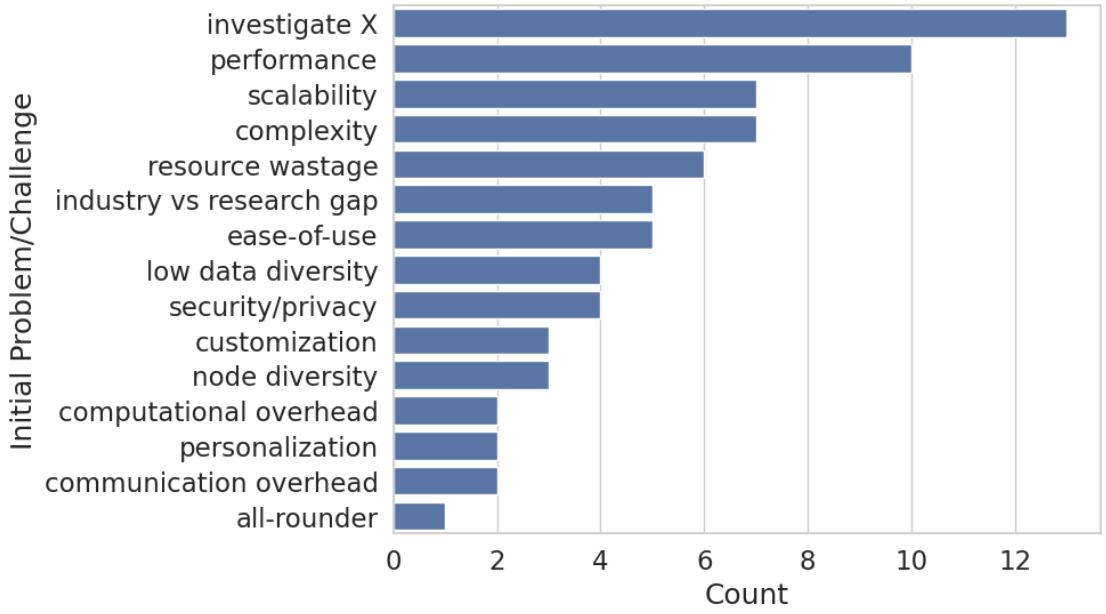

Figure 2.7: Targeted Problems & Challenges of FL Papers

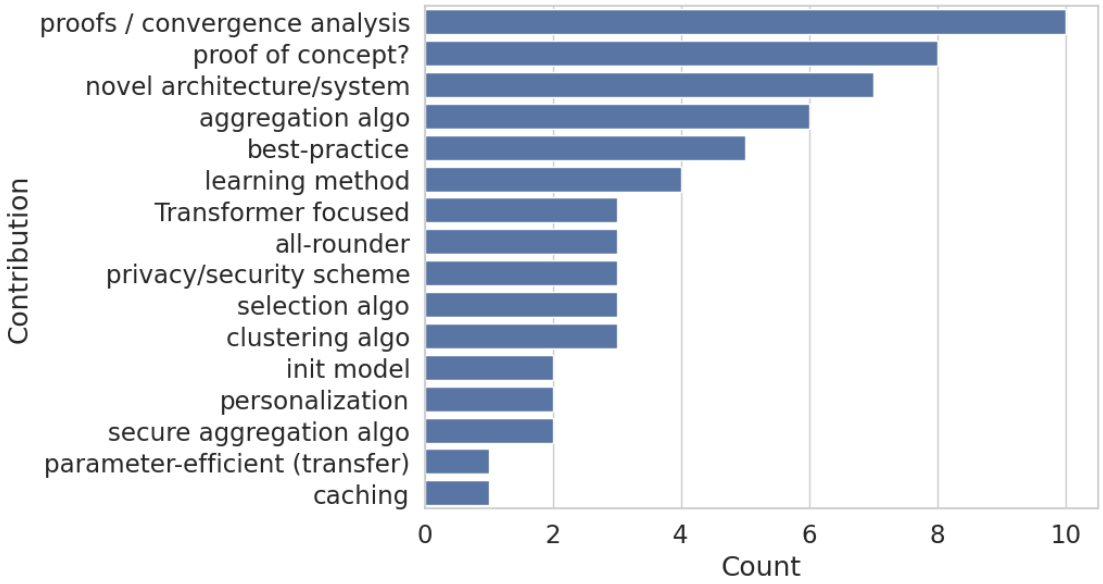

Figure 2.8: FL Paper Contributions

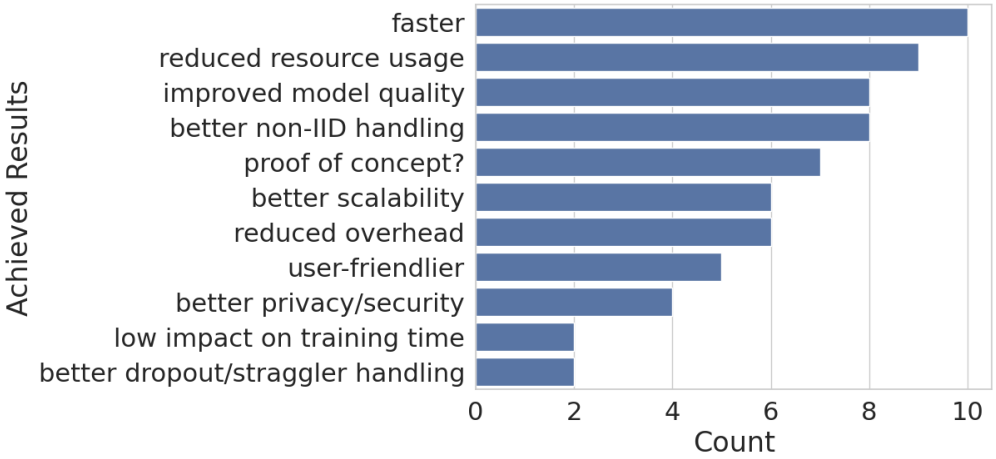

Figure 2.9: Achieved Results of FL Papers

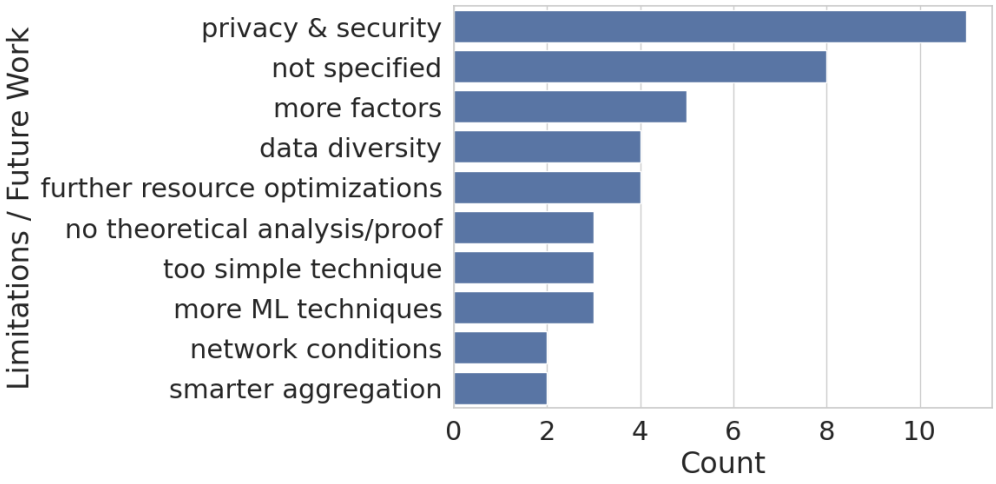

Figure 2.10: Limitations & Future Work of FL Papers

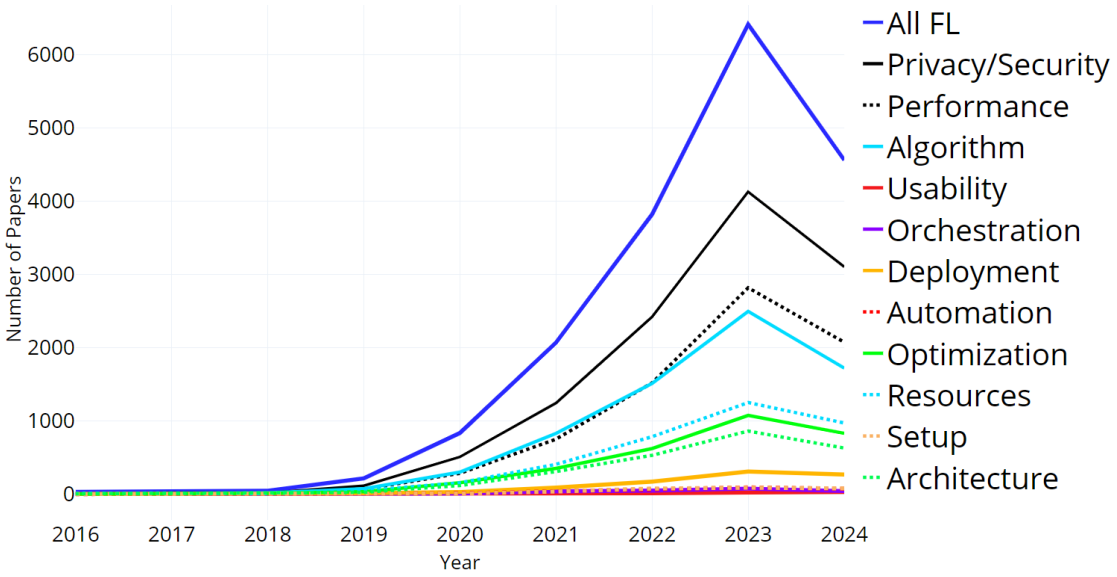

Figure 2.11: Evolution of FL Publications based on Keywords

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.

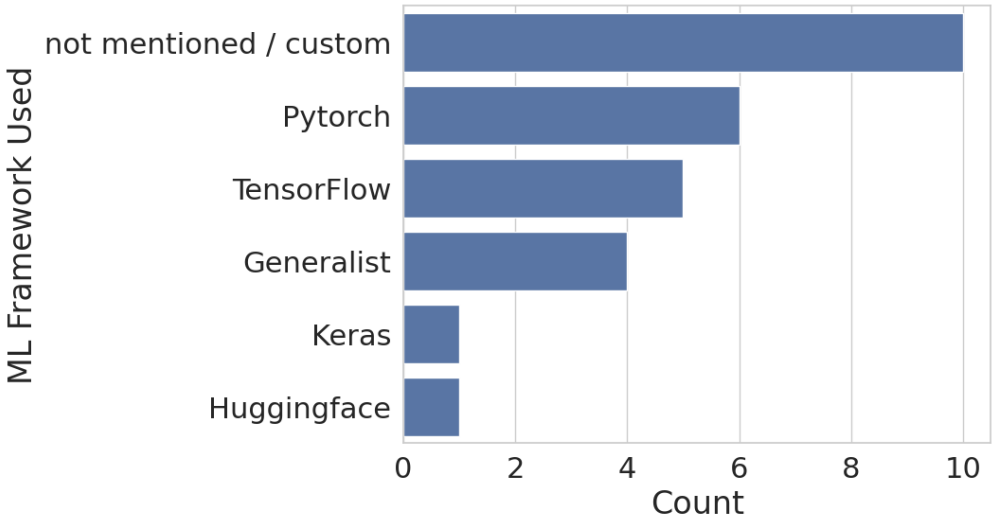

Figure 2.12: Distribution of mentioned ML Frameworks in FL Papers

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.

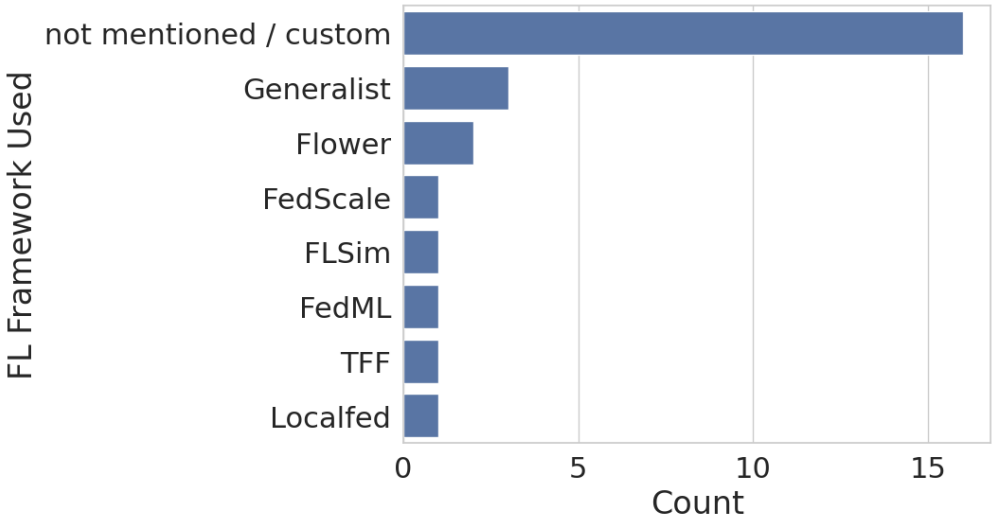

Figure 2.13: Distribution of mentioned FL Frameworks in FL Papers

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 [94] from 2023. He examined FL libraries, frameworks, and benchmarks. Many FL tools exist for specific niche use cases and architectures [94]. 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 [94].

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 [84]	0.9.0	two weeks ago	9.4k	2k	same day	2
Tensorflow Federated [98]	0.85.0	two days ago	2.3k	578	same day	168
FedML [31]	0.8.9	11 months ago	4.1k	776	3 months ago	118
Flower [39]	1.10.0	3 weeks ago	4.7k	815	same day	284
OpenFL [79]	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 [95] 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 [92] or [45]. [92] 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 [11] 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 [39]. 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 [9] 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 [36] is part of this ecosystem. This project is still in its infancy (v0.3.0). It allows users to pull HuggingFace [49] 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 [40], an extensive set of tutorials, guides, example projects [38], and documentation [41, 37] 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 [55]. 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 [55] 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 [72] 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 [71] 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 [68], code examples [70], 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 [43]. 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 [67]. 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 [78] or the Moby Project [73]. 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. [104] 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. [80] 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 [10].

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.

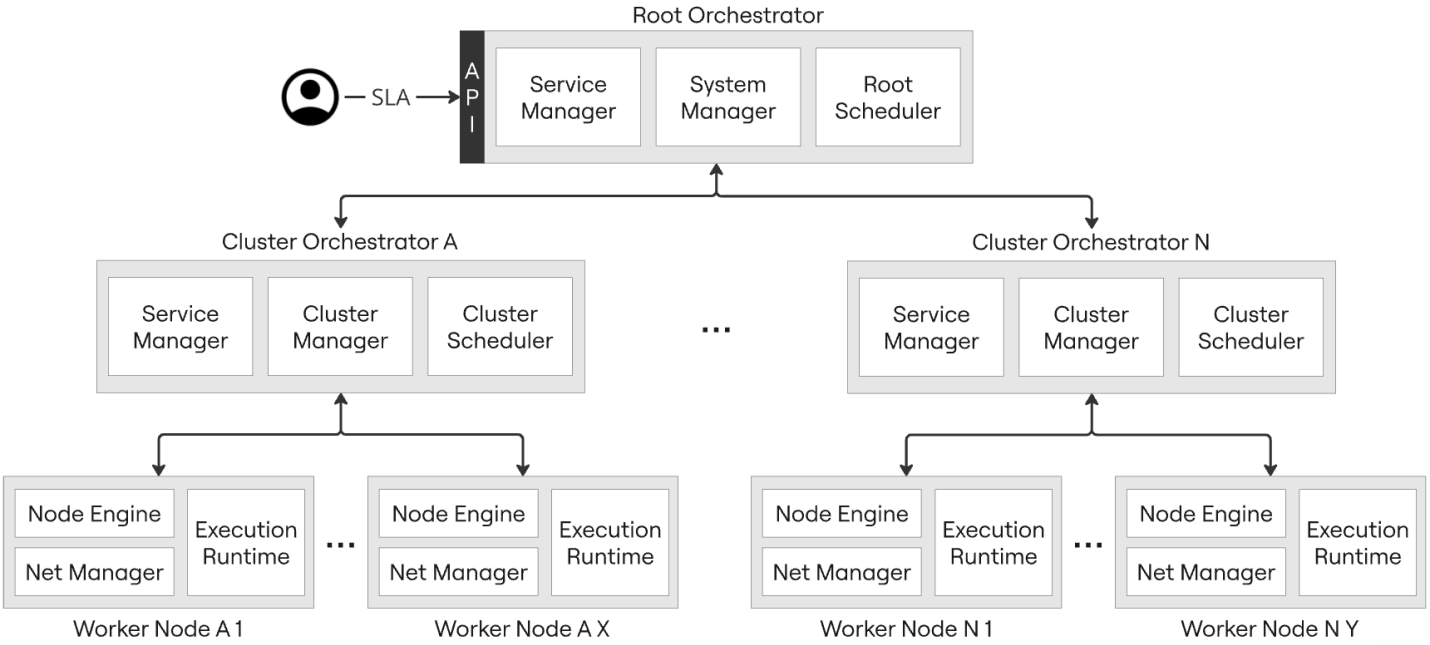

Figure 2.14: Simplified Oakestra Architecture

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 [18, 93] 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. [18] 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. [93] 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 [18] 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 CO2. 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. 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. This part also includes system models, which illustrate how the system works and fulfills the found requirements. 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. [14]

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 [33]. The system design part directly follows and relies on the requirements engineering part [14]. 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 [14]. Usually, these aspects are described via dynamic models, which directly correspond and depend on concepts discussed in the requirements analysis [14]. 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

Requirements engineering combines requirements elicitation and analysis. Requirements elicitation is an activity used to specify requirements. The problem statement (1.1), objectives (1.3), and weaknesses found in the FL field (2.1.4) are inputs to elicit requirements. These requirements are specified, analyzed, and concretized. Requirements elicitation and analysis are cyclically connected. After requirements are specified, they get analyzed, which can start another elicitation process. With each cycle, the understanding and specification of the requirements improve. This section derives functional and nonfunctional requirements based on the Requirements Analysis Document Template by Brügge et al. [14].

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. [14]

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 [14]. 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 abstractions such as analysis models representing how the system satisfies these requirements. The models do not depict the exact implementation. Instead, they show a simplified conceptual representation of FLOps' architecture and workflows. This includes involved components and their relationships. 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. 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. [14]

3.2.1 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. [14]

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

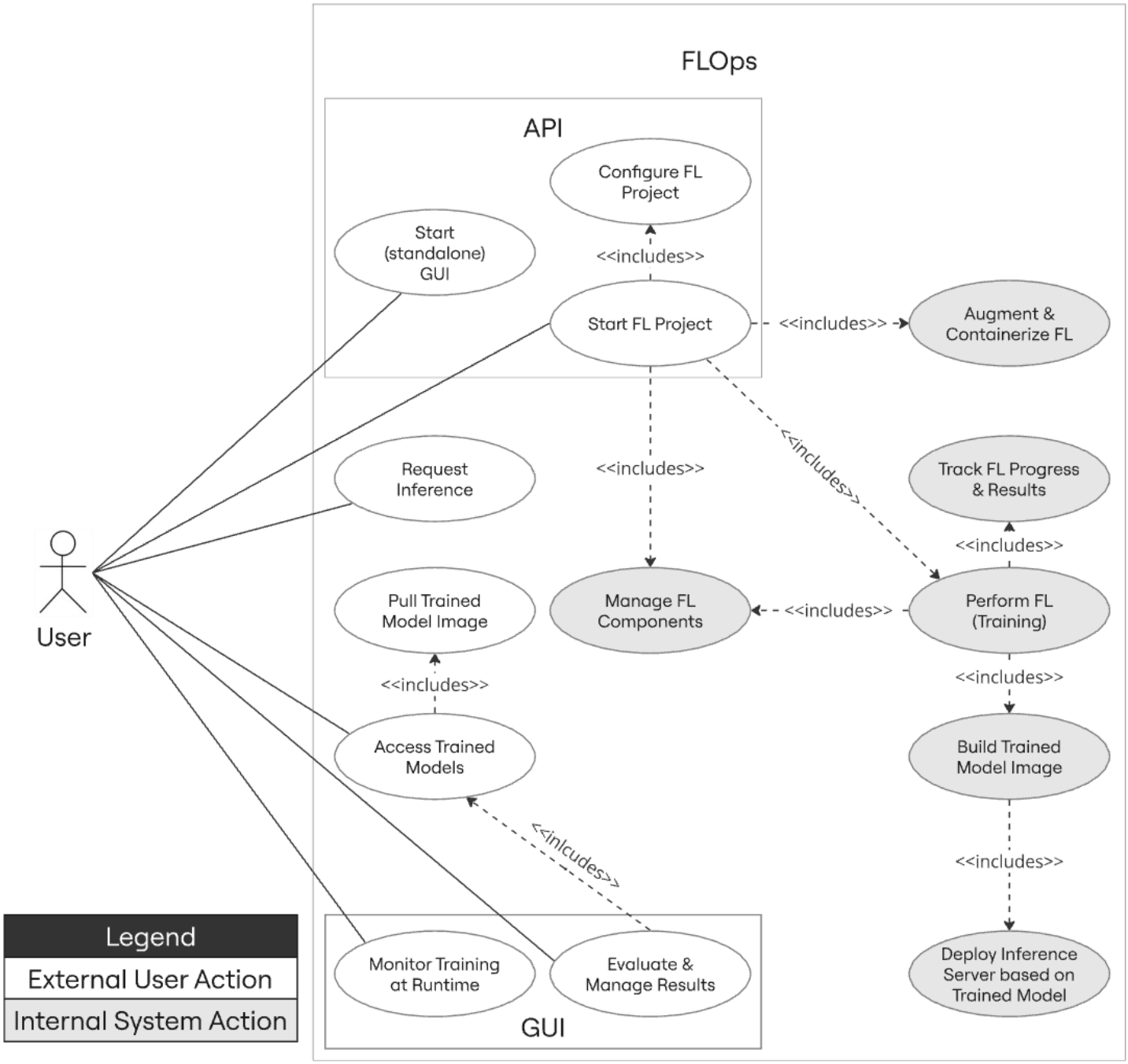

Figure 3.1: FLOps UML Use Case Diagram

(FR-4). After training, the model can be containerized and deployed as an inference server (FR-5, FR-6). The user can access this trained model (FR-5) and request services from its inference server (FR-6).

3.2.2 FLOps Overview

From this section onwards, we introduce the structures and concepts of FLOps that satisfy the gathered requirements 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.

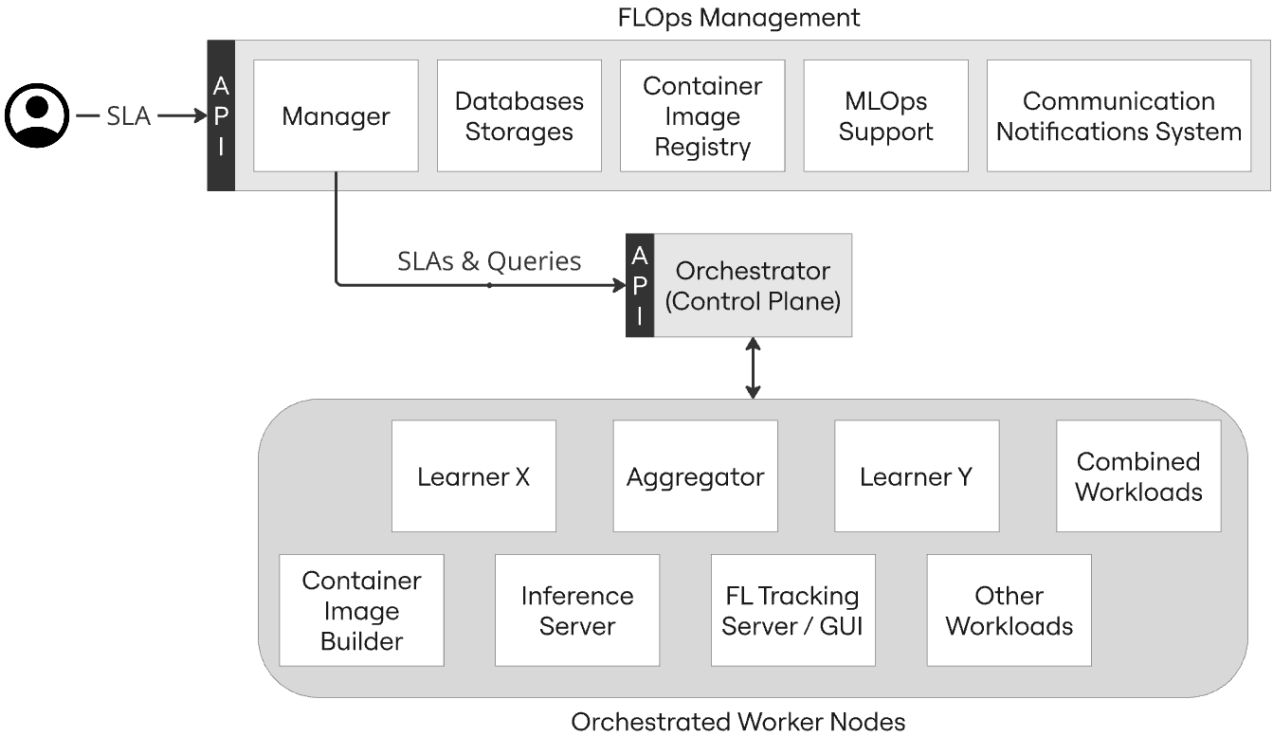

Figure 3.2: FLOps Structural Overview

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.

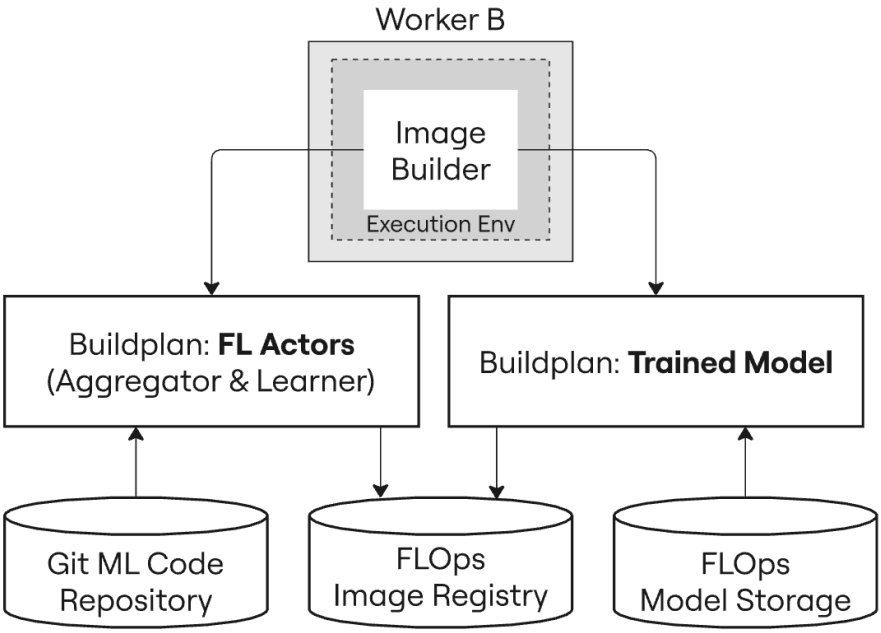

Figure 3.3: Simplified FLOps Image Builder Processes

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.

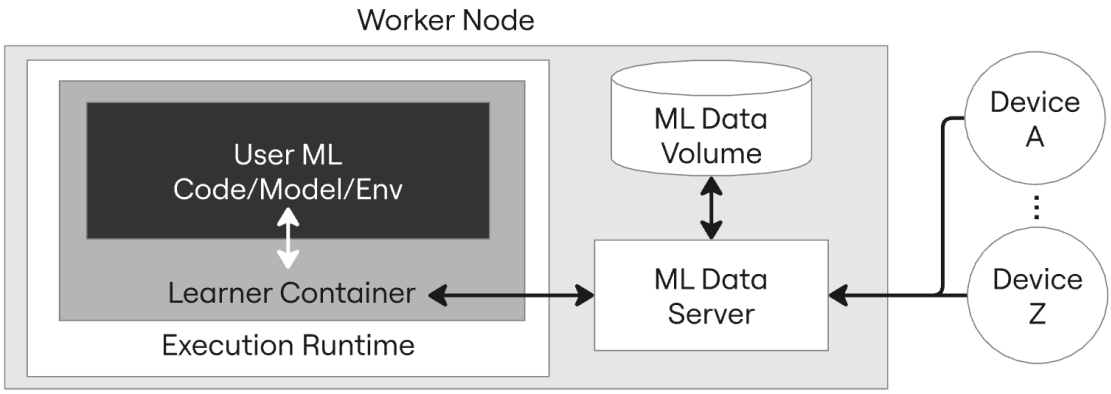

Figure 3.4: Simplified FLOps Local Data Management

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 [93].

3.2.3 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 [14].

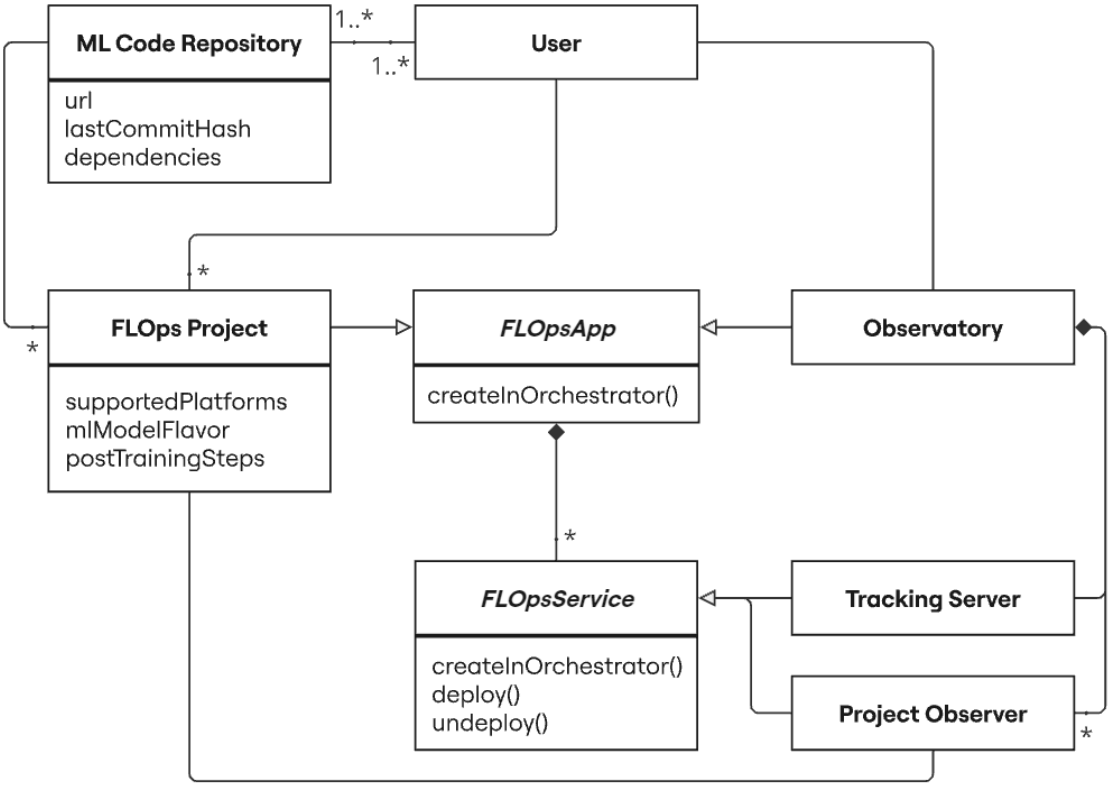

Figure 3.5: FLOps Core UML Analysis Object Model

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.

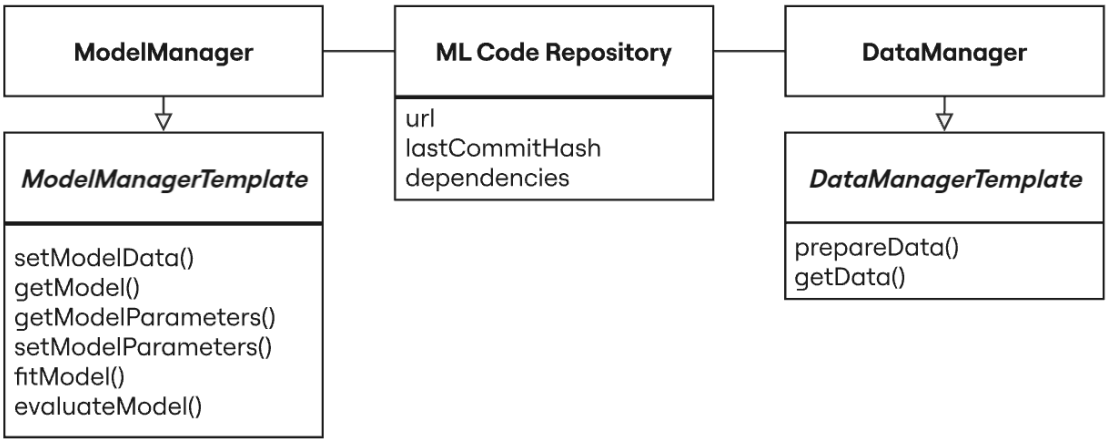

Figure 3.6: FLOps ML Code Repository UML Analysis Object Model

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.

Recommendation: 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 [34].

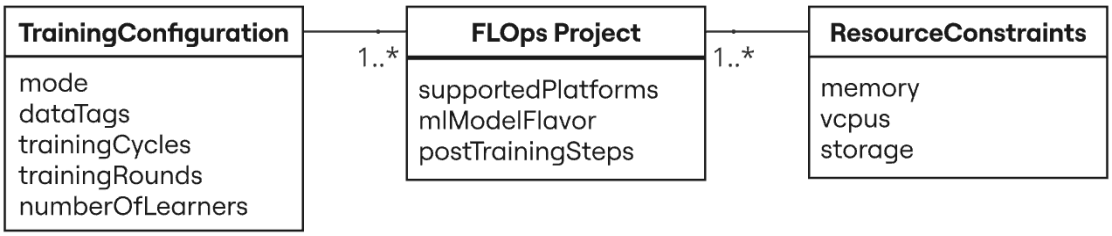

Figure 3.7: FLOps Project UML Analysis Object Model

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.

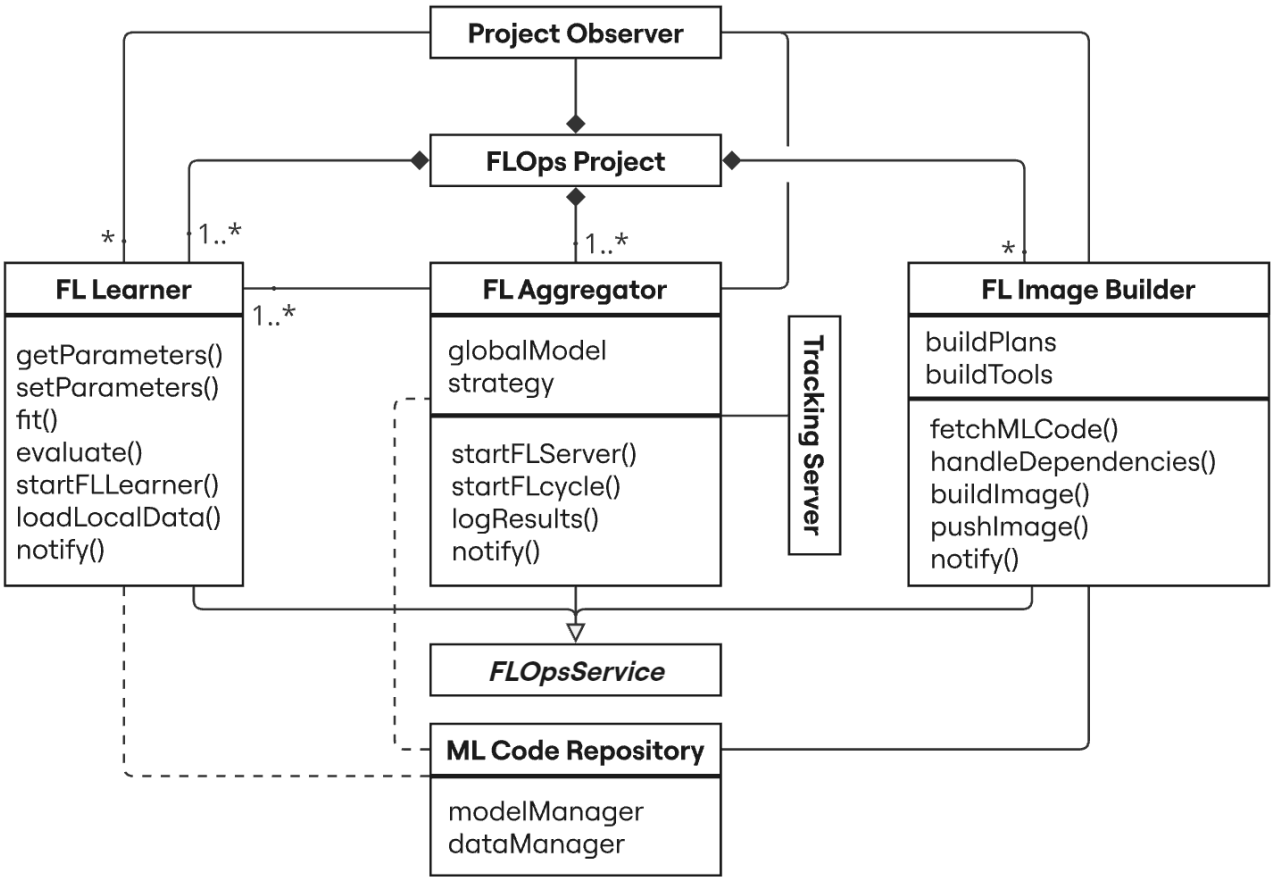

Figure 3.8: FLOps Project Services UML Analysis Object Model

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 build plans 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.

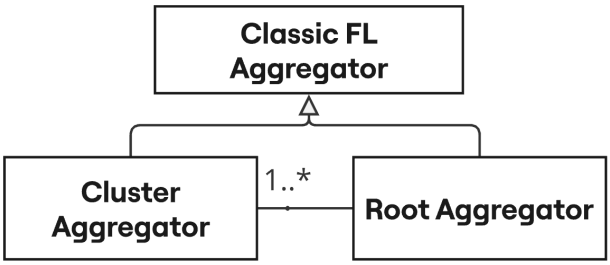

Figure 3.9: FLOps Aggregator Types UML Analysis Object Model

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.

3.2.4 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. 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. To improve readability and comprehension we split up the project into stages.

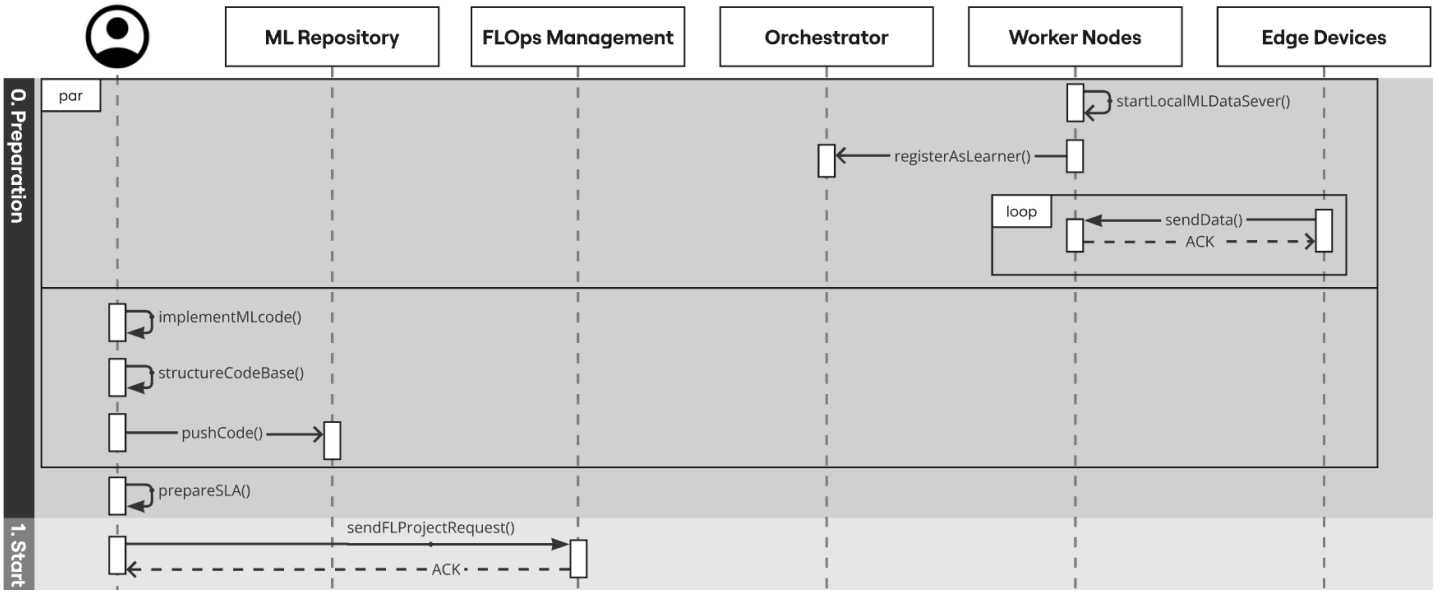

Figure 3.10: FLOps Preparation - UML Sequence Diagram

0. Preparation

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 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. When 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.

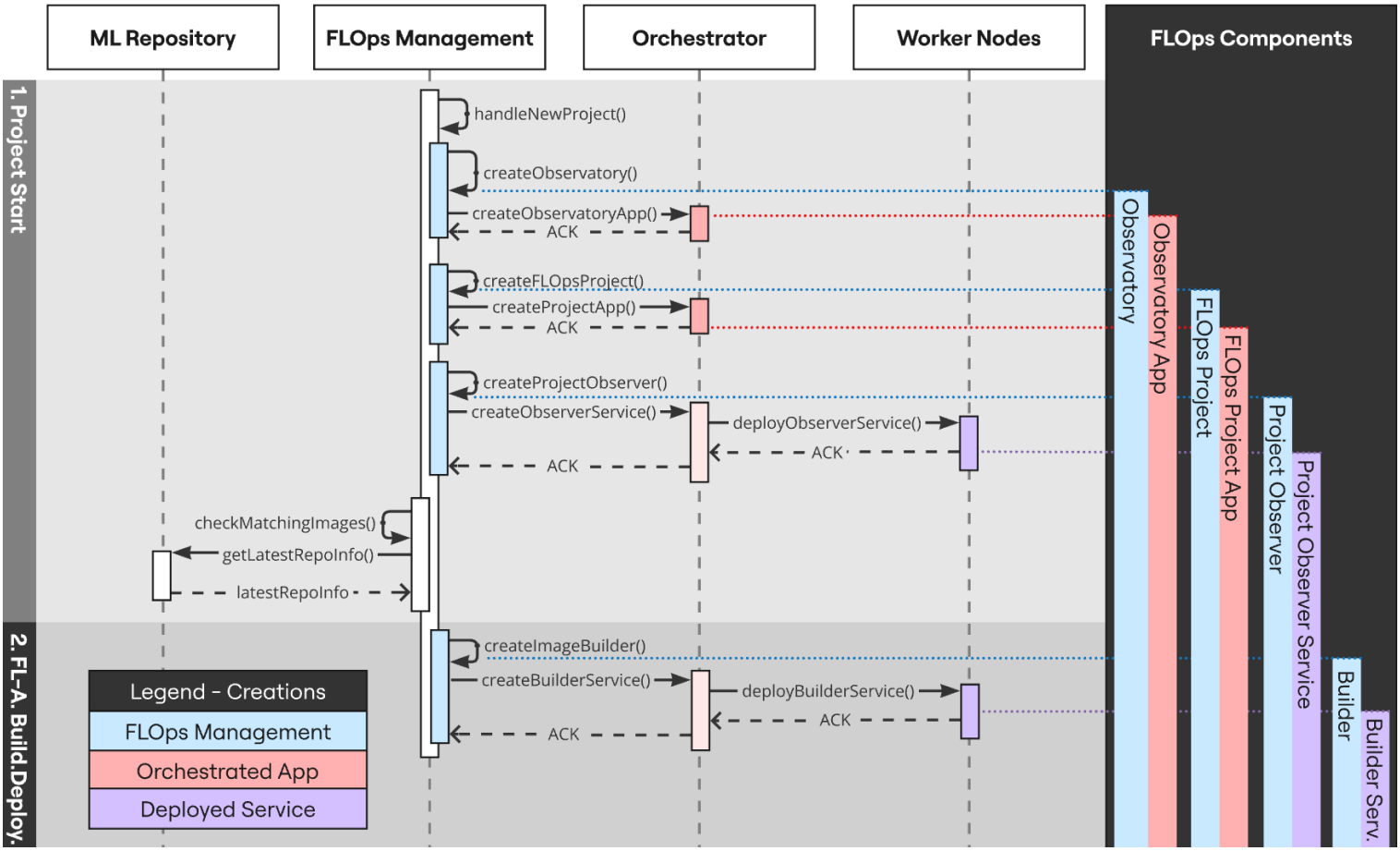

Figure 3.11: FLOps Project Start - UML Sequence Diagram

1. Project Start

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. When 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.

2. FL-Actors Image-Builder Deployment

The management creates a new image builder service and deploys it similarly to the project observer.

3. FL-Actors Image Build

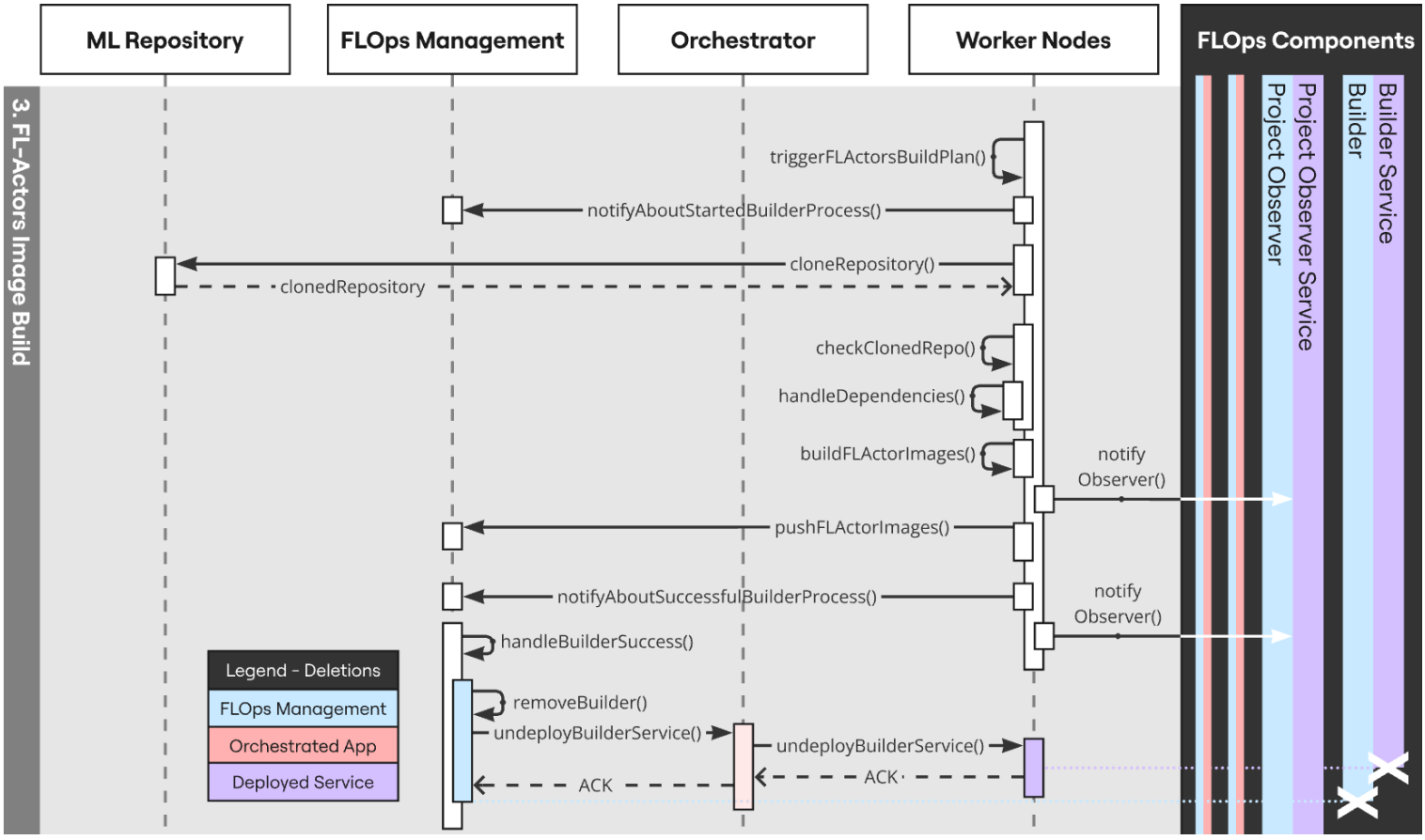

Figure 3.12: FLOps Image Builder Processes - UML Sequence Diagram

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. When 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.)

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.

4. FL-Actors Deployment (Aggregator Deployment Stage)

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

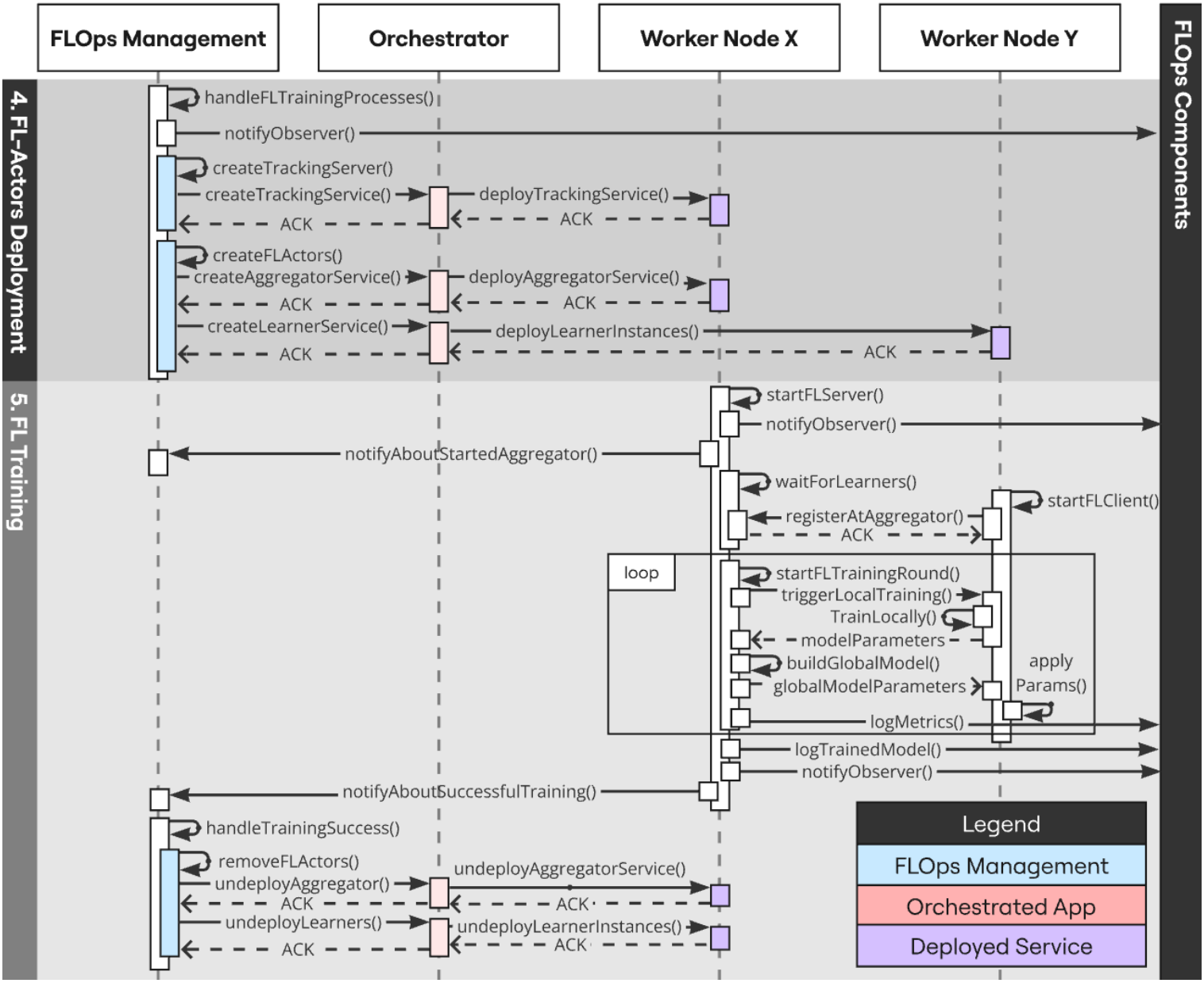

Figure 3.13: FLOps FL Training Processes - UML Sequence Diagram

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.

5. FL Training

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. When 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 a single time 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.

Further Stages

Similarly, FLOps realizes more complex configurations, modes, or post-training steps. For the post-training steps, the builder gets deployed again. 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.

3.2.5 Subsystem Decomposition

This subsection provides a more detailed look at FLOps' overall architecture and subsystems. This is a concretion of the system overview seen in Figure 3.2. Decomposing an extensive system into its sub-components provides new insights and improves comprehension of the system. The general approach for this endeavor is to use a UML component diagram. It analyses if the system follows the open-closed principle of good software design. This principle states that a system should "stay open for extension, but closed for modification" [14]. To achieve this the system should strive for minimizing coupling and maximizing cohesion. Cohesion expresses how tightly components of the same subsystem work together. Coupling describes how components from different

subsystems directly depend on each other without utilizing unifying interfaces, access points, or facades.

Figure 3.14 shows a UML component diagram of the major FLOps components and subsystems. The largest subsystems are the orchestrated layer and the FLOps management. The FLOps management consists of six components. The backend and artifact stores keep training metrics and models, respectively. They are MLflow components. The FLOps database stores all persistent information about FLOps' projects, components, apps, and services. Note that this database is cleared when the management suite is restarted unlike the MLOps storages that are persistent across management restarts. This simplifies state management and can be modified in the future. The FLOps manager coordinates all FLOps processes. These processes include serving a RESTful API for user requests, coordinating deployments with the orchestrator, checking requirements via its image registry, and accessing the user's repository. The manager is the most important and largest part of the FLOps management suite. The MQTT broker enables lightweight communication between the manager and deployed FLOps services. The FLOps image registry provides complete control and direct access to images built by the image builder services.

The orchestrated layer contains the control plane and the worker nodes. The control plane is independent of FLOps. Only the FLOps manager interacts with it via its REST API. The control plane resolves the FLOps management requests and creates, (un)deploys, or deletes components necessary for FLOps to run. The relevant apps and services for FLOps are deployed on single or multiple worker nodes. The two key FLOps apps are the project and observatory. All project services can share their status with the manager or project observer over MQTT or socket messages. The three services are dependent on the FLOps image registry. The image builder needs to push images to it, whereas the FL actors are pulled from it. By default, the learners and aggregator(s) communicate via gRPC (Flower). The observatory services are the project observer(s) and tracking server. The tracking server hosts the HTTP-based GUI and is accessible through a REST API. The aggregator sends its logged metrics and model over the tracking server to the management stores. The user can interact with the system via the GUI or by accessing the project observer's event (logs) over the control plane API or FLOps CLI.

The management image registry, project image builder, and observatory services are pulled from FLOps' public git image registry [33].

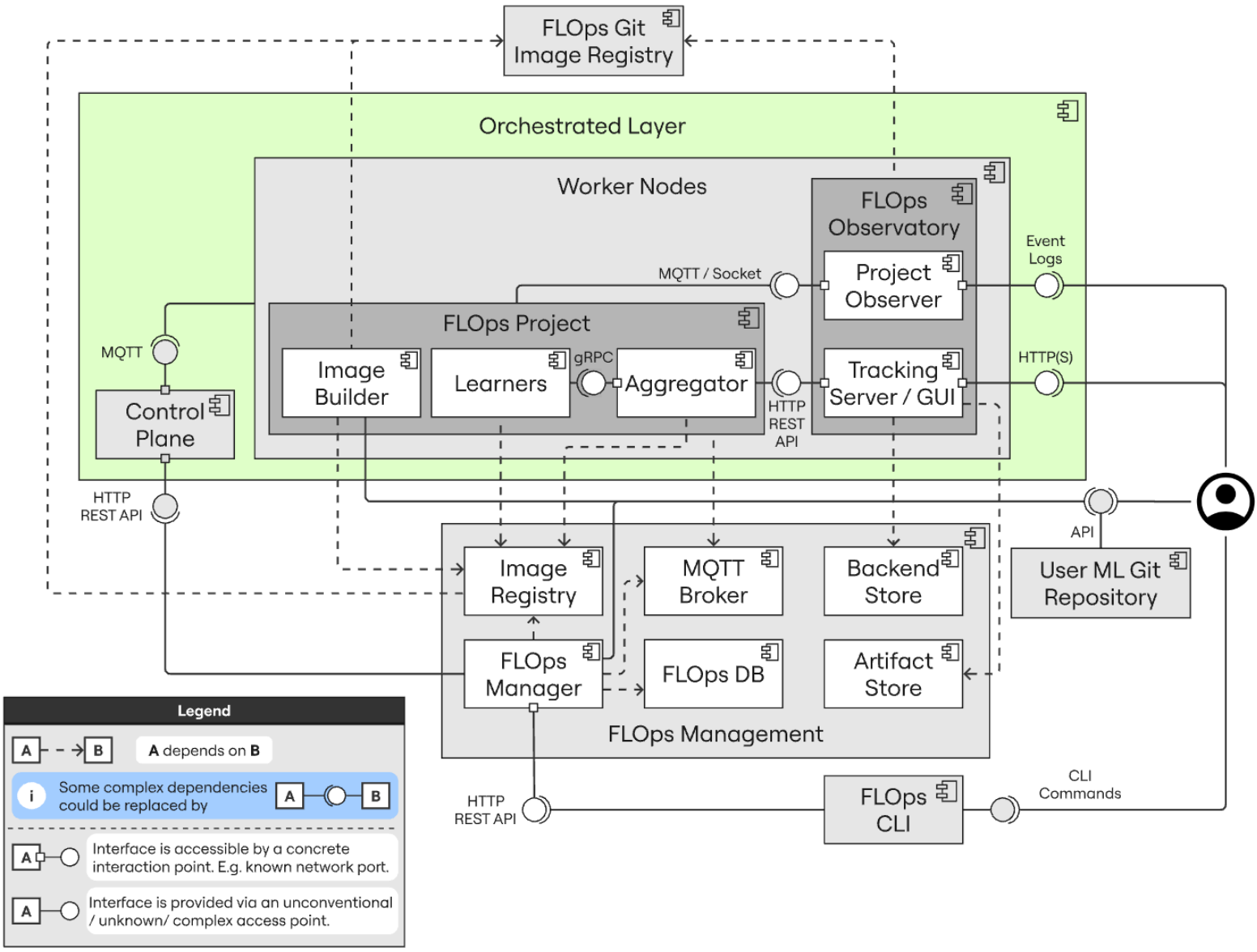

Figure 3.14: FLOps Subsystem Decomposition

4 Implementation Details

After discussing requirements and simplified system models, this chapter discloses significant details of the implementation that powers FLOps. We implemented FLOps mainly in Python. Our goal is to use state-of-the-art tools for FLOps. We analyzed and compared different open-source libraries and tools.

Verdict	GitHub	Name	Usage - ...	Category	Version	Open Is...	Contrib...	Stars	Forks	Used by	Last Commit	Last Release	Already in OAK?
Do Not Use	github.com/bp	flask-authorize	Authorization a	authorizat...	0.2.7 (no offici	9	5	64	12	-	2 months ago	Dec 2023	no
Several Issues - Careful if/w...	github.com/py	flask-restx	Fork of Flask-RI	REST API ...	1.3.0	252	121	2k	328	20.6k	2 months ago	Dec 2023	no
"Situational Must Have" - d...	github.com/Fla	flask-security	Quick and simp	Security ac	5.3.3	18	60	594	147	2.4k	7 hours ago	Dec 2023	no
"Situational Must Have" - d...	github.com/vir	flask-jwt-enxte	An open source	JWT	4.6.0	9	88	1.5k	240	-	2 week ago	Dec 2023	only in system manag
Do Not Use	github.com/fla	flask-restful	Simple framew	REST API ...	0.2.12	98	147	6.7k	1k	103k	9 months ago	2014 !	only in system-manag
Looks Good	github.com/mi	flask-httpauth	Simple extensi	authentic...	4.8.0	9	33	1.2k	225	18.2k	4 months ago	Aug 2023	no
Better alternatives Exist - n...	github.com/du	flask-praetoria	Strong, Simple,	Security JW	1.5.0 (no offical	4	19	339	64	-	3 months ago	Nov 2023	no
Do Not Use	github.com/lin	flask-user	Customizable U	authentic...	1.0.2.2 (only tag	99	36	1k	297	-	5 years ago	2019	no
Looks Good	github.com/ma	flask-login	Flask user sessi	authentic...	0.6.3	11	106	3.5k	806	158k	1 month ago	Oct 2023	no
Several Issues - Careful if/w...	github.com/sv	flask-swagger-	Swagger UI blu	Auto Doc...	4.11.1	17	6	175	58	-	2 years ago	2022	only in system-manag
"Situational Must Have" - d...	github.com/py	pydantic	Data validation	Data Valid...	2.6.1	326	496	17.6k	1.6k	327k	18 hours ago	2 weeks ago	no
Better alternatives Exist - n...	github.com/ma	marshmallow	A lightweight li	Data Valid...	3.20.2 (only tag	146	182	6.8k	640	119k	5 days ago	Jan 2024	several places
Looks Good Better alternativ	github.com/ma	flask-smorest	DB agnostic fra	REST API ...	0.24.1	54	20	611	69	2.6k	5 days ago	Aug 2020	several places
Looks Good	github.com/luc	flask-openapi3	Generate REST	REST API ...	3.0.2	11	16	140	29	672	3 weeks ago	3 weeks ago	no
Looks Good	github.com/py	pyopenssl	A Python wrapp	SSL Securit	24.0.0	63	99	851	426	-	3 weeks ago	1 month ago	no

Figure 4.1: Screenshot of internal python projects/libraries analysis

Figure 4.1 shows a screenshot of a subset of our internal comparison. We refrain from exploring and discussing our findings to avoid bloating the thesis. Especially because this landscape is constantly changing, and better options might exist since our analysis. We carefully considered every dependency that FLOps relies upon. The dependency listings are available in the codebase [33].

4.1 User Interactions with the FLOps Manager

This section details how users can interact with the FLOps manager. It shows the currently available API endpoints and the SLA structure.

4.1.1 API

We implemented the FLOps manager API via the Flask-OpenAPI3 framework [32]. It is a relatively young and small framework that aligns Flask with OpenAPI3. It uses Pydantic to verify data and automatically generate REST API and OpenAPI documentation that is compatible with popular frameworks such as Swagger. We use Waitress [101], a production-quality WSGI server, to host the manager's API.

Currently Available Endpoints

/api/flops/projects
This POST endpoint triggers a new FLOps project. It expects users to provide a JSON SLA with the required project configurations and a bearer token authorizing the user on the orchestrator. If no matching images exist, an image builder is created and deployed. If an adequate image already exists, the request concludes straight away. The user receives a confirmation that the new project has successfully started.

/api/flops/tracking
The tracking endpoint allows users to spawn their personal tracking servers at will independently from an active project. Usually, a tracking server is created during FL training. This GET endpoint returns the tracking server / GUI URL.

/api/flops/database
This DELETE endpoint only allows admins to reset the FLOps database. Otherwise, the entire FLOps management suite needs a restart. It returns a confirmation for the user.

/api/flops/mocks
This POST endpoint creates mock data providers and deploys them on fitting learner machines. We discuss these data providers later on in this thesis. Similar to the project, this endpoint returns a confirmation to the user.

4.1.2 SLAs

The FLOps manager can only instantiate a new project via an SLA. This service layer agreement currently has the following structure.

```
{
    % This key enables more verbose logging in the manager and project observer.
    "verbose": true, % default=false, optional
    % This ID should be the same as for the orchestrator.
    "customerID": "Admin",
    % FLOps has only been tested for GitHub so far.
    "ml_repo_url": "https://github.com/Malyuk-A/flops_ml_repo_mnist_sklearn",
    % Supported flavors include: sklearn, pytorch, tensorflow, keras.
    "ml_model_flavor": "sklearn",
    % This key only works for special repositories intended for development.
    % It tells the builder to use prebuilt base images to significantly speed up image builds and development.
    "use_devel_base_images": false, % default=false, optional
    % This key expects a list of target platforms on which the built images should run.
    % It supports linux/amd64 and linux/arm64.
    "supported_platforms": ["linux/amd64"], % default=["linux/amd64"], optional
    "training_configuration": {
        % This key specifies the FL type.
        % FLOps supports classic and hierarchical modes.
        "mode": "classic", % default="classic", optional
        % Requested data tags should match available data on learner nodes.

        % Multiple different tags can be requested.
        % The ML data server will use local data fragments that match any of the provided tags.
        % If no data tags are provided the learner will notify watchers that it cannot find any data.
        "data_tags": ["mnist"],
        % Training cycles only apply to the hierarchical mode.
        "training_cycles: 1,% default=1, optional
```

```
        % This key tells learners the number of training and evaluation
            rounds to perform.
        "training_rounds": 3,% default=3, optional
        % Clients mean learners in this context.
        "min_available_clients": 2,% default=1, optional
        "min_fit_clients": 2,% default=1, optional
        "min_evaluate_clients": 2% default=1, optional
    },
    % FLOps supports these two post-training steps.
    "post_training_steps": ["build_image_for_trained_model", "
        deploy_trained_model_image"], % default=[], optional
    % These are optional values that require fine-tuning.
    % Note that these values are not recommendations but placeholder
        values.
    "resource_constraints": {
        "memory": 250,% in MB
        "vcpus": 1,
        "storage": 25% in MB
    }
}
```

The following SLA shows a simple classic FL project configuration with both post-training steps enabled. It requires two learners for training.

```
{
    "verbose": true,
    "customerID": "Admin",
    "ml_repo_url": "https://github.com/Malyuk-A/
        flops_ml_repo_mnist_sklearn",
    "ml_model_flavor": "sklearn",
    "training_configuration": {
        "data_tags": ["mnist"],
        "min_available_clients": 2,
        "min_fit_clients": 2,
        "min_evaluate_clients": 2
    },
    "post_training_steps": ["build_image_for_trained_model", "
        deploy_trained_model_image"],
}
```

The next SLA displays a more advanced configuration with HFL, multi-platform support, and more FL actors.

```
{
    "customerID": "Admin",
    "ml_repo_url": "https://github.com/Malyuk-A/flops_ml_repo_cifar10
        _pytorch",
    "ml_model_flavor": "pytorch",
    "supported_platforms": ["linux/amd64", "linux/arm64"],
    "training_configuration": {
        "mode": "hierarchical",
        "data_tags": ["cifar10"],
        "training_cycles": 10,
        "training_rounds": 5,
        "min_available_clients":3,
        "min_fit_clients": 3,
        "min_evaluate_clients": 3
    },
    "post_training_steps": ["build_image_for_trained_model", "
        deploy_trained_model_image"],
}
```

4.2 Image Building

From our understanding the automatic image build process that turns user ML code into FL-capable containerized multi-platform images is one of the major novel contributions of this work. This process is a non-trivial endeavor requiring advanced understanding and application of various domains. This chapter is dedicated to analyzing these required synergies. Firstly, it elaborates on the need to build these images and explains related challenges. Secondly, it discusses different approaches and their limitations in building images in the given environment. Thirdly, this chapter showcases details about internal FLOps image builder processes. The last subsection explains how FLOps handles multi-platform image builds.

4.2.1 Dependency Management

The goal of using containerized images is to avoid the need and struggle to set up and configure machines individually. This setup and configuration part is especially crucial for ML workloads. Many different ML frameworks and libraries exist and need to

cooperate with other software tools. These tools cover various aspects of ML, ranging from unsupervised to supervised learning. Subfields include clustering, reducing feature spaces, classification, regression, and neural networks. Just neural networks as a field represent a multifaceted domain that requires different tools. Neural networks span from simple nets with a single hidden layer to deep neural networks, convolutional networks, transformers, and many more. Besides targeting different ML disciplines, they can be fine-tuned for specific environments, such as massive supercomputers, end-user work machines, or IoT and edge devices. Popular tools include TensorFlow, Keras, PyTorch, and Scikit-learn, which all have different extensions and forked projects. Even small basic ML projects can already contain several hundred dependencies. All of these tools have different versions and require careful consideration and version management to avoid unexpected side effects and errors.

The relevant literature almost never touches upon the initial device configuration and setup (2.1.4). Most authors assume that these dependencies are already properly installed and configured on the devices. Many works in FL discuss and propose better ways of selecting and distributing training loads and model parameters (2.1.4). They also omit the aspects of dependencies. Before it is possible to train a model via FL or ML, the device needs to have all the necessary capabilities correctly configured. Containers resolve most of these issues, especially for heterogeneous devices and environments. These reasons motivated us to develop and include automatic image-build processes.

Conventionally, ML workflows are implemented in Python. Therefore, the mentioned dependencies are handled via Python's own package repository, PyPI (Python Package Index). These packages are usually installed and managed via Python's package installer pip. These standard tools display weaknesses when resolving dependencies in extensive projects. It can happen that pip fails to resolve and properly install the dependencies in a compatible way. Additionally, because pip is implemented in Python, which is not the fastest programming language, this dependency resolution can take a long time. Other tools try to resolve these weak points.

A popular alternative to pip is the Conda suite. Conda [23] is an open-source package and environment manager that is popular with ML and Data engineers. It focuses on Python but supports other languages as well. Conda provides and enables convenient virtual Python environments and resolves dependencies more successfully and quickly than pip. Anaconda [4] is Conda's Python distribution that includes Conda and comes with more than 100 commonly used packages. This distribution is heavy-weight and unfit for lightweight, containerized, or CI workflows. Miniconda [66] is a minimal Anaconda version that only includes Conda and a small mandatory set of dependencies. It is ideal for lightweight environments that should only include necessary (custom) dependencies. Conda is also implemented in Python. Thus, the dependency resolution process speed remains a bottleneck. Mamba [63] is a reimplementation of Conda in

C++. This allows Mamba to resolve dependencies a lot faster than Conda. Similarly to Miniconda, Micromamba [65] is a minimal distribution of Mamba.

Additionally, there is no single unified source for packages or standard for building such packages. Various package servers, mirrors, channels, and package builders exist. One example is Conda-Forge. Furthermore, the structure, build, and publish processes for Python packages have changed vastly over the years. Native Python only recently switched to a more homogeneous approach via pyproject.toml files [97]. Previously setup.py and setup.cfg files were used for these purposes. In addition, there are other tools like Poetry [83] or the very new and lightning-fast uv [100] manager that is implemented in Rust. In conclusion, Python's dependency management ecosystem is vast and complex. The newer a tool, the quicker it tends to be, but it also lacks sophisticated maintenance and might not support all necessary packages or versions.

FLOps should support as many projects and dependencies as possible. For this reason, FLOps uses miniconda. It supports different Python versions, easily resolves and handles dependencies, and is a lightweight solution. Conda now also supports multiple different dependency resolvers. Since the end of 2023, Conda has been using the libmamba resolver by default, which uses Mamba [24]. Therefore, FLOps uses a fast and reliable dependency resolver.

Besides the complexity of Python's dependency landscape, ML dependencies can be huge.

Image	python:3.12.4	python:3.12.4-slim	anaconda3:latest	miniconda3:latest
Size	1.02 GB	133 MB	4.5 GB	611 MB

Table 4.1: Conda Python Image Size Comparison (29.08.2024)

Table 4.1 shows four pulled images and their sizes. All images use Python version 3.12.4. The default Python image is one GB in size. Its official slim alternative is almost ten times smaller. The full anaconda image has 4.5 GB. The miniconda image is more than seven times smaller than the full anaconda version but more than five times larger than the slim Python image. Note that these are the pulled docker image sizes, not the compressed registry ones. (The Conda images are from the official continuumio project.)

Image	smizy/scikit-learn-docker	tensorflow/tensorflow	pytorch/pytorch
Size	515 MB	1.86 GB	7.6 GB

Table 4.2: A selection of popular ML library images and their sizes (29.08.2024)

Table 4.2 shows a selection of pulled images of popular ML tools. All images are the latest official images by the ML tool providers, except scikit-learn. These images can

vary enormously in size. Some of them require a lot of disk space and bandwidth to pull. Any FLOps image built on top of these dependencies will be even larger due to the additional FL and MLOps dependencies.

4.2.2 Image Builders

The most popular and widespread containerization tool is Docker. It is also preferred when building images. Usually, building images via Docker is straightforward. One creates a Dockerfile and runs Docker commands to build a new image on a host machine based on this file.

FLOps requires a more complex approach to build images. Its image builder is a service running on a worker node that Oakestra orchestrates. This service is a containerd container. It is not possible to simply run docker in such an environment to build images. The critical issue here is that docker cannot be run trivially in another container due to restricted privileges and access to host system capabilities. A popular workaround, especially for CI/CD workloads, is called Docker in Docker (DinD). Many CI/CD tools support DinD natively. One example includes GitLab's CI/CD. Setting up and using DinD manually on a host machine is non-trivial and error-prone. Getting docker to work inside managed Oakestra containers is even more challenging. One significant reason for these issues is that docker requires its daemon to run for building images. Nested Docker daemons are a complicated matter that can require significant adjustments to how the orchestrator coordinates containerization on worker nodes. For example, the orchestrator could require worker nodes to mount their docker sockets to the container, which would lead to further security risks.

The only functionality that FLOps requires here is building images inside containers, not executing them from within. There exist various alternatives to Docker that specialize in building containerized images. Thanks to the coordinated efforts of the OCI [78] and others, these tools all build compatible images. Alternative tools include kaniko, Podman [82], and Buildah [16]. FLOps uses Buildah.

Buildah is an open-source daemon-free tool that specializes in building OCI-compliant images. It can run on host machines or inside containers. It features many equivalent Docker commands and supports building images step-wise, programmatically, or via Dockerfiles. Red Hat released Buildah initially in 2017, and its first major version was released in 2018. Podman and Buildah are complementary tools that have some shared documentation and aspects. Great resources to find out more about Podman, Buildah, and their relationship are available here [90, 89, 15].

It was challenging to make container-internal image-building work for FLOps via Oakestra. This image build process is especially complicated because it performs heavy dependency resolutions and installations. We needed to add a new optional

flag to Oakestra deployments that enabled to mount /dev/fuse. The FUSE userspace filesystem framework enables non-privileged and secure mounts [60]. In addition, Buildah has to build its images with the chroot isolation flag enabled. As a result, FLOps uses a highly fine-tuned and optimized builder environment that enables it to build images inside containers of orchestrated worker nodes.

Furthermore, we put a lot of thought and time into refining and polishing FLOps' docker files. They build the foundation for all FLOps components, from project services and management components to auxiliary containers.

4.2.3 FLOps Image Builder Details

This subsection showcases how the FLOps image builder service concretely builds its different images. The following expands upon Figure 3.3 from 3.2.2.

Figure 4.2 depicts the details how FLOps Image Builder works. The grey Ms represent the untrained model (structure). Purple Ms stand for the trained model. Hexagons symbolize container images. The service tracks the time different steps take and returns to the FLOps manager a summary of its total runtime and the runtime of individual steps. The image builder supports two different build plans.

The first step in the FL Actors build plan is to fetch the user's ML code from his specified repository. Secondly, the service builds a base image that contains all dependencies common to the learner and aggregator. Due to the current multi-platform solution, the service pushes the base image to the image registry hosted in FLOps management. Building and pushing the base image take up most of the service's total runtime. The service continues to build the FL actor images one after another, pushing them in the end. Thanks to the base image, these steps are relatively quick. Pushing the base image does not generate meaningful overhead because of image layer caching. The FL actor images reuse all the base image layers. Thus, pushing them is accelerated, and the image registry recognizes and reuses its base image copy's layers.

Flower's design does not require the aggregator to possess any information about the model, including its structure or dependencies. The aggregator's job is to average the received model parameters. This process is based on simple mathematics and requires no other model-specific information or dependencies. Therefore, the aggregator image and node can be relatively lightweight compared to the learners. However, logging the trained model via MLflow requires access to the complete trained model, especially its structure. The corresponding dependencies are necessary because a model structure is defined in a concrete ML framework. Therefore, FLOps explicitly also includes these dependencies and the model structure in the aggregator. During FL training only the model parameters are transmitted. The aggregator's model copy is only needed at the very end of training. The aggregator populates its untrained model copy via its

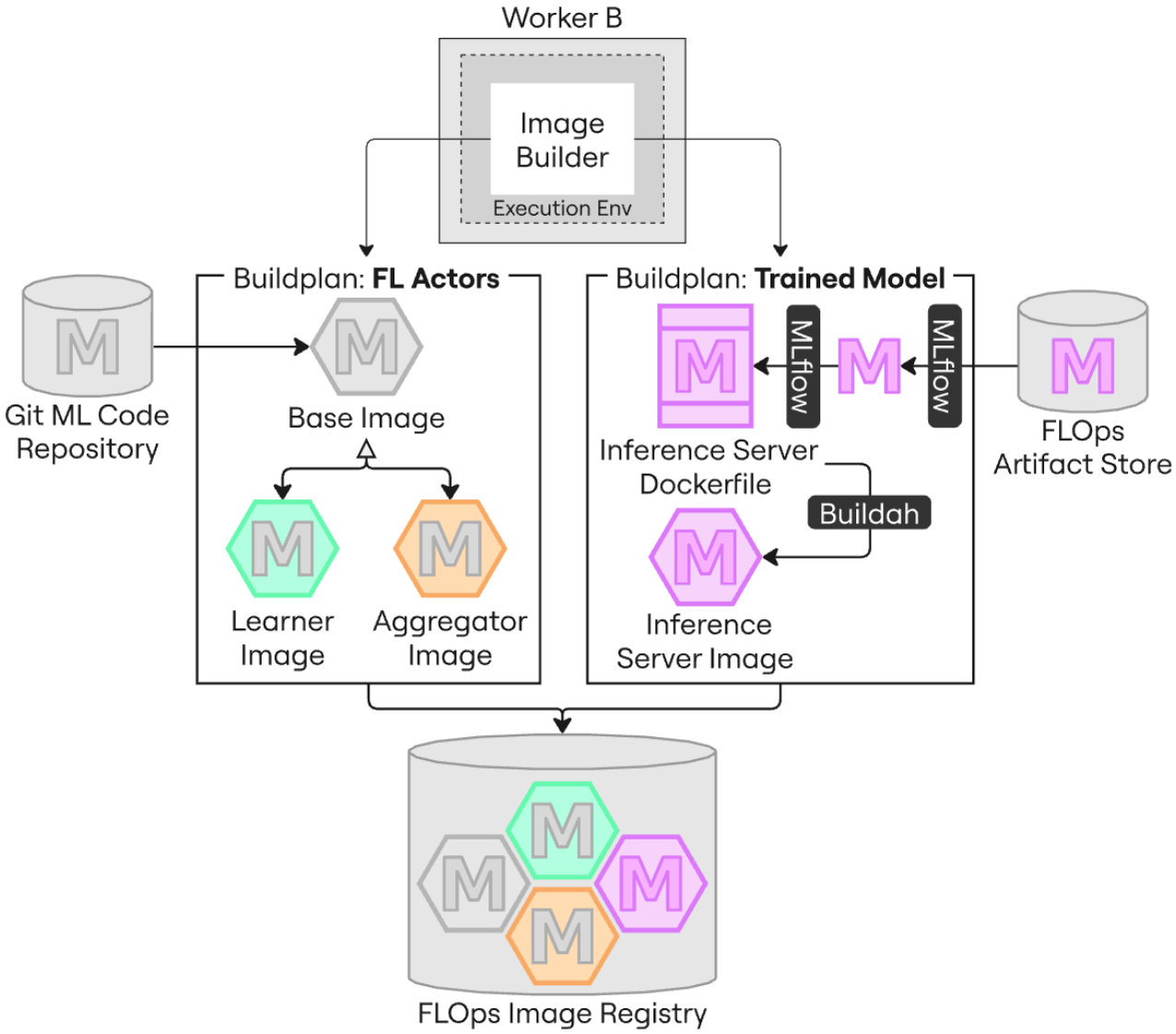

Figure 4.2: Detailed FLOps Image Builder Processes

final global model parameters. This populated model gets logged. Note that the model structure is initially defined in the user's ML repository and can be easily cloned and injected into images via the builder service.

The trained model build plan can only be run after the FL training is completed and the trained model is saved in the artifact store, which is hosted in the FLOps management. MLflow provides commands to turn stored models into containers [69]. The issue with this approach is that MLflow only provides this capability via Docker for all these commands, including building. This approach works well directly on host machines but not inside containers (4.2.2). As a workaround, FLOps uses MLflow to pull the stored trained model into the builder container. Then, the service uses MLflow again to create a dockerfile based on this model. FLOps' builder service augments this dockerfile to support multiple platforms and builds the trained model image via Buildah. This built image wraps the trained model via an inference server. One optional FLOps post-training step deploys this inference server directly after the builder service terminates.

Image	Image Builder	MLflow	Client	Server	SuperNode	SuperLink
Size	3.0 GB	819 MB	1.16 GB	1.13 GB	232 MB	232 MB

Table 4.3: Important FLOps Image Sizes (30.08.2024)

Table 4.3 shows the sizes of different relevant images to provide more context for the final results and build processes. The Image Builder image is the FLOps builder service. It is noteworthy that FLOps' builder service without the MLflow (2.12.1) dependency is 900 MB and the official MLflow image is 819 MB. The increase of more than 2.4 GB only due to this dependency is worth investigating. The remaining images from the table are all from Flower [35]. Flower recently introduced a significant change (Flower Next API [42]) in how they use FL clients and servers. They plan to deprecate the old approach that FLOps is using. We tried migrating to the new Flower paradigm but encountered several issues with our current implementation. Explaining Flower Next and FLOps' challenges with it would bloat this thesis. This aspect is a great topic for future FLOps improvements. The client and server images from the table are deprecated. Their new replacements are the SuperNode and SuperLink, which are significantly smaller. We mention these images to compare them with FLOps' FL actor images.

Standalone	**Base**	**Learner**	**Aggregator**	**Total Process**	**Base Image Build**
515 MB	2.79 GB	2.79 GB	3.55 GB	6min 20s	3min 53s

Table 4.4: Simple Scikit-learn MNIST Build Example

Table 4.4 shows a singular example of running a FLOps project using Scikit-learn and the MNIST dataset. The Standalone refers to the standalone Scikit-learn image from Table 4.2. The base and learner images are equally large because the learner image only changes the files it works with but reuses all dependencies from the base image. Therefore, the aggregator has a superset of the learner dependencies. The total execution time for the builder service took 6 minutes and 20 seconds. The base image build alone took almost four minutes.

FLOps Management uses and hosts an instance of the open-source CNCF (Cloud Native Computing Foundation) Distribution Registry [22]. This registry allows FLOps to be independent of any other registry provider. FLOps has complete control and immediate access to this registry.

4.2.4 Multi-Platform

This subsection explains how FLOps builds multi-platform images. Usually, end users build images only for a single platform. Their builder knows their host's architecture and uses it as a target platform. When inspecting popular public images, they support multiple target platforms. Each image tag can have multiple digests, each representing a different platform. For example, the latest Alpine image [3] supports at least linux/amd64 and linux/arm/v6. By default, it is impossible to run images that are intended for different architectures on machines with incompatible host architectures. Specific workarounds via emulation exist.

The key of building and referencing images that support multiple platforms are manifests. A plain manifest file contains information about a unique image digest. This information includes its media type, size, layers, and architecture. When an image supports multiple platforms, it has multiple digests, thus one manifest per digest. Image indexes were invented to group these different manifests. Note that image indexes refer to the OCI standard term [77]. In Docker, they are called fat manifests or manifest lists [28]. As a result, different host architectures can use the same image tag and pull their matching digest image. This happens because the local builder reads the image index and picks a suitable manifest. Examples for manifests are available here [28].

Multi-platform images are a deep topic. Because of its rapid development, many different media types, versions, and schemas for manifests exist. Build machines need to use the same conventions as their image registries. Discussing these details here would lead to bloat. Excellent information about this topic is available here [50].

Previously, one had to manually build one image per target architecture on a machine of that architecture, provide the image tag with an architecture suffix, and push it. Once all these different images were pushed, an image index had to be created and

pushed. Nowadays, a convenient solution for building multi-platform images is using docker's buildx builder [27]. It can build and push multi-platform images concurrently with a single command.

The FLOps image builder does not use docker to build its images (4.2.2). Buildah also supports building multi-platform images but lacks the convenient new features of docker's buildx. From our experience, Buildah lacks sufficient documentation regarding building and pushing multi-platform images. We needed to look into its source code, read other sources, and experiment extensively until we made this work.

Another significant requirement for building multi-platform images on a single machine is emulating other architectures. Otherwise, only images for the host's architecture can be built. It is also possible to cross-compile or use multiple dedicated builder nodes with proper architectures [29]. Conventionally, QEMU [87] is used to emulate such tasks. Docker Desktop includes QEMU for Mac and Windows by default [29]. QEMU translates the requested target architecture instructions into ones the host machine can understand. Due to FLOps' special circumstances, which involve building complex images in orchestrated, containerd containers on heterogeneous devices, various approaches seem to be available to realize emulation. Ideally, the emulator would be part of the builder image, thus avoiding the need to modify or require anything from the worker nodes. After many unsuccessful attempts, we decided to require worker nodes that should build multi-platform images to pre-install QEMU. For Linux machines, this can be done by installing an open-source package called qemu-user-static.

In conclusion, using this approach, the FLOps builder service can build multi-platform images. All other FLOps (static) images, including the builder service or project observer, are also available for linux/amd64 and linux/arm64. For example, all these images can be started on a Raspberry Pi 4, but these devices seem to lack sufficient resources to handle the FL training. An additional downside of multi-platform image builds comes with the slowness of emulation.

Platform	Full Build	Base Image	Actor Images
linux/amd64	4min	2min 30s	1min 30s
linux/arm64	18min	12min	6min

Table 4.5: Simple Scikit-learn Multi-Platform Build Times Example

Table 4.5 shows a simple example of FLOps' builder service's build times for different platforms. The build machine natively supports linux/amd64. Therefore, this build is much faster than the emulated arm one. Both rows show the build times when only a single platform is requested. The build times would be combined to support both platforms simultaneously.

4.3 Local Data Management

This section explains how FLOps handles local data on learner nodes. Firstly, it covers what kind of data and datasets suit FL and why. Secondly, it discusses how state-of-the-art projects in the industry handle enormous amounts of data for ML and Big Data applications. Thirdly, it explores how exactly FLOps manages the local data and what architecture it uses for this task. The last subsection showcases FLOps' mock data provider service, which makes development and testing more convenient.

4.3.1 Appropriate Data for FL

FL, especially cross-device FL, specializes in massive numbers of heterogeneous devices with diverse non-IID data. For FL, one can use conventional datasets, such as MNIST or CIFAR10. However, such homogeneous and IID data is not representative of data found on real devices. Many works in the field of FL specialize and compare how well they perform on non-IID data (2.1.4). For this reason, various datasets and benchmarks have been created explicitly for FL. One recurring prominent FL benchmarking tool is LEAF [17]. It does not solely specialize in FL but in more general federated settings. It includes several implementations and datasets. As mentioned in Table 2.2, we had little success working with this benchmark. This finding is noteworthy because LEAF seems to be the primary and sole source for the FEMNIST dataset, which many FL papers mention and use for evaluation.

FEMNIST seems to be one of the most popular dedicated FL benchmarking datasets. It is a federated version of the Extended MNIST (EMNIST [30]) dataset. FEMNIST splits the EMNIST dataset into multiple classes or data partitions based on individual (digit/symbol) writers. Extended projects exist that wrap the access to FEMNIST via the high-speed HDF5 binary data format [46]. The FEMNIST dataset is a prominent example of many dedicated FL datasets and benchmarks. A detailed listing and comparison of other similar resources is available in [94].

FLOps requires a convenient way of accessing suitable data for development and testing purposes. Our experience trying out LEAF taught us that using these dedicated datasets can be challenging and error-prone. Instead of figuring out how to unify these different dedicated FL datasets and benchmarks, we use Flower Datasets [36]. We already discussed Flower Datasets briefly in 2.1.6. The vital thing to know about this young project is that it uses and splits up Hugging Face datasets into non-IID data fragments. It enables users to turn conventional ML datasets into FL-optimized ones. Users can configure this approach freely.

4.3.2 ML & Big Data Formats

The field of Big Data and ML formats is vast and complex. It provides many insights into different optimization approaches. Great resources to find out more are available here [56, 58, 26, 6]. The following are significant takeaways after investigating this domain.

Managing Big Data is a booming Field
Storing and handling Big Data is a massively popular, expensive, and profitable business that regularly attracts hundreds of million-dollar investments. This environment leads to healthy competition, solid standards, and bold advancements in the field.

Reuse
Data management and optimizations are universally needed. These areas have several decades of solid research to back up best practices and avoid known pitfalls. Similarly to security, one should avoid reinventing and reimplementing foundational features from the ground up. Instead, it is recommended that existing open-source industry-favored solutions be reused.

(De)Serialization is a critical Bottleneck
When each subsystem has its own internal memory format, significant amounts of CPU work get wasted on (de)serialization. The recommended way to avoid this is to stick to a uniform format. The more tools support such a uniform format, the easier and faster cooperation, communication, and transmission can be.

Big Data and ML Data should use Columnar Formats
Usually, dataset features are split up into different columns. These features can be diverse and require different data types for storage. When storing and handling conventionally stored row-wise data, all these different data types and features complicate and slow down processing. Instead, if the data is stored and handled column-wise, advanced optimizations, and compressions can handle homogeneous features and their data type. As a result, the same data can be processed more compactly and faster.

The sources above recommend the following open source state-of-the-art formats and technologies. All of them are from Apache.

Arrow
Arrow is a language-agnostic columnar memory format. It is optimized for modern CPUs and GPUs. It supports zero-copy reads, which avoid serialization and accelerate

data access. Arrow is especially popular for interoperability. This format should be used for data in memory. [7]

Parquet
Parquet is also a columnar file format. Its focus is on efficient data storage and retrieval. Its benefit over Arrow is that it needs less space due to its special compression and encoding. This format should be used to store data on disks. [8]

Arrow Flight
Arrow Flight is a gRPC-based framework. It supports parallel data streams. When used with compatible data, it overcomes (de)serialization overheads and speeds up data transfer. Flight should be used to transport Apache formatted data over the network. [5]

The last subsection mentioned that FLOps uses Flower Datasets, which uses Hugging Face Datasets underneath. Hugging Face Datasets use Arrow [48]. Therefore, the findings in this subsection are directly applicable and relevant to FLOps.

4.3.3 FLOps' Local Data Management Architecture

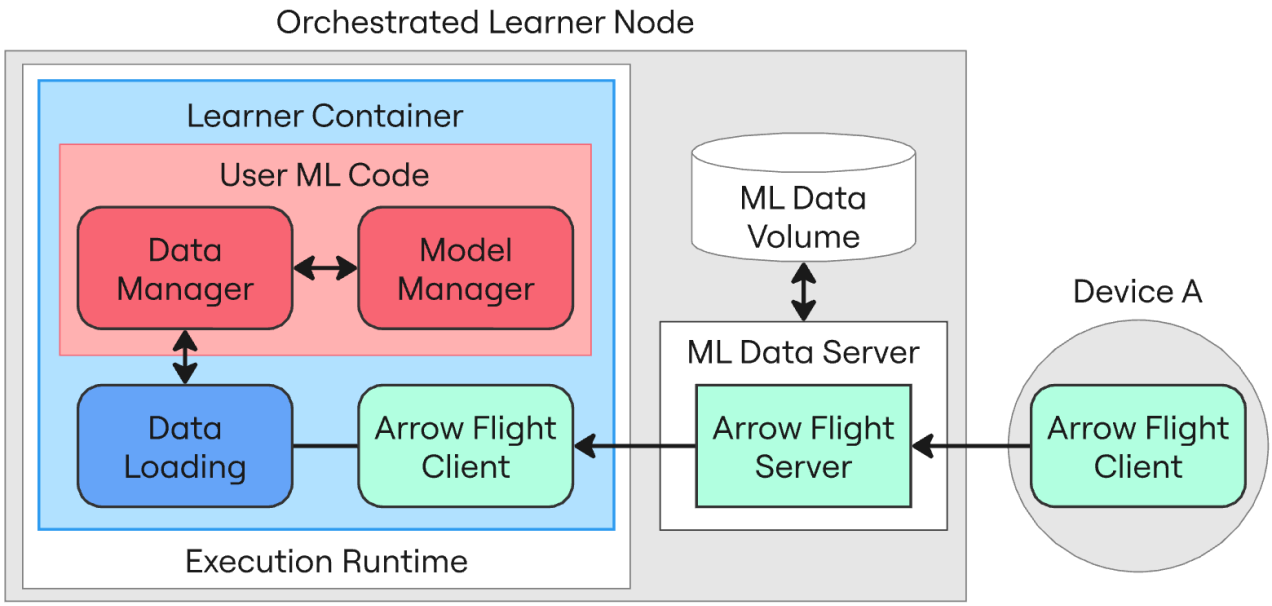

Figure 4.3: FLOps Local Data Management Structure

Figure 4.3 depicts the architecture of FLOps' local data management. The important concretions compared to the simplified version in Figure 3.4 from 3.2.2 are as follows: The learner container and the data-providing device must have an Arrow Flight client installed and connected to the Arrow Flight server in the ML data server. The data loading component in the learner uses the Flight client to retrieve the matching data. It gets added by the builder service and is not part of the user's ML code. The data and model manager utilize this loaded data. Note that the figures in this subsection depict a single data source device for optimizing page space. Arbitrary many devices are supported by this setup.

Figure 4.4 depicts FLOps' local data management processes in more detail. Firstly, the unique data resides on device A. The device can store its data in an arbitrary format. The device trusts the learner node in its proximity and transfers its data A to its ML data server via Flight. For this, it needs to convert its data into Parquet format. Secondly, the ML data server receives this streamed data and stores it locally on the learner node in a dedicated ML data volume. This volume now contains several different Parquet files from different sources. It uses Parquet files based on the last subsection's recommendations.

The next sequence of steps starts from the user's data manager. During its initialization (A1), when the leaner service is run, it triggers its prepareData function (A2). This method calls the wrapper/adapter function loadDataset (A3). The loadDataset function calls the loadDataFromMLDataServer function (A4). This function is not available for users during development. It is instead injected by the builder service, so it is exclusively available in the learner image's augmented FL code.

The Data Loading's loadDataFromMLDataServer function contacts its Flight client to request all matching files from the local ML data server (A5). The match is performed based on the user's project SLA. This SLA includes a dataTags key that is a list of string tags. When devices send over their files, they must provide a data tag. The ML data server will store these data files in the format seen in the Figure's legend. The name starts with a singular data tag and ends with a unique hash based on the file's content. Users and data providers need to cooperate to ensure that these data tags match. The ML data server takes all files which name's prefix matches the user requested SLA data tags and streams them over into the learner container (A6). In the example shown, only data files A and B have matching data tags. The Flight server streams both over to the data-loading component.

Now that the matching data files are available inside the learner container, they must be transformed to fit the user's needs. The data loading component converts its received Parquet files into Arrow format (A7) for better in-memory data management. Usually, ML code expects and works with whole datasets instead of multiple split ones. For this reason, the data loading component also merges its received files into a

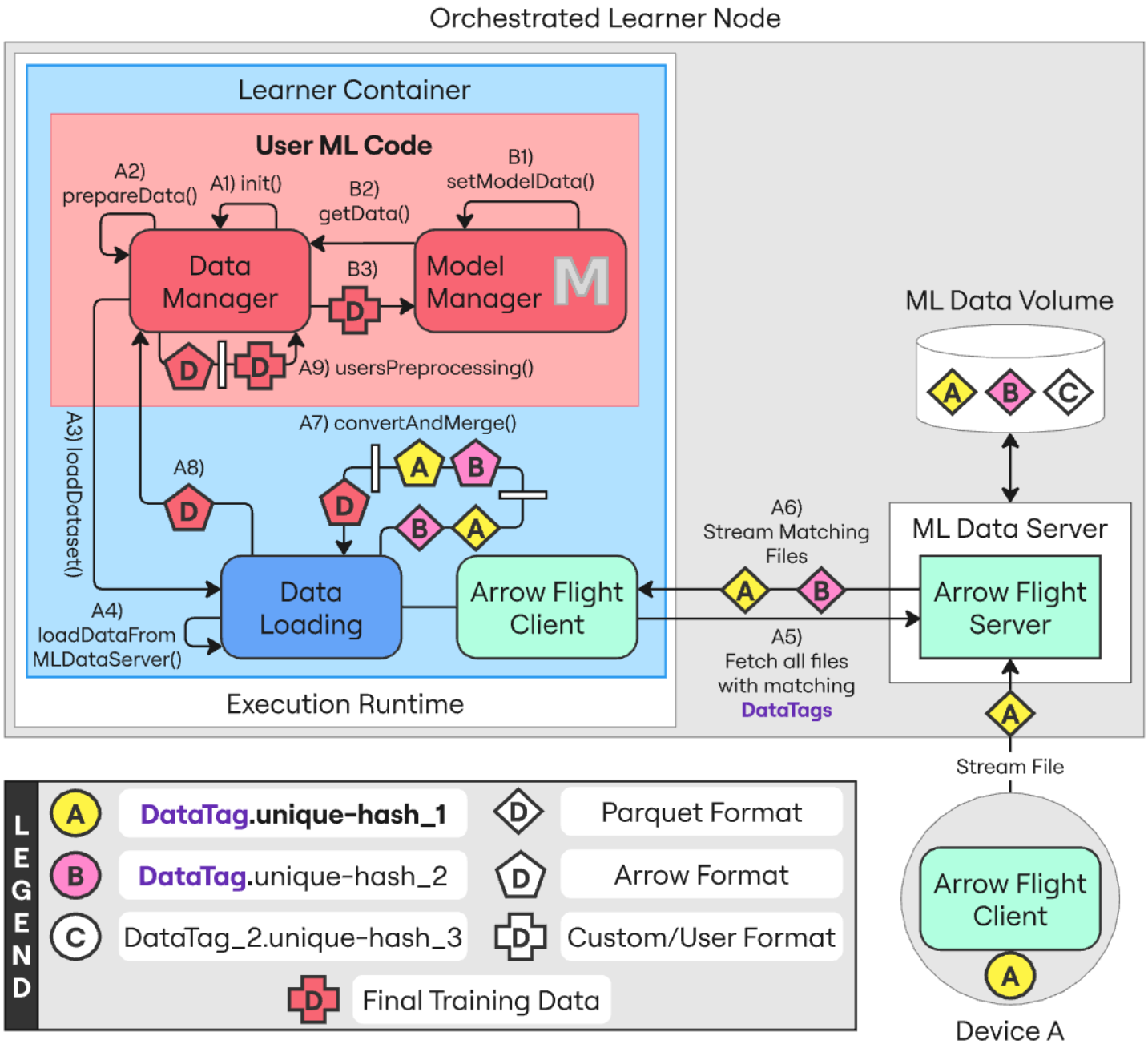

Figure 4.4: Detailed FLOps Local Data Management Structure

single dataset (A7). The data loading component then sends this dataset to the user's data manager (A8). The data manager now has access to the necessary dataset and can perform custom preprocessing and transformation steps (A9). Users can freely configure and implement this preprocessing to ensure the retrieved data is usable for their ML model.

When FL training starts, the user's model manager will initially set the model data (B1). Its getData method (B2) contacts the data manager and returns its prepared compatible dataset (B3). In conclusion, these steps enable FLOps to manage and provide real local FL data to diverse user ML code for training and evaluation.

4.3.4 Mock Data Providers

Coordinating and managing real data providers while developing or testing can be challenging. FLOps offers its own mock data providers to make these processes more convenient. These optional services can be deployed on learner nodes via the orchestrator. They split datasets into partitions and send them to the ML data server exactly as real devices. It is enough to run them once to populate the local learner nodes with data. These mock data providers also act as implementation examples for real devices when converting data, setting up a Flight client, and communicating with an ML data server. The code is available here [33]. Subsection 4.1.1 shows the API endpoint for creating such mock data providers. The FLOps Helper SLA for mock data providers looks as follows:

```
{
    % The ID has to match the user's orchestrator ID.
    "customerID": "Admin",
    "mock_data_configuration": {
        % This value can be any dataset name available in Hugging Face.
        "dataset_name": "cifar10",
        "number_of_partitions": 1,
        "data_tag": "cifar10"
    }
}
```

Figure 4.5 depicts FLOp's mock data provider. The mock data provider runs as an orchestrated service in the container execution environment, similar to the learner service. It uses the requested Hugging Face dataset name to download the dataset via Flower Datasets. The data provider uses Flower Datasets to split this monolith dataset into multiple partitions. The user defines the number of partitions. Each partition is individually sent to the ML data server as if real edge devices were contacting the

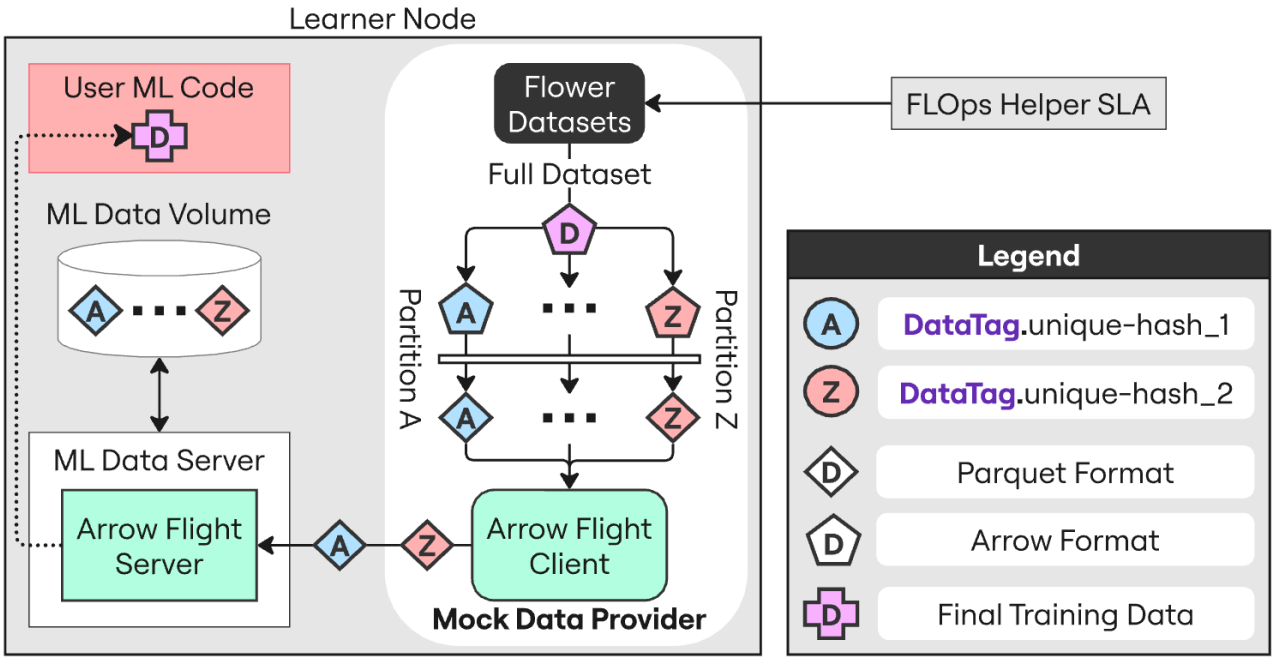

Figure 4.5: FLOps' Mock Data Provider

server. The ML data server stores each partition separately in the ML data volume. Ultimately, these partitions will be merged and preprocessed to be fit for training.

4.4 MLOps via MLflow

This section showcases how FLOps enables MLOps. The first subsection discusses its MLOps components and how they work together. The second subsection shows briefly how the GUI looks and works. Many details were already mentioned in previous parts of this work. Thus, this subsection will not repeat them but provide new insights or concretions.

4.4.1 MLOps Components & Architecture

This subsection builds on top of 2.2.3 and 3.2.5. Figure 4.6 shows FLOps' MLOps architecture. MLflow powers many FLOps' MLOps capabilities. After every training round, the aggregator logs lightweight artifacts like metrics, parameters, tags, or runs. In addition, the aggregator stores exactly one global model copy locally. After every round, the aggregator checks if the new model's performance is better or worse. The

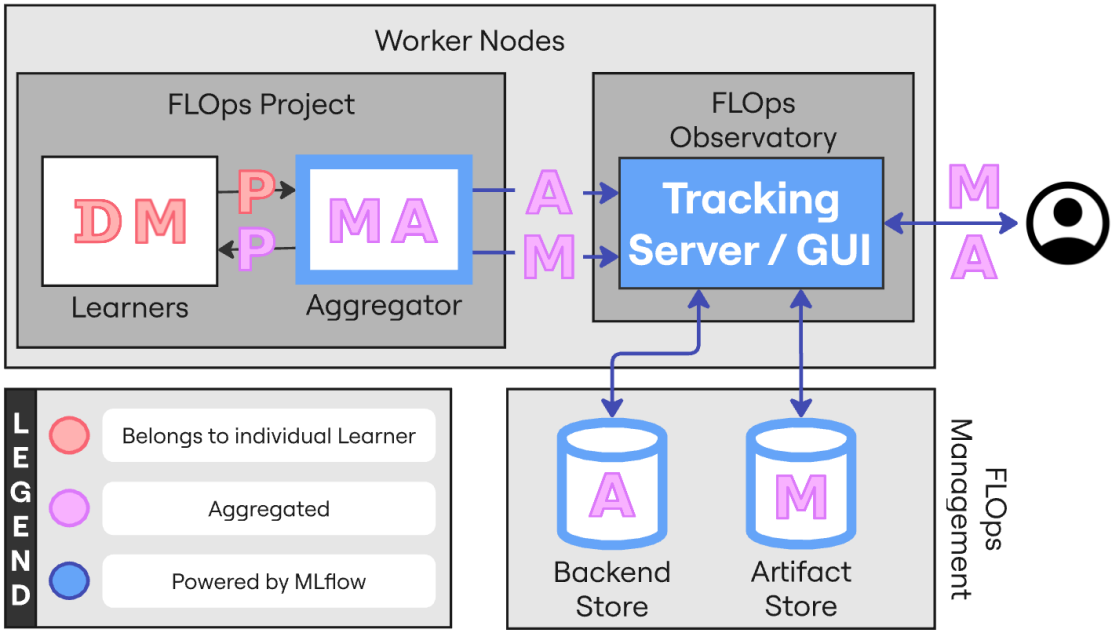

Figure 4.6: FLOps' MLOps Architecture

aggregator will update its local model if the new model is better. At the end of the last training round, the aggregator sends the trained global model to the artifact store.

An MLflow run represents an individual execution of (usually ML) code. Each run can collect various pieces of information, such as metrics, hyperparameters, or custom tags. These lightweight elements are represented as A in the Figure. An MLflow experiment gathers multiple runs. FLOps maps these MLflow terms directly to FL. An experiment becomes an FLOps project and runs are FL training rounds.

The aggregator logs everything besides local elements over the tracking server. The tracking server works as a proxy for artifacts. Thus, any access to any logged objects goes through the tracking server. The tracking server itself does not have any state. Its GUI showcases the stored elements in the backend and artifact stores hosted via the FLOps management. Note that these stores can be deployed and scaled individually onto different machines. There are various ways of setting up and provisioning MLflow components. For example, the backend store can be a local directory, a remote database, a cloud file server, or blob storage. The backend store hosts lightweight elements, and the artifact store hosts heavy-weight elements such as models or images. FLOps currently uses a MySQL database for the backend store and a vsftpd (very secure FTP daemon) server for the artifact store.

It is noteworthy that no MLOps logging takes place on the learners. Only the

aggregator uses these techniques. This approach works as expected regarding concrete FL metrics and models. MLflow also provides a way to track system metrics, which FLOps uses. These metrics only capture information about the aggregator, not the connected learners. No information belonging to individual learners gets logged. Furthermore, FLOps ensures that users can only access their own recorded artifacts. FLOps explicitly upholds these separations to minimize possible privacy hazards and attack vectors.

4.4.2 GUI

This subsection showcases a few key GUI elements. FLOps uses MLFlow's GUI and does not modify it. Therefore, this chapter only provides a brief selection of impressions of the GUI. Excellent further details are available directly at MLflow [72].

The first screenshot 4.7 shows MLflow's experiments overview page. The left column lists all recorded experiments/projects. Only a single one is currently selected. Details about it are displayed to the right. Multiple experiments can be selected simultaneously to view their combined contents. The centerpiece offers a table view of the different FL rounds. Users can customize and sort this table to their liking. Each table row depicts a single FL round, when it was recorded, and its duration. Only the best round contains a logged model. In this example, the best round was the last (10th) round.

Figure 4.8 shows the detailed view of a single recorded experiment/project. Currently, FLOps focuses on the model's accuracy and loss. The centerpiece of the screenshot shows the evolution of both across different FL rounds. It shows that the model's accuracy improved, and its loss decreased over time. FLOps users can access this GUI during FL training and observe how this FL-rounds table grows in real-time.

The third screenshot 4.9 depicts concrete FL round details. These details include general information such as if and what model was recorded, the run/round ID, or when the run was created. In addition, it displays custom parameters that FLOps injected, including the user-provided number of clients/learners via the SLA. This page also displays other metrics, such as accuracy or system metrics.

The last screenshot 4.10 shows the logged model details page. The left folder shows the different aspects that were recorded. The model requirements, conda environment, and model (pkl) file are all present. This concrete example showcases a registered model (2.2.3).

MLflow is a feature-rich and well-documented MLOps tool. Its GUI directly supports in-build techniques to compare, analyze, and visualize these logged results. All of these recorded properties can be exported and shared with other people. This thesis does not cover or use all MLflow's (GUI's) features. Further information is available here [72, 68].

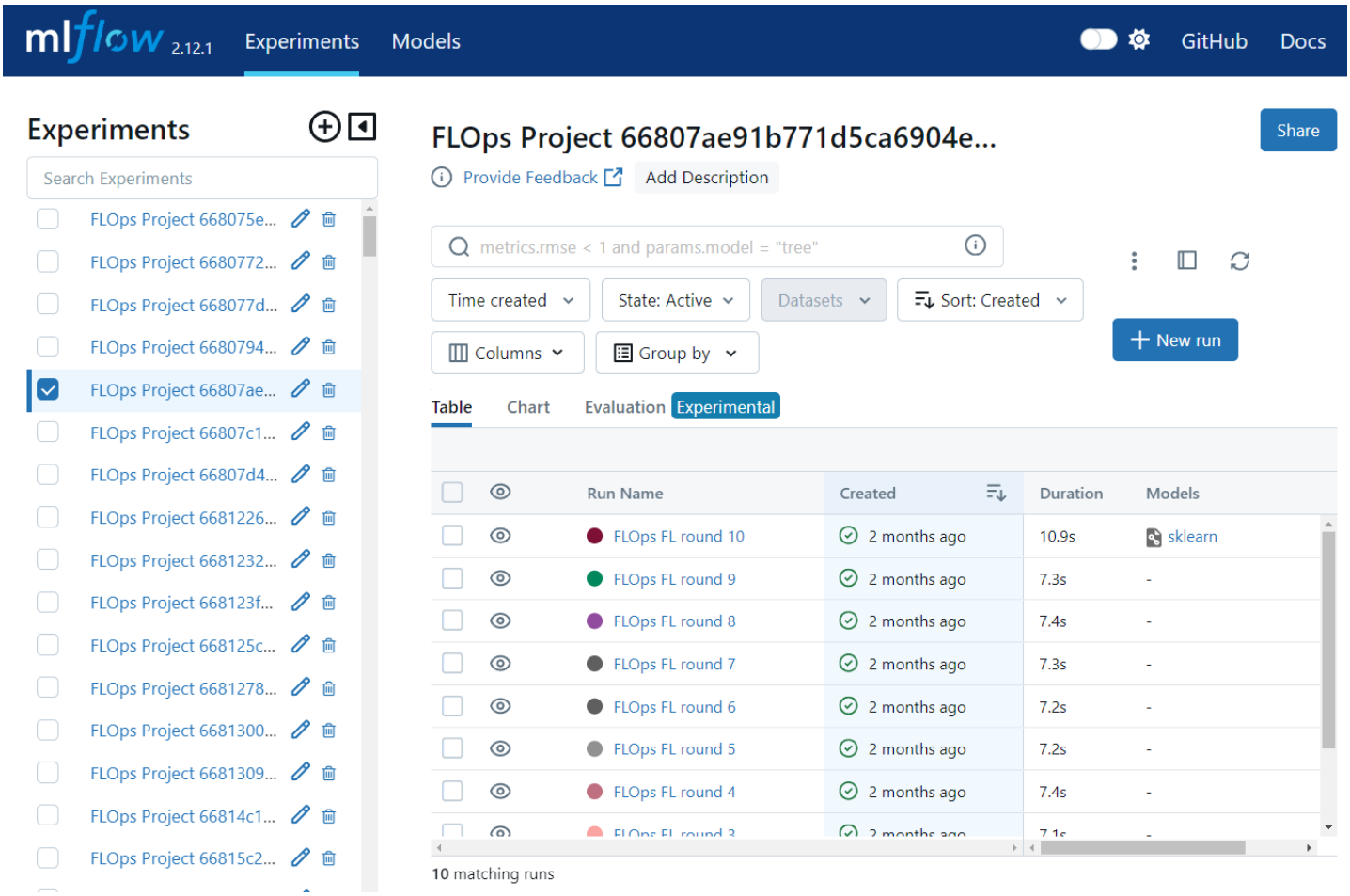

Figure 4.7: MLflow's GUI Screenshot - Experiments Overview

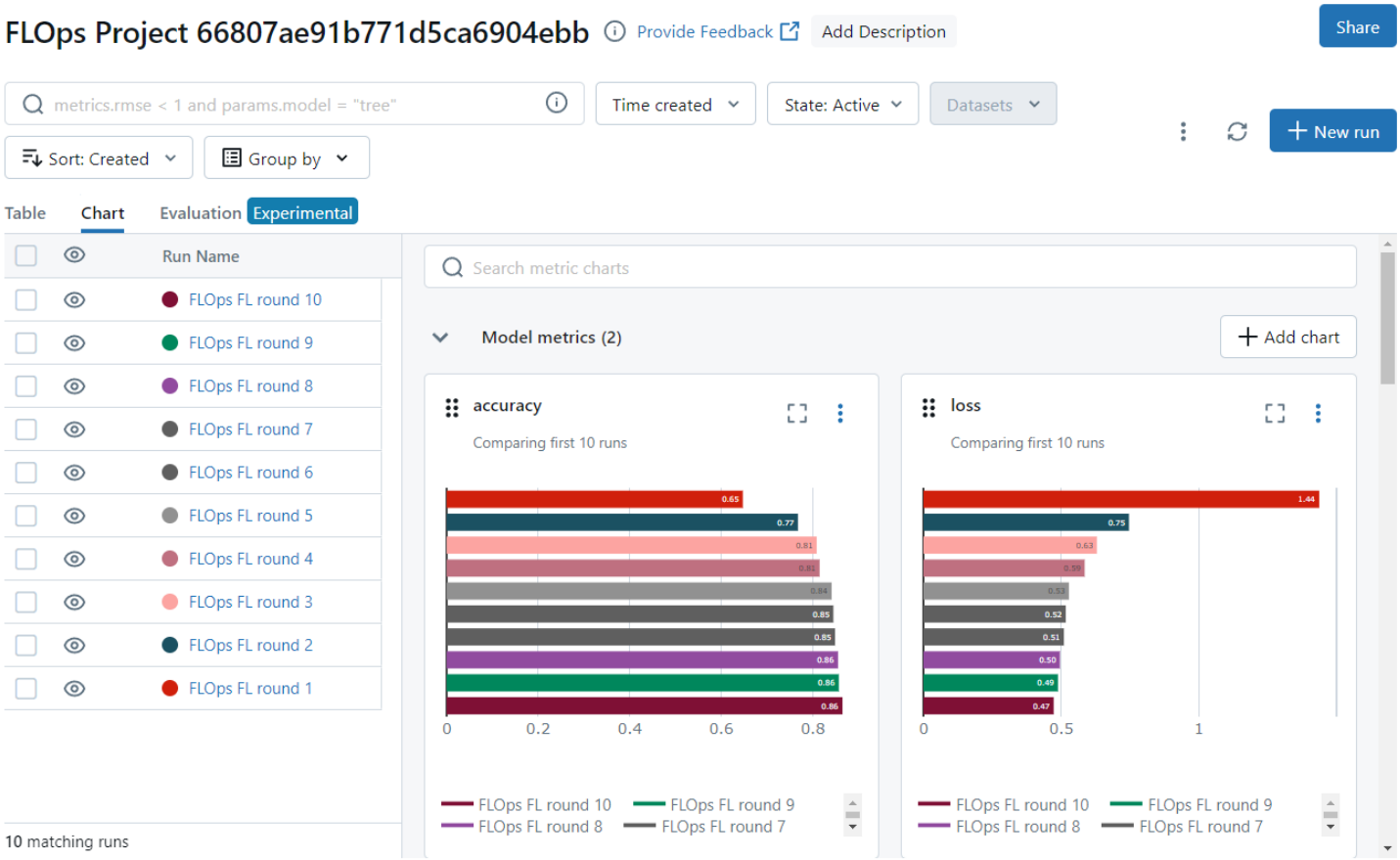

Figure 4.8: MLflow's GUI Screenshot - Experiment Details

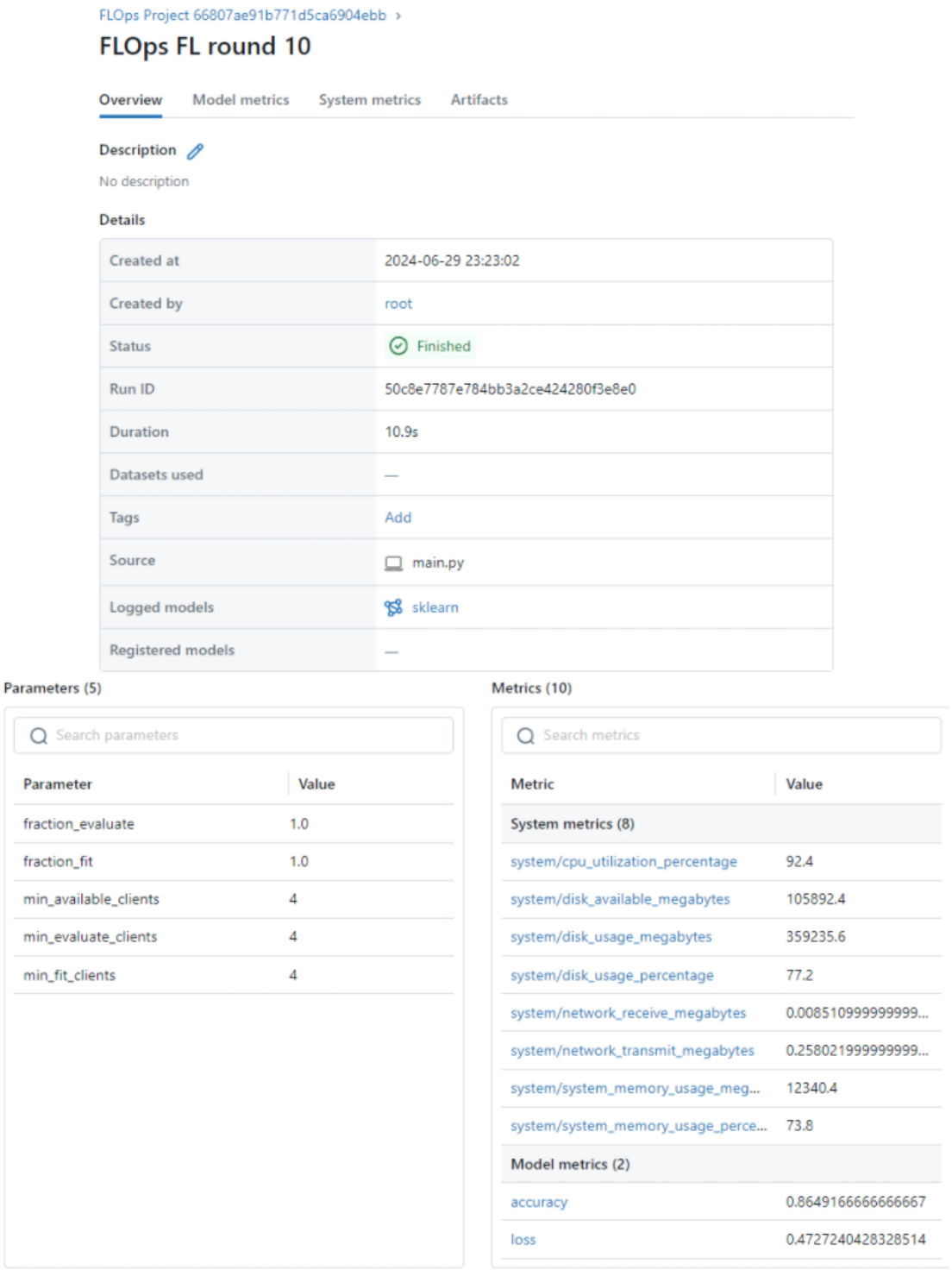

Figure 4.9: MLflow's GUI Screenshot - FL Round Details

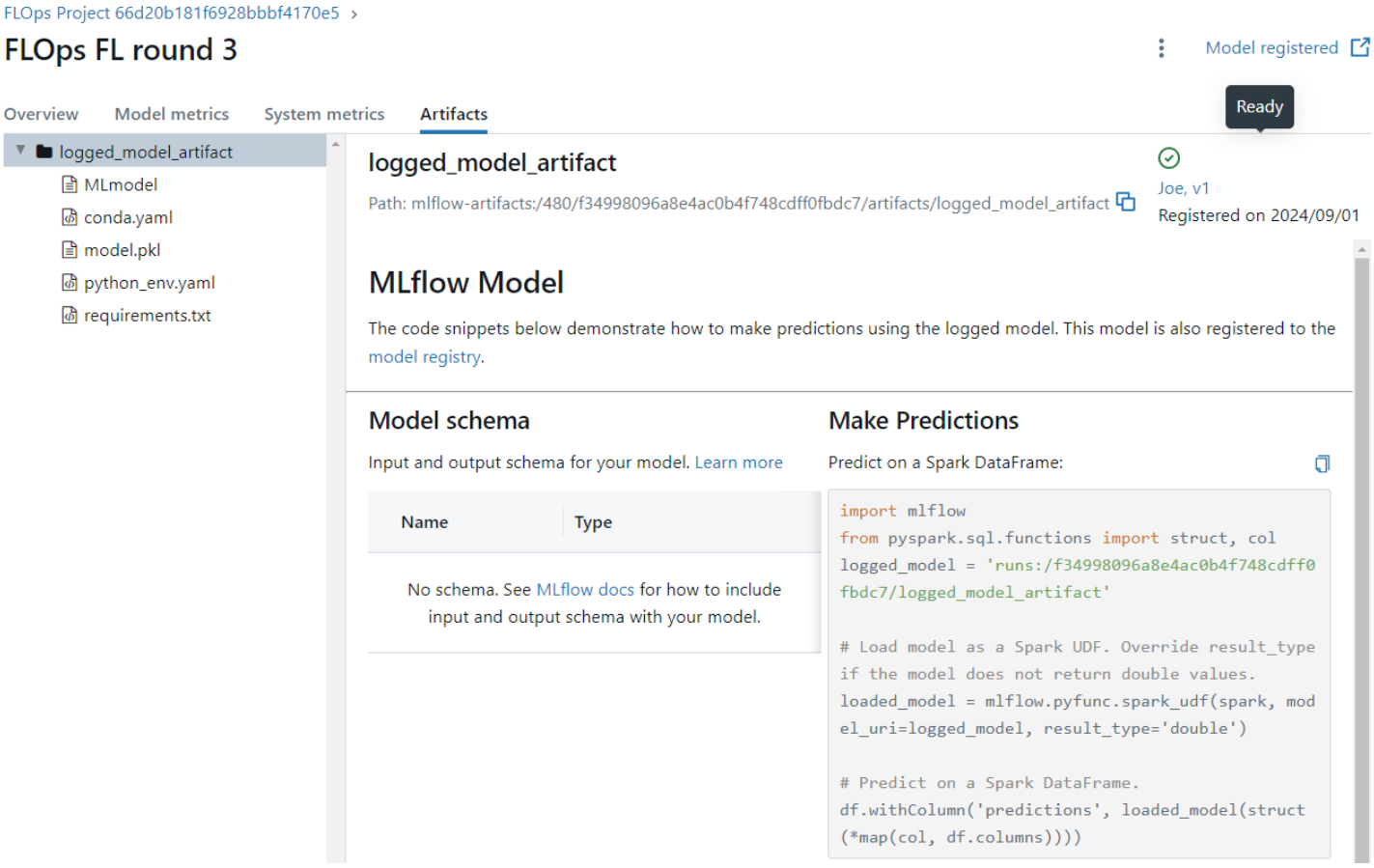

Figure 4.10: MLflow's GUI Screenshot - Logged Model Details

4.5 Clustered HFL

Besides classic FL, FLOps supports (clustered) HFL. Oakestra's three-tiered layout supports geographically dispersed clusters. Each cluster has its own cluster orchestrator and set of worker nodes. This structure naturally alludes to the use of clustered and hierarchical FL. Remembering Figure 3.9, FLOps uses two different types of aggregators for HFL. The root and cluster aggregators are deployed as services on worker nodes to distribute computational load. Only a single root aggregator exists. It can reside in any cluster. Each orchestrated cluster hosts a single cluster aggregator. A cluster aggregator only works with learners inside the same cluster. This type of geographic clustering is why FLOps' HFL is a clustered approach. Root aggregators treat cluster aggregators as plain learners, precisely as in classic FL. Cluster aggregators are a combination between learner and aggregator. Note that Flower does not natively support HFL. Therefore, this approach of realizing HFL via Flower is a custom novel solution.

Figure 4.11 shows the detailed architecture of how FLOps realizes clustered HFL. This figure reuses and expands upon the stylistic conventions seen throughout this thesis, starting from Figure 2.2. Every visible element beside the root aggregator is part of a single cluster. This setup supports multiple clusters. Because the root aggregator interacts with the cluster aggregators as if they were plain learners, cluster aggregators need to offer the same learner interface. The cluster aggregator implements the same learner interface and model manager as user ML code repositories. This approach requires implementing this interface properly and maintaining the state during multiple training cycles. Therefore, the cluster aggregator needs to be able to modify and access the underlying user ML model. This model is the main reference point for model parameters in a cluster aggregator.

At the start of a new training cycle the root aggregator calls the cluster aggregator's fitModel method. It triggers the cluster aggregator's handleAggregator method, which all aggregator types in FLOps have. The cluster aggregator performs conventional FL training with its learners and fuses new intermediate global parameters (pink P). Then, it updates its model copy stored in the user's model manager. By default, Flower also evaluates the model during training. The custom FLOps aggregator strategy can store and accumulate evaluation results. When the root aggregator requests to evaluate the cluster aggregator, it retrieves the stored values from the strategy and context objects. When the root aggregator calls the cluster aggregator's getParameters method, the cluster aggregator calls its user's model manager getParameters method. The same applies to setting parameters.

In other words, the cluster aggregator mimics a learner by using the same interface, which works on the same user-provided ML model. The main differences between a learner and the cluster aggregator are that the fit model method performs classic FL

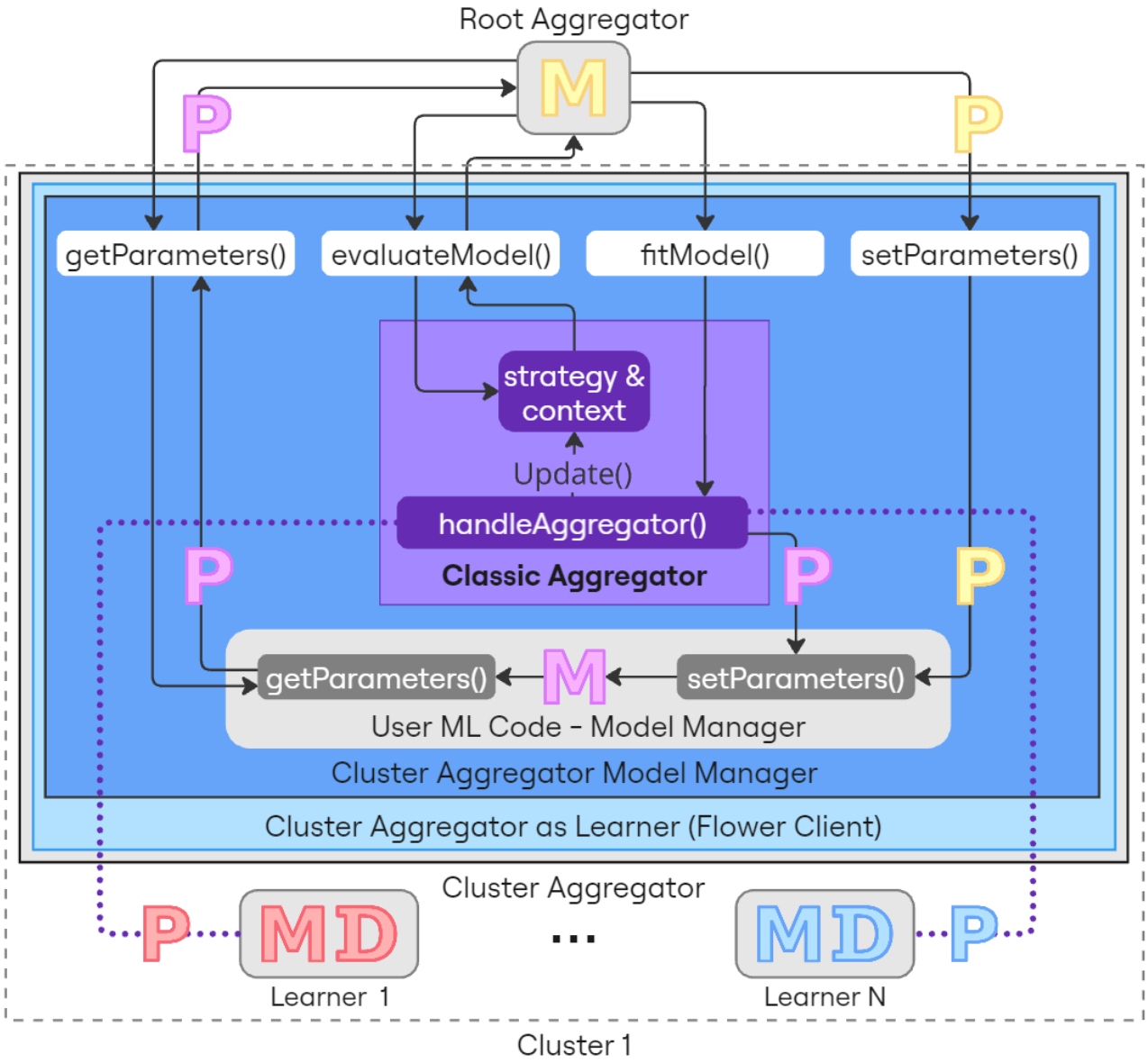

Figure 4.11: FLOps clustered HFL Architecture

training rounds, the evaluate function retrieves recorded results from the aggregator objects, and that the aggregator has no access to data. This way, FLOps can perform clustered hierarchical FL. Note that the underlying code is shared among all aggregator types, thus avoiding several similar implementations. I.e., the same aggregator image gets deployed with different parameters that decide the aggregator's behavior.

4.6 CLI

While developing FLOps, we implemented several different pieces of code to automate tedious repetitive manual tasks. We decided to share and offer these custom auxiliary scripts by combining them into a single CLI tool called OAK CLI [75]. We have steadily improved it over this work, which resulted in various features. Because of envisioned rapid future changes, we will not discuss concrete technical details but provide a broad overview of its capabilities. We decided to discuss this CLI as part of this work because FLOps can be considered DevOps for FL, and DevOps also includes techniques to improve development workflows. One way to support users and developers is to help them use the target application more conveniently, for example, via a CLI tool. Notably, Oakestra has a custom early-stage work-in-progress GUI/Frontend dashboard application [76] and a minimal CLI tool. The OAK CLI is independent of both these components and replaced the legacy Oakestra CLI tool.

We carefully considered what libraries to use for this CLI to be flexible and easily extendible. The OAK CLI initially used Python's argparse [86] and argcomplete [85]. Argparse is an established feature-rich standard library. Its downside is that it requires a lot of boilerplate code, especially when the argument structure is a complex nested hierarchy. Argcomplete enables auto-/tab-completions of CLI commands. Enabling this functionality in a pure Python tool can be tricky and require specific workarounds. OAK CLI now uses Typer [99]. Typer requires minimal decorator augmentations to enable CLI applications. This is possible because Typer smartly utilizes available Python-type hints in function signatures. It automatically comes with auto/tab completion and includes highly readable, pretty UI features powered by Rich [91]. Rich is a prominent library for Python terminal formatting. Typer is built on top of Click [21], one of the most popular Python CLI frameworks. The downside of Typer is that it is still in relatively early development. It lacks a first major release version. It has some minor cutbacks compared to argparse, but discussing those would bloat this work.

We were motivated to create this CLI to alleviate the following challenges. FLOps does not offer a tool besides the OAK CLI to interact conveniently with its API. External tools like Postman are necessary to do so. Oakestra components need to be prepared and launched manually. Interacting with its API is possible via external tools or its

early-stage dashboard. We needed to work on several devices while developing FLOps, especially its HFL features. Each machine must be appropriately configured and set up to enable Oakestra workloads. This setup included installing dependencies such as Docker, Golang, and other custom aliases and scripts to launch, clear, and restart Oakestra components. In addition, developing and observing FLOps was heavily bound to the observatory features provided by Oakestra's dashboard application. Accessing and working with this dashboard was cumbersome even during local development on a single machine because of repetitive click-based tasks such as mandatory logins. We automated this by creating auto-clicker scripts via Selenium. Accessing this flaky dashboard on remote devices behind firewalls required even more manual steps, such as code modifications and multiple SSH tunnels. We only required the dashboard to get an overview of the deployed application and services and their statuses and logs. The OAK CLI satisfies these needs on its own, thus bypassing our need for the dashboard while automating manual steps. As a result, this CLI significantly accelerates the development and usage of FLOps and Oakestra.

4.6.1 CLI Requirements Discussion

The OAK CLI needs to satisfy the following requirements:

Interface for APIs
The CLI should interact with the APIs of FLOps and Oakestra to alleviate the need for users to use external tools, know all API endpoints, and know how to interact with them appropriately. The key activities the CLI should support are managing applications and services in Oakestra and start projects in FLOps. For example, if the user wants to create a new application in Oakestra he first needs to login. For the login, the user needs to know the login URI, create a fitting request, send it, and extract the received bearer token for authentication and authorization. Only afterward can users prepare their application SLA, add their token, and send it to the application POST endpoint. Instead, the CLI should offer a single command so that users only need to provide their application SLA. The CLI will perform the login and handle the API interactions.

Observability
FLOps handles many different applications and services concurrently. Especially during development, it is crucial to observe whether components behave as expected and identify unexpected errors or behavior as quickly as possible. For this, the CLI needs to provide its users with an overview of the current state of applications and deployed services. This overview should include comprehensive information about

these components, such as their current status and critical properties and details. It is crucial to provide timely information to the user. The CLI should support a way to observe the current situation close to real time.

Accelerated Workflows via Automation
The more manual, tedious, repetitive tasks can be accelerated via automation, the more high-quality work can be done and less frustration generated. The tool should provide ways of installing dependencies to make Oakestra's and FLOps' setup quicker and easier on new machines. The CLI should handle starting, stopping, restarting, and rebuilding Oakestra and FLOps components to speed up and simplify development cycles. In addition, it should be a common place to host valuable additions from different individuals, including aliases or scripts. The CLI should be easily modified and extended to accompany future user demands.

Based on these requirements, it is easy to see that this singular tool has many responsibilities to uphold and features to offer. These features are of no equal interest to all its possible users. Casual end-users require and demand other features than administrators or developers. The usable feature set is also heavily dependent on the machine on which the CLI runs. A machine can be a standalone control plane that only includes the root or cluster orchestrators. It can be a standalone worker node or a hybrid of several scenarios. A machine can also simultaneously include all these mentioned components and be a monolith system. All these options require different needs and, conversely, do not require all features or are not capable of or intended to provide all features. For example, a worker node should be unable to restart the root orchestrator, whereas the root orchestrator should not be able to tinker with sensitive worker node configurations.

A tool that fulfills all mentioned requirements can be divided into several, one for each use case, or provided by a dynamic large single tool. Multiple tools have the benefit of being less overwhelming, leading to smaller and tidier repositories, and users do not need to understand the big picture. The downside of multiple tools is the risk of entangled and divergent dependencies that need to be managed. Such tools will still show interdependencies and coupling, leading to split comprehension. Users, and especially developers, would need to know exactly what tool is responsible for what and how they are interconnected, potentially making things more complex. With split tools, there is the risk of increased code duplication over time, especially if different people develop different parts without understanding related tools. The benefit of a single tool is that everything is in one place and forms a single source of truth. Developers can efficiently work with a single well-structured repository. Users only need to install and update a single tool instead of several. Code and logic can

be more easily reused between the same code base. The downsides of a single tool are the risk of high coupling and increased complexities as the project grows. OAK CLI is a single tool because we explicitly structured it to support low coupling and high cohesion. Its flexible design enables easy separation and extension. Thus, all FLOps-specific functionalities can be easily isolated and converted into a separate CLI tool. We aim towards a modular monolith design that suits a smaller team like ours.

4.6.2 CLI Features

The root command is **oak**. The following is a simplified overview that does not depict or explain every available option and feature to avoid bloating this work because the CLI underlies active change. Most of the subcommands below have shorter aliases to help accelerate typing.

Auxiliary/Meta

help : Shows auxiliary information. This flag is available for every subcommand.

version : Shows the version of the currently installed OAK CLI.

api-docs : Shows a link to Oakestra's Swagger API documentation page.

Applications

a : The pre-command to work with Oakestra applications.

create : Creates one or multiple Oakestra applications based on the provided SLA. The additional optional flag -d automatically deploys all services present in the application SLA. The CLI comes with pre-build common app SLAs that users can inspect and modify to their liking.

delete : Deletes one or all applications.

show : Displays an overview of the current applications. The overview comes in three different variations. The simple view shows a table of all applications with minimal additional information, such as their number of services and ID. The detailed view shows a table with additional information columns. The exhaustive view shows the verbatim underlying JSON object showing all available details. The -v flag (verbosity) toggles between these versions. This overview gets printed a single time. The -l flag (live-display) starts an overview that refreshes itself every three seconds.

Services

s : The pre-command to work with Oakestra services. Remember that services are part of applications. They cannot be created in isolation. Usually, one creates an application and then deploys the services mentioned in its SLA.

deploy : Deploys a new service instance.

undeploy : Un-deploys a specific or all service instances of a single or every service.

show : The show subcommand for services works exactly as for applications, including the verbosity and live-display flags. The difference between them is the displayed information.

inspect : The inspect command shows detailed information of a single service instead of all services. The main benefit of inspecting a service is to see its latest logs. This command also offers the -l (live-display) flag, so users can observe service instance logs close to real-time.

FLOps

addon flops : FLOps is one of Oakestra's addons. The addon subcommand tells the CLI to show addon-related commands. The flops subcommand shows the available FLOps CLI commands.

project : Starts a new FLOps project. The CLI currently provides several pre-build project SLAs. They use different ML frameworks such as Scikit-learn or Pytorch with different datasets such as MNIST or CIFAR-10. They use varying training configurations, including various numbers of learners and training rounds. SLAs are available for classic and HFL projects.

tracking : This command returns the URL to the user's tracking server. If no tracking server exists, it will create one.

mock-data : Launches a mock data provider based on the specified FLOps Helper SLA.

clear-registry : Clears the image registry hosted via the FLOps management.

reset-database : Resets the FLOps management database.

restart-management : Restarts all FLOps management components.

Oakestra Docker Containers

d : Oakestra's control plane runs on two Docker compose files for the root and cluster aggregators. When developing and modifying the code of one of these

components, many developers rebuilt and restarted the entire compose file or even the entire cluster to see the changes take effect. This command allows one to rebuild a single container of an Oakestra compose file directly, which leads to the same result. Rebuilding a single container takes a fraction of the time required to rebuild the entire compose file or cluster. Thus, this command enables accelerated development/change cycles. Users can decide if they want to restart or rebuild an Oakestra container. A –cache-less flag is available for the rebuild command to ensure all changes are propagated.

Worker Node

The w pre-command includes commands that are specific for worker nodes.

ctr delete-images : This command is especially useful when developing images. For example, a worker node must pull the FLOps image builder service image before running its container. When developing this image and pushing it as the same (latest) version, containerd sees that the tag is already present locally and does not pull the updated version. After using this command, one can be sure that the next image used will be the latest pushed one.

Installer

The installer pre-command hosts commands to install and set up necessary dependencies on the host machine. The OAK-CLI uses Ansible to perform the installation.

fundamentals : The fundamentals command will install core dependencies such as Git, Docker, and Golang.

Configuration

The CLI is configurable to avoid overwhelming users with all these available features and only shows applicable and useful commands for their concrete use case. It stores these configurations persistently in a config file via Python's configparser library. Users should set their intended use case via the CLI to unlock the relevant commands. As a result, the CLI can be a fine-tuned tool for different user groups and scenarios.

c : The pre-command to enter the CLI configuration commands.

show-config : Displays the current CLI configuration.

local-machine-purpose : This command allows users to pick their preferred CLI features.

key-vars : Various commands depend on properly configured CLI key variables. If users try to run a command where a key variable is undefined, the CLI will

ask the user to define it first. One example of such a key variable is the path to the cloned Oakestra or FLOps repository. These pointers let the CLI know where Oakestra's or FLOps' compose files reside.

Evaluation
The evaluation commands are very experimental and might get removed from the CLI. We used them to run the evaluations for FLOps.

evaluate : Shows a list of commands to control evaluations. This includes starting manual or automatic evaluation cycles or displaying CSV files that get populated during evaluation. Further details are available in the evaluation chapter.

4.6.3 CLI Showcase

This subsection presents several screenshots of the current OAK CLI. It only focuses on visually interesting command outputs. The CLI is in active development and is subject to change.

```
Usage: oak [OPTIONS] COMMAND [ARGS]...

Run Oakestra's CLI. Many commands are hidden initially to avoid overwhelming new users and to focus
on the reasonable commands for the current configuration. New commands can be un-locked by
configuring your OAK-CLI installation further. If you want to unlock all capabilities of the CLI
configure the purpose for this machine as 'everything'.

 Options
 --install-completion            Install completion for the current shell.
 --show-completion               Show completion for the current shell, to copy it or customize the
                                 installation.
 --help                -h        Show this message and exit.

 Commands
 a                 Command for application related activities.
 addon             Command for addon related activities.
 api-docs          Shows a links to the Swagger api-docs for Oakestra.
 c                 Command for OAK CLI Configuration related activities.
 d                 Command for docker(compose) related activities.
 installer         Install Oakestra dependencies & components.
 s                 Command for service related activities.
 version, v        Shows the version of the currently installed OAK-CLI.
 w                 Command for Worker related activities.
```

Figure 4.12: OAK CLI main help text: oak -h

The first screenshot 4.12 shows the main help text of the OAK CLI. Figure 4.13 depicts an example of a single app displayed in different verbosity modes. Figure 4.14 shows the simple and detailed view of the services from the single app from the previous figure. We ommit showing the exhaustive view because its JSON representation is very verbose. Screenshot 4.15 shows the detailed inspection view of a concrete service with two instances. The grey text at the top shows service properties. The light blue text underneath it presents service instance information. The green text shows the latest logs of each deployed instance.

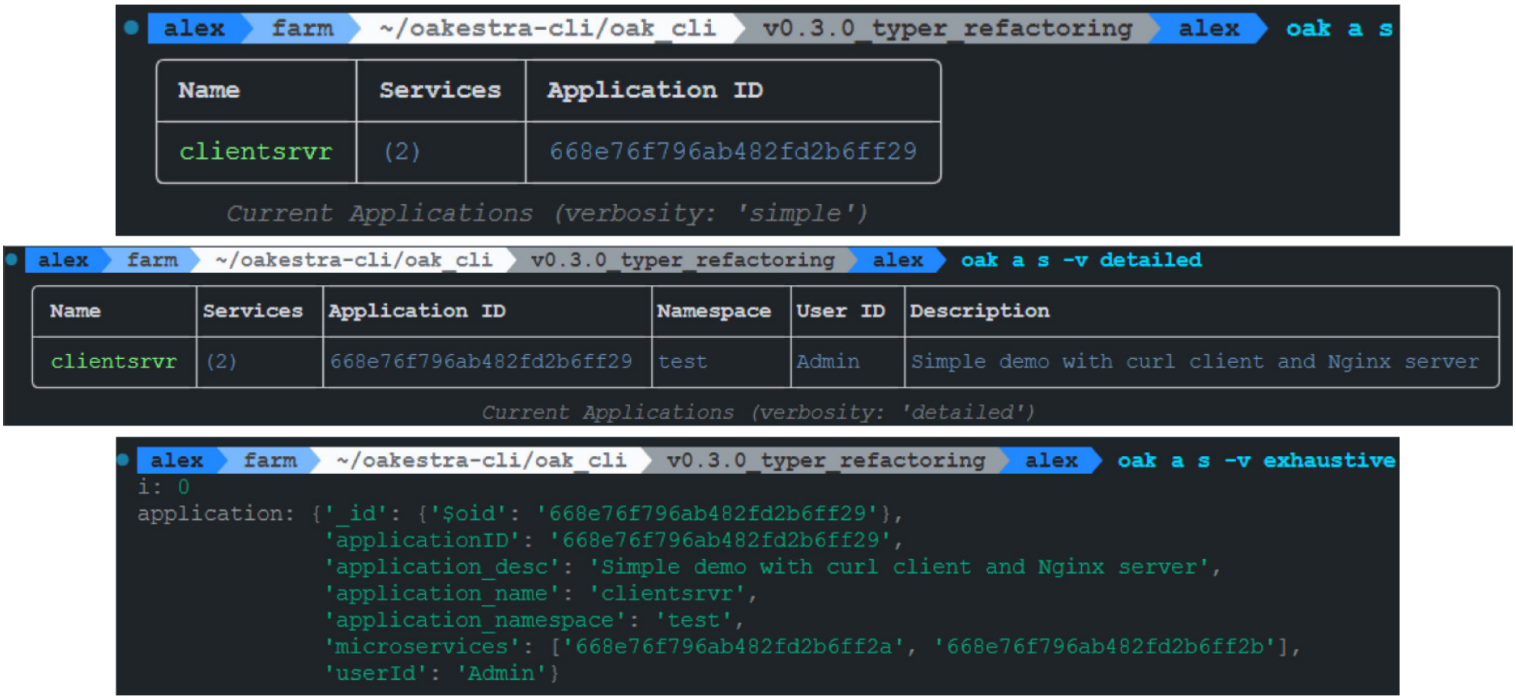

Figure 4.13: OAK CLI Application Views

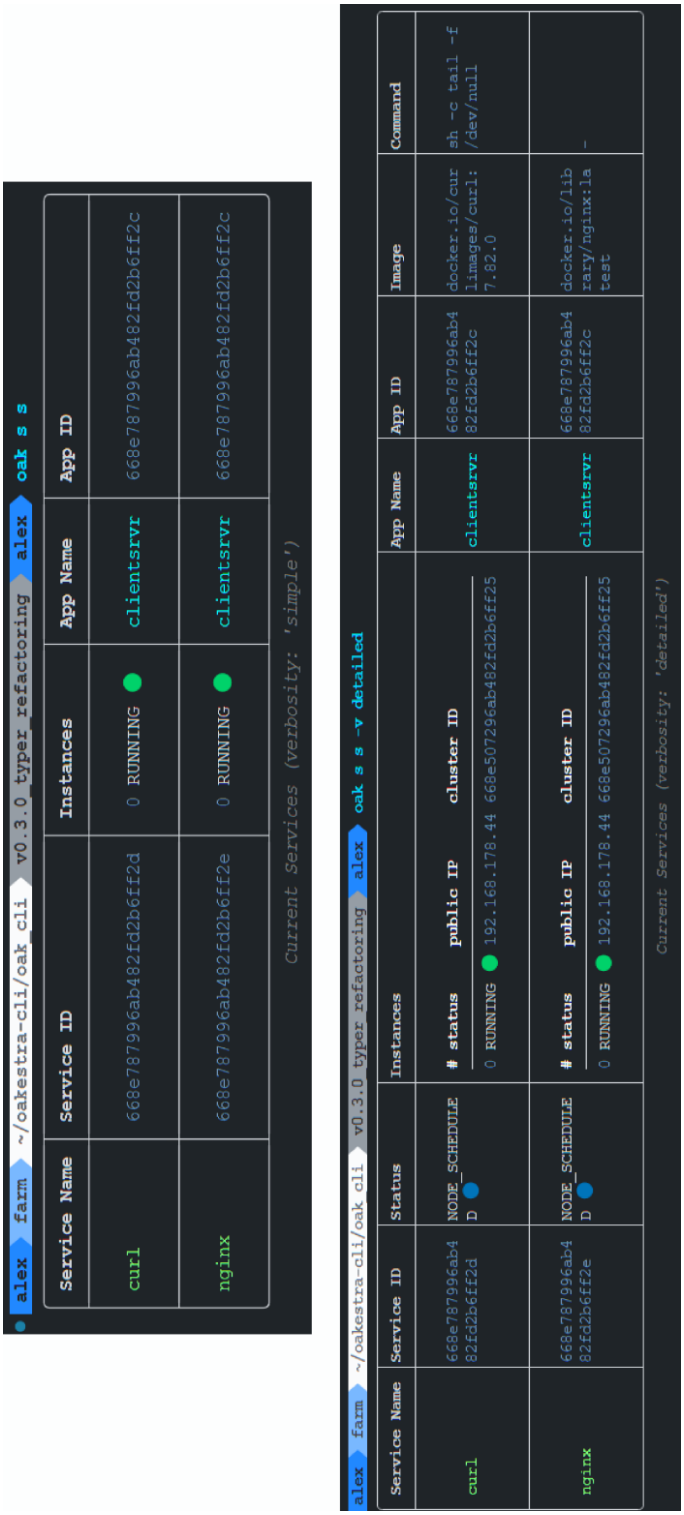

Figure 4.14: OAK CLI Service Views

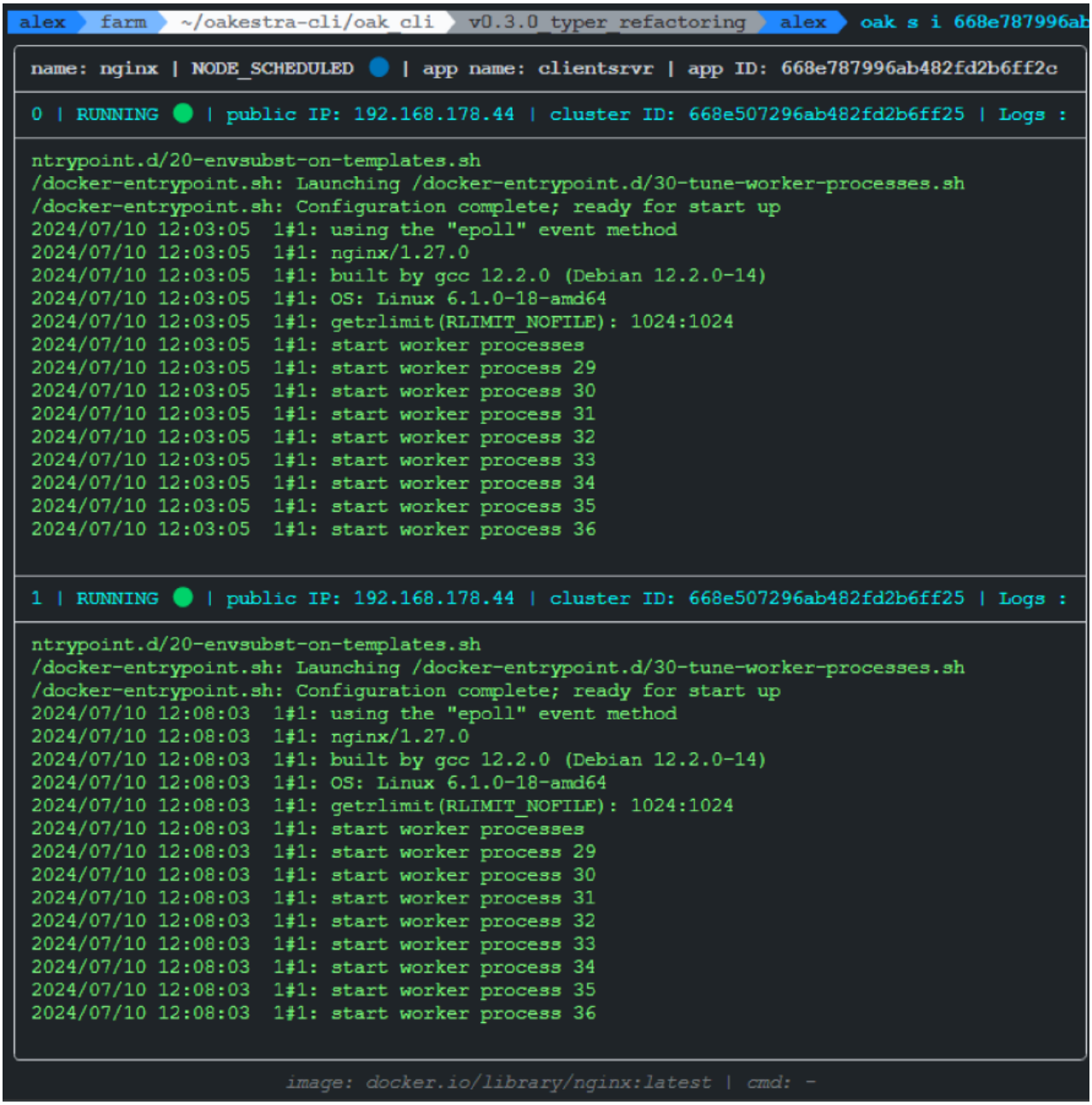

Figure 4.15: OAK CLI Service Inspection

5 Evaluation

This chapter starts by explaining the selected evaluation experiments and the motivation and goals behind them. An overview of the different setups for evaluation and the general procedure follows this. Lastly, we split up the results into different subcategories and analyze them.

5.1 Rationale

FLOps is a new system connecting various custom and pre-made components that power a large set of features and functionalities. This plethora of interconnected parts enables a large number of possible combinations that each could be evaluated. We look at the SLA examples from 4.1.2 and formalize configuration parameters as variables.

Variables defining a FLOps Project

R : The ML Repository. Includes arbitrary complex and different ML code.

F : The ML Model Flavor / Framework.

D : The dataset that is used for training and evaluation.

B : A boolean value indicating if base images should be used to speed up builds during development.

P : The number of supported platforms. Currently FLOps supports two different platforms, which can be used together or in isolation. Thus, P offers three different options.

H : The boolean flag indicating if FLOps should use classic or hierarchical FL.

T : The number of training rounds or cycles.

L : The number of required learners.

S : The underlying setup of devices. We worked with a monolith and multi-cluster setup.

Even if we greatly simplify and assume that all these variables have only two possible assignments, we get an overwhelming number of combinations. Each of these variables are independent of each other. Therefore, each simplified boolean variable combined results in $2^9 = 512$ unique experiments. We even omit several additional possible variables, such as running these tests on ARM or AMD devices, including Raspberry PIs, or using GPU workloads. We aim for representative results, so every experiment is repeated ten times. We call such repeated experiments evaluation cycles. Such cycles take half an hour to several hours to complete. This long runtime, combined with the total number of possible permutations, makes it infeasible to test every single combination. This is a classic case of the curse of dimensionality, where increasing numbers of features result in an exponential increase of their feature space.

We focus on a manageable subset of different experiments that represent comprehensive combinations of these variables to cover a wide range of possible scenarios. In addition, FLOps is a foundational new system. Thus, we aim to verify and evaluate whether it works rather than how competitive it is against modern benchmarks and state-of-the-art training results. We focus on the general workflow and the necessary steps to power such processes instead of optimizing FL training results. Tables 5.1 and 5.2 depict our selected experiments and our reasons for choosing them. The tinted cells highlight the critical changes between experiments. Small classic FL experiments use two learners for three training rounds, and large classic FL experiments use four learners with ten learning rounds. Small HFL experiments use two learners per cluster for three training rounds and two training cycles. In total each learner trains for six rounds. We explicitly try to change only singular variables to facilitate comparisons and understand how individual variables impact the whole workflow.

ID	Experiment	Rationale	Setup	Data	ML	T+L	HFL	B	P
1	Base Case - Simplest	Represents the base case for comparisons and references. If it does not work nothing else should. Should be as simple and lightweight as possible.	Monolith	MNIST	Sklearn	Small	No	No	AMD
2	Simple Large	Same as (1) but with more training rounds and learners. Checks if different, larger project sizes work. Focuses on the impact of project sizes.	Monolith	MNIST	Sklearn	Large	No	No	AMD
3	Simple Base-Images	Same as (1) but uses development BaseImages to speed up builds. Verifies that using base images leads to the same end results, thus they are viable option for development. Checks the build acceleration due to BaseImages.	Monolith	MNIST	Sklearn	Small	No	Yes	AMD
4	Simple Multi-Platform	Same as (1) but supports multiple target platforms. Verifies that FLOps supports multi-platform images. Checks the impact on build times.	Monolith	MNIST	Sklearn	Small	No	No	AMD + ARM
5	Simple Pytorch	Similar to (1) but uses a different ML repository, framework, and dataset. Verifies that FLOps supports various ML repositories, frameworks, and datasets.	Monolith	CIFAR10	Pytorch	Small	No	No	AMD

Table 5.1: FLOps Evaluation Experiments I

ID	Experiment	Rationale	Setup	Data	ML	T+L	HFL	B	P
6	Simple HFL	Same as (1) but uses HFL. Verifies that HFL works in a controlled environment on a single machine with a single cluster. The base case for HFL. If this case does not work no other HFL should work.	Monolith	MNIST	Sklearn	Small	Yes	No	AMD
7	Simple Multi-Cluster	Same as (1) but on the the multi-cluster setup. Verifies that the base case works on the other setup. If it should not work no other experiments make sense on that setup.	Multi-Cluster	MNIST	Sklearn	Small	No	No	AMD
8	HFL on Multi-Cluster	Same as (7) but using HFL. Verifies that real clustered HFL works as intended on multiple clusters.	Multi-Cluster	MNIST	Sklearn	Small	Yes	No	AMD

Table 5.2: FLOps Evaluation Experiments II

5.2 Experimental Setup

We evaluated FLOps on two different setups. The first setup is a monolithic development machine that hosts all Oakestra and FLOps components.

Monolithic Setup

OS : Debian 12 (bookworm) linux/amd64 (bare metal)

CPU : Intel i7-6700K - 8 cores - 4.2 GHz

Memory : 16 GB

Storage : 470 GB

The second setup is a multi-cluster with three identical virtual machines that are part of our chair's infrastructure. We used one machine as a pure control plane that only managed Oakestra's root orchestrator and FLOps management components. The

other two machines represented one cluster, each with its own cluster orchestrator and worker node.

Multi-cluster Setup

OS : Ubuntu 20.04.6 LTS linux/amd64

CPU : Intel Xeon E5-2697A v4 - 4 cores - 2.6 GHz

Memory : 8 GB

Storage : 80 GB

5.2.1 Evaluation Procedure

We extended the OAK CLI to evaluate FLOps automatically. We have added several sophisticated Ansible playbooks and roles that the CLI triggers to run workloads on the monolith or multi-cluster setup via SSH. These Ansible components heavily utilize other OAK CLI commands to clean up experiments and launch new projects. As a side-effect, this automation simulates how real users might interact with the system. Before each experiment run, the playbook ensures a clean experiment environment. This step includes flushing any legacy experiment CSV files, removing all orchestrated applications and services, clearing containerd image caches, restarting the FLOps management components, and flushing its image registry. The playbook starts a new evaluation round once the environment is clean for a new experiment/project. At each round, multiple CSV files are generated. In addition, it starts custom Python daemon scripts on every device that will scrape system metrics and append them to a CSV file every five seconds. Afterward, the playbook requests a specific project configuration that matches one of the selected experiments.

While the project is running, the playbook actively listens for event messages in the FLOps manager logs. Any FLOps project's lifetime can be split into stages such as FL-Actors-Image-Build or FL-Training. A follow-up stage can only occur if the previous one was successful. Once the playbook spots a message indicating a transition to the next stage, it will write this change to a file. The Python daemons actively scan this file and include its stage content in the CSV files. Consequently, the CSV files that include the system metrics also include the respective project stage and timestamp. At the end of each project, the playbook repeats its cleanup steps and starts the next evaluation round. Once all evaluation rounds are finished, the playbook terminates and prints out the sums of individual step runtimes.

One CSV file per evaluation run gets stored and numbered on the local device. In the multi-cluster case, the playbook copies all CSV files from all devices onto the caller

device. At the end of each experiment cycle, a folder of CSV files representing the recorded results is available. To visualize these results, we created several Python scripts to parse, combine, pre-process, and visualize the information stored in the CSV files. We use Seaborn and Matplotlib for this endeavor. As a result, we have uniform, minimalistic Jupyter notebooks that present the recorded information via sophisticated and visually appealing graphs. All this code is available here [75]. Note that we hardcoded several aspects of this code to our concrete use case. Expanding and making this evaluation code more reusable is a prime candidate for future work.

5.3 Results

We split our findings into the following subsections. The basics explain and showcase the simplest base-case and what graphs we use to visualize our findings for all experiments. Afterward, we analyze how different variables change the image build processes. Penultimately, we discuss how different ML repositories, frameworks and datasets perform in FLOps. Lastly, we focus on HFL and verify that our custom novel solution is sound.

5.3.1 Basics

This subsection introduces the different plots we use to visualize all our results. It is more verbose and includes explanations and additional plots than the following subsections, which are similar. This part focuses on experiment (1). All mentioned graphs are also available for all other experiments. Showcasing all of them would heavily bloat this work. Thus, we omit them. These graphs and the underlying CSV files are available in the extended OAK CLI code's evaluation folder [75].

CPU & Memory

Graph 5.1 shows the recorded CPU and memory utilization across a project's lifetime with stage information. This information shows the mean of all ten evaluation runs with a 95% confidence interval. The colored areas represent specific project stages. The graph unveils that the memory utilization stays relatively stable throughout a project and only slightly increases during FL training and the non-base image FL actor builds. Note that the FL-Actors Image Build stage represents the entire image build and push process, which includes the base image and the actor images. Deployment stages represent time frames in which components and services for the next stage are created and deployed via the FLOps manager and orchestrator, but these services/images do not yet start their workloads. Most stages do not utilize much of the available CPU

except during the FL actor deployment stage and FL training, which makes sense because this experiment uses the CPU for training. On average, this simple base case takes 12 minutes to complete on the monolith system.

Figure 5.2 shows a box-violin plot of the CPU utilization for different experiment stages. The largest median CPU utilization occurs in the FL training stage. What is remarkable is that the deployment stage for the FL actors (Aggregator Deployment in the plot) also has high CPU utilization. Multiple services' rapid creation, deployment, and orchestration can explain this. Both image build stages have many outliers, indicating that the build process is highly heterogeneous.

Figure 5.3 is similar to the previous plot but depicts the memory utilization per stage. The FL training stage is the most consuming one. All other stages are below 60% memory utilization except for the FL actors builder and its deployment stages, which have multiple outliers that reach the high 70s. Unlike CPU outliers, which only lead to throttling, memory outliers can lead to out-of-memory exceptions and failures. Thus, it is vital to be aware of such behavior.

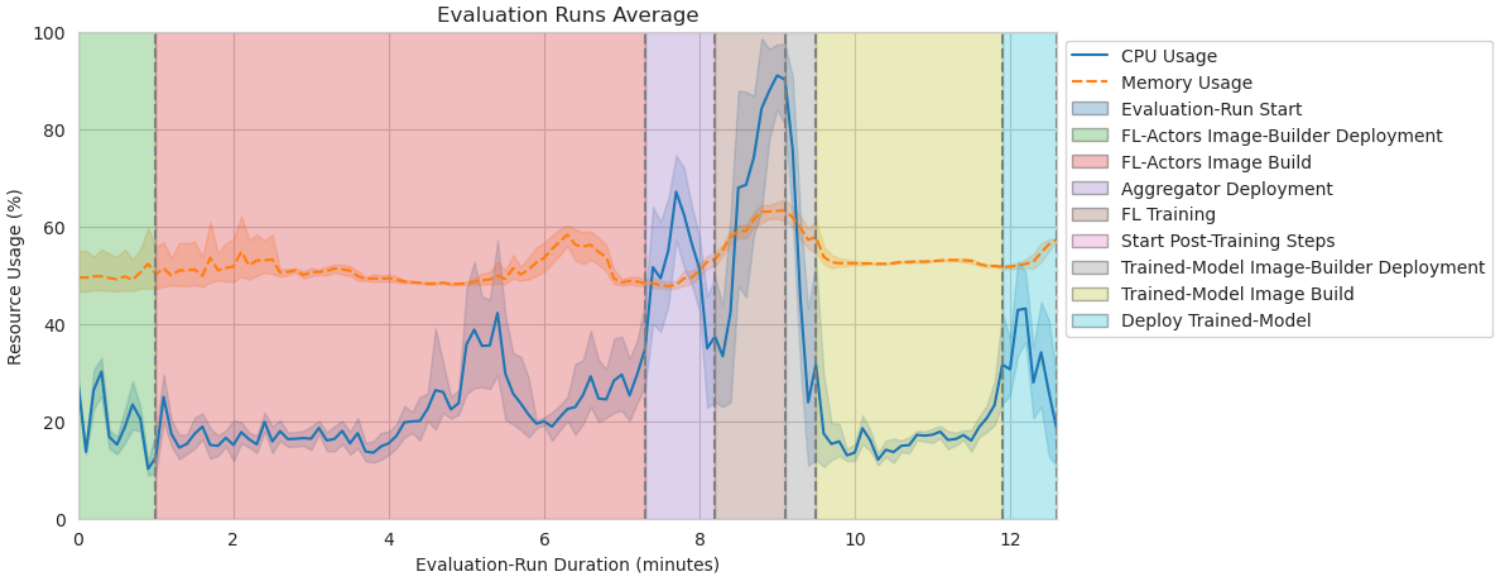

Figure 5.1: Experiment 1: CPU & Memory Utilization

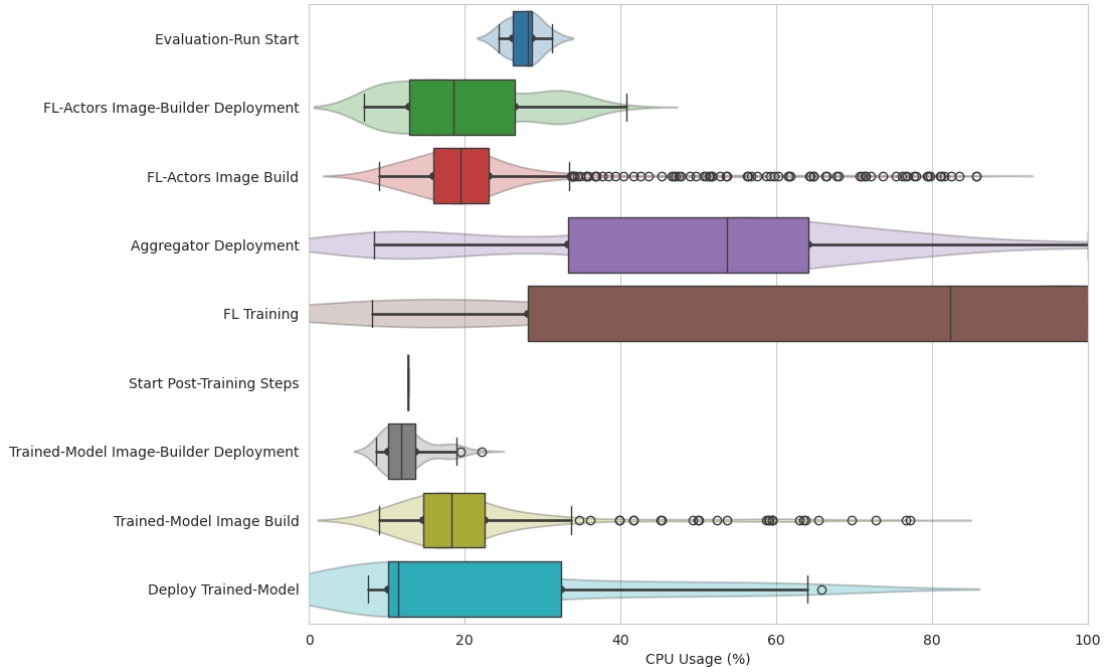

Figure 5.2: Experiment 1: CPU Utilization by Stage

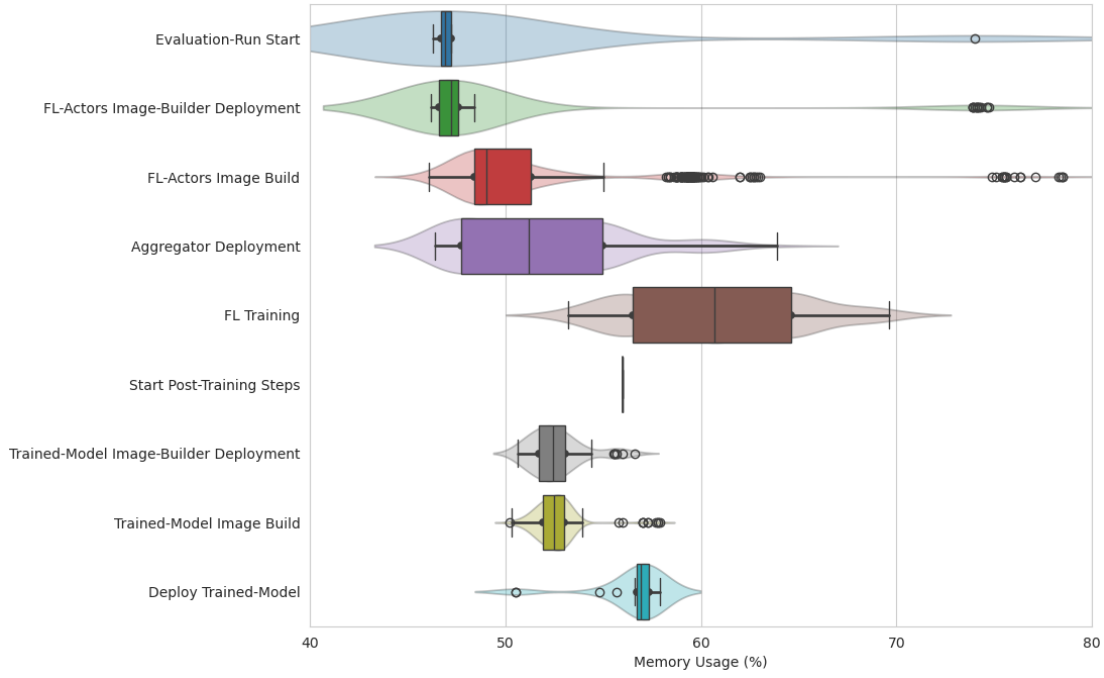

Figure 5.3: Experiment 1: Memory Utilization by Stage

Normalization

Individual evaluations range in duration. The reasons for this can be manifold, such as different current image pull speeds due to local network conditions or remote registry loads. Figure 5.4 shows the memory usage of individual evaluation rounds over time.

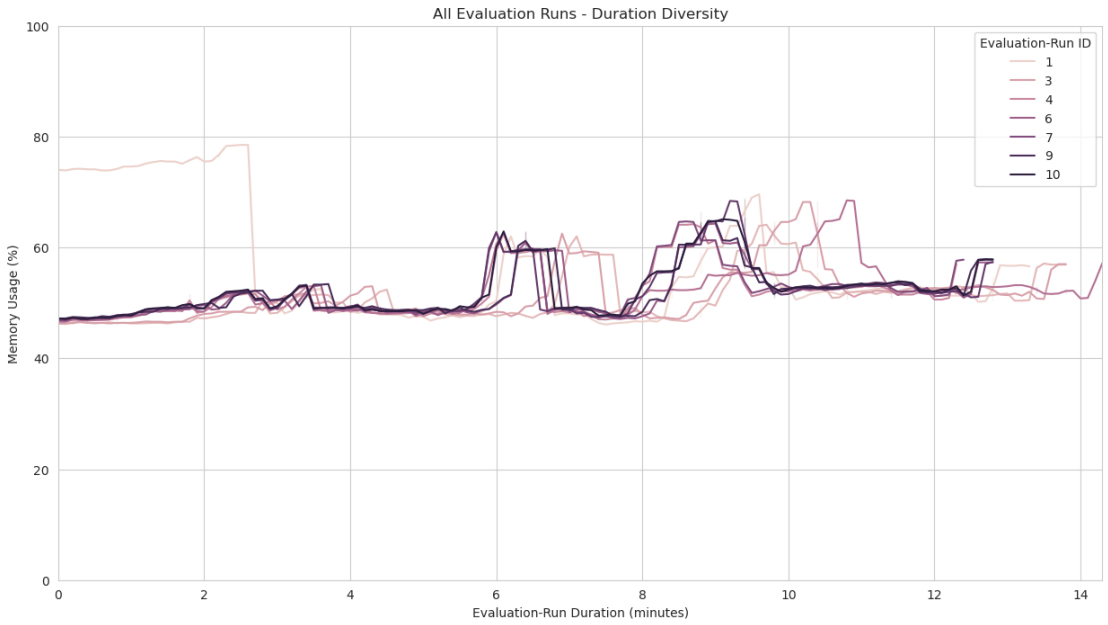

Figure 5.4: Individual Experiment Memory Usage

It is clear to see that the usage pattern is the same but shifted over time. We normalized the average of these rounds to compare and visualize them properly. Otherwise, stage borders become duplicated and overlapped, means and confidence intervals do not lead to meaningful outcomes, and the graphs are more confusing than helpful.

Disk Space

Graph 5.5 shows how the disk space changes over the project's lifetime. There was a total average increase of 14 GB. It starts with the Image-Builder-Deployment stage, where the Image-Builder image is pulled. Many components and dependencies are downloaded and pushed to the registry on the same monolithic device during the FL actors' build process. The jump in disk space in the aggregator (FL actors) deployment stage is because containerd needs to pull these build images. Thus, the monolith will have the same image in the image registry and its local containerd image context. During FL training, the disk space remains the same, which verifies that even when

using FLOps with demanding lengthy training configurations and models, no disk space issues will arise due to its training process. Disk space occasionally goes down due to the system's garbage collection, which is independent of FLOps or Oakestra. The aggregator (FL-actors) deployment stage takes up the most space. The trained-model image deployment stage is minimal compared to the first builder deployment because of the local containerd image storage. Containerd pulls the builder image once and reuses it afterward. This means that dedicated worker nodes that handle several project builds can pull these reoccurring images once and reuse them for all projects. Especially during image-building processes, much space is freed up again due to garbage collection.

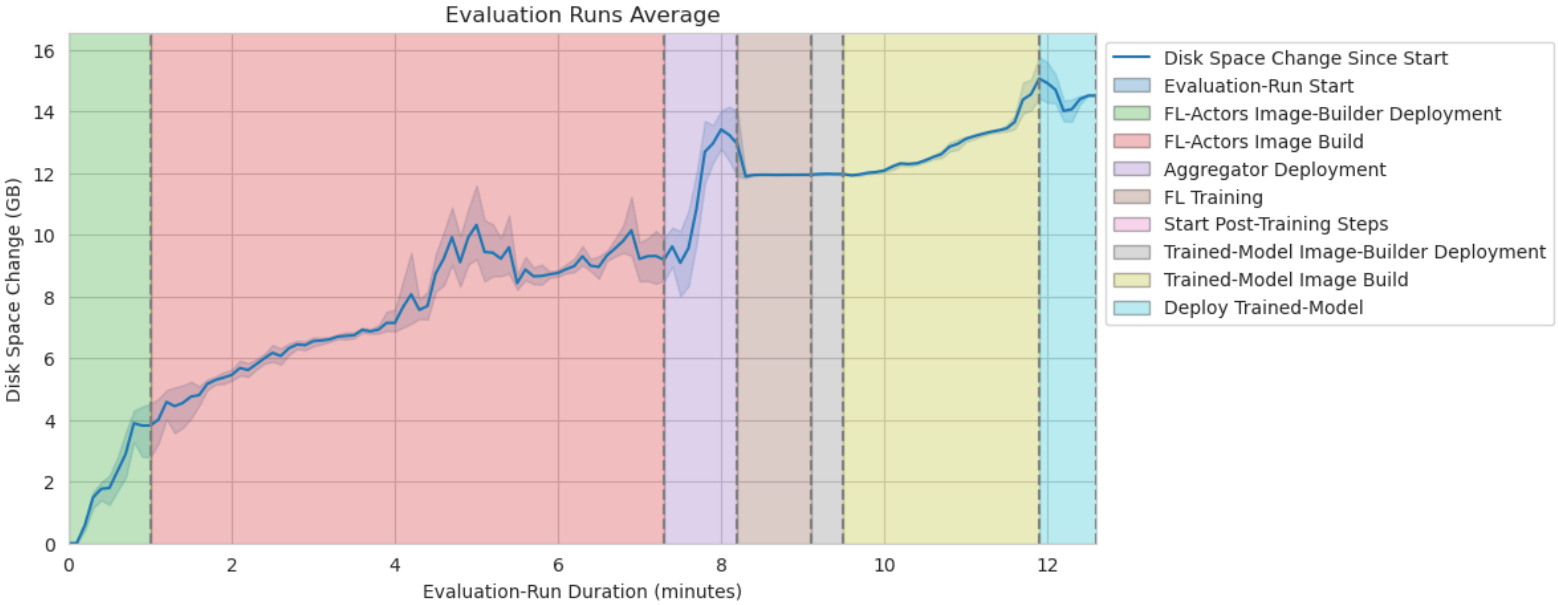

Figure 5.5: Experiment 1: Disk Space Changes over Time

Network IO

Figure 5.6 shows the network loads received and sent over a project runtime. The most significant increases are during built image pushes. They occur around minute five when the base image is pushed, at the end of the FL-Actors Image Build stage phase, and when the trained-model image build stage bleeds into its deployment stage. We omit to present further (violin-box) plots detailing net-IO because this additional information does not lead to any significant insights.

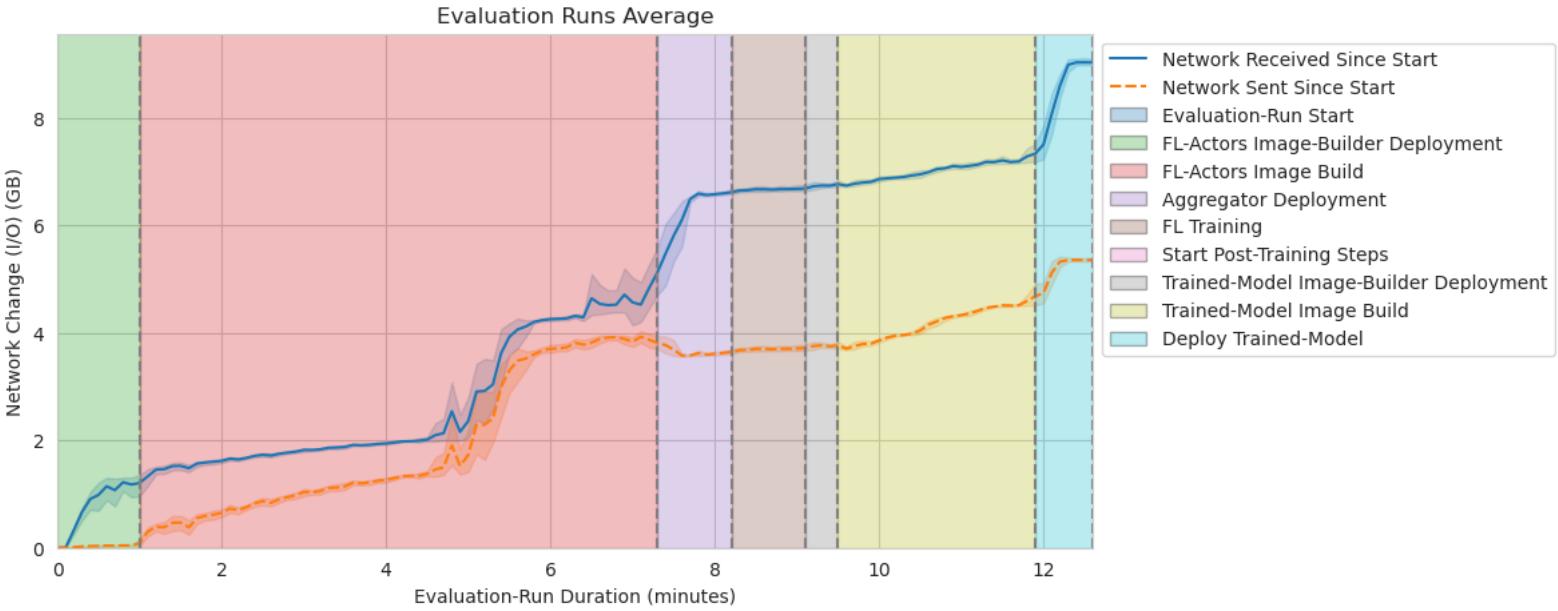

Figure 5.6: Experiment 1: Net-IO over Time

These values should be strictly increasing due to the accumulative nature of network IO counters. This plot shows dips. The reason for this is connected to removed containers. The displayed lines are the sums of all received and sent traffic detectible on all network interfaces. This includes virtual network interfaces of containers. For example, there is a noticeable decrease between the FL-Actors image build and aggregator deployment stages. I.e., between the moment the builder service finishes pushing its images and terminates and before these build services get deployed. The image builder container had its own virtual network interface, which was included in the total sum and represented as a plotted line. Once the container is removed, its virtual network interface is also deleted, and its accumulated net IO will be removed from the sum of the following system metrics scrapes.

Stage Runtimes & Training Results

Figure 5.7 shows each stage's average duration. The FL training stage is relatively short because the training configuration is minimal. The image build stages both take up the vast majority of time. The FL-actors image build process involves more images with complex dependency resolutions. Thus, this build stage takes over twice as long as the trained-model image build. Figures 5.8 and 5.9 show the accuracies and losses of

the trained models after each evaluation round. They prove that FLOps can train ML models in a stable way.

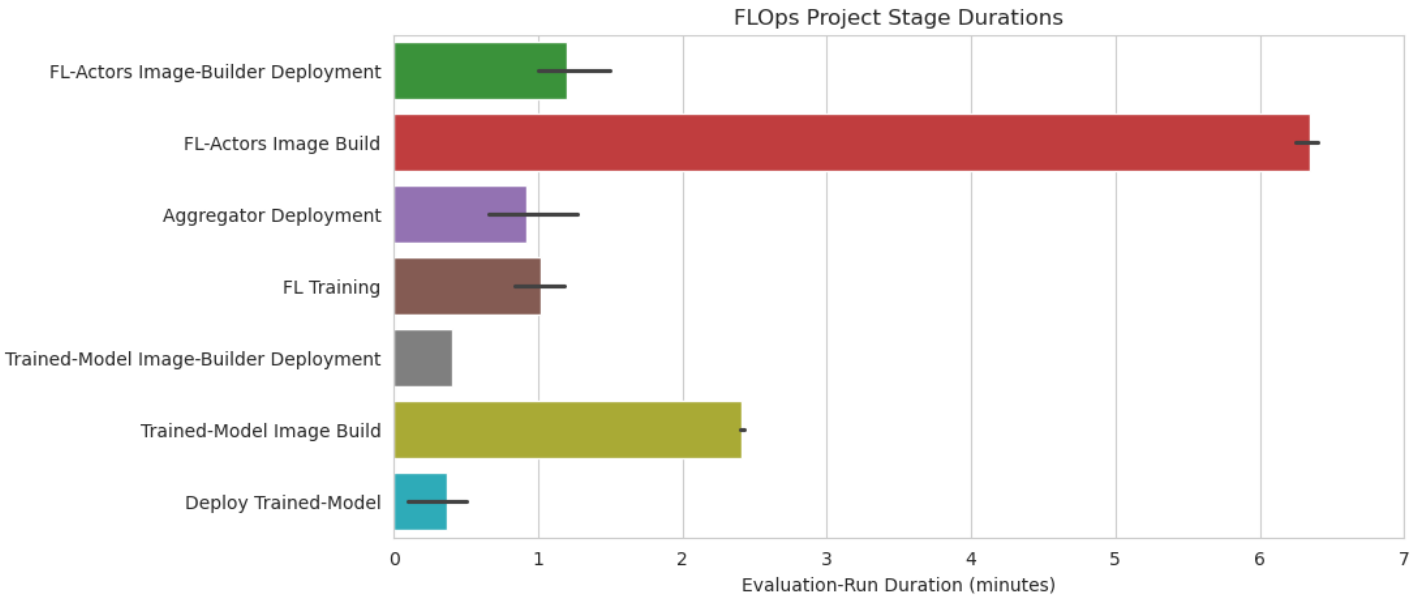

Figure 5.7: Experiment 1: Stage Durations

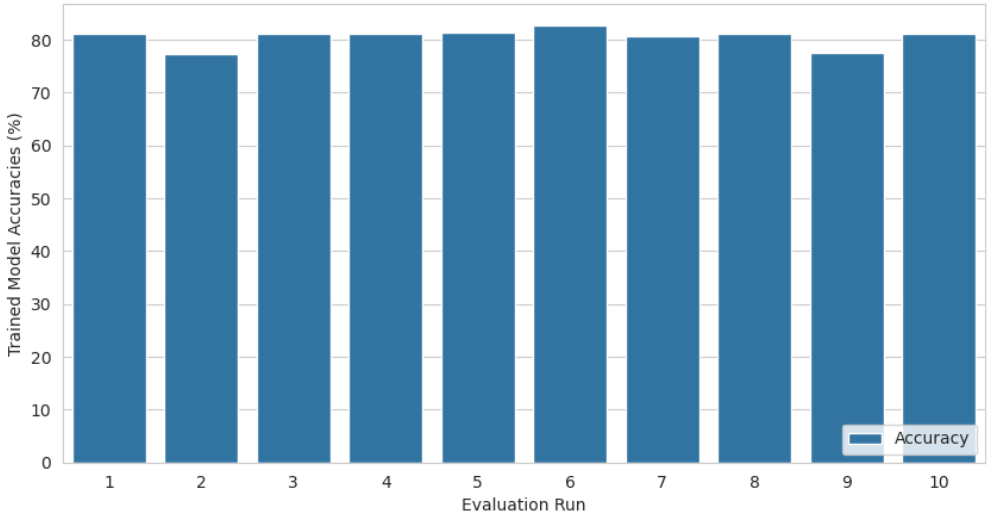

Figure 5.8: Experiment 1: Trained Model Accuracies

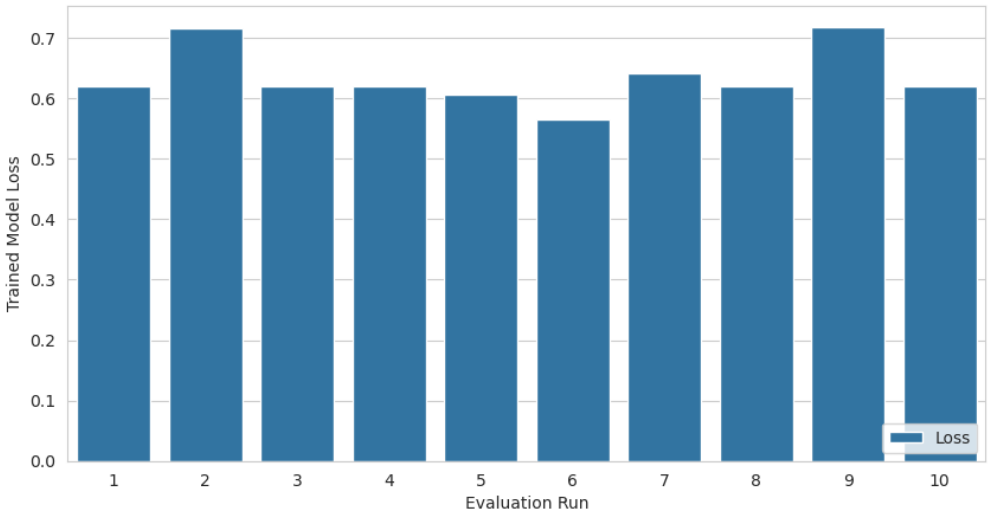

Figure 5.9: Experiment 1: Trained Model Loss

5.3.2 Image Builder

This subsection highlights how different variable configurations impact the total project, especially its image build times, which make up a significant part of it.

Figure 5.10 depicts the CPU and memory utilization of experiment (3). Compared to (1), a project now only takes approximately eight minutes, and the FL-actors' build times shrunk from six to two and a half minutes. The use of development base images leads to even greater acceleration when the underlying dependency structure of the ML repository is larger. The base images do not apply to the trained model build because each trained model is unique. The stage durations graph 5.11 of experiment (3) highlights this acceleration. All other metrics, such as net-IO, disk space, memory, or results, stay the same as in (1) because the only change is the reuse of base images. Therefore, when people develop FLOps further, they should utilize this base-image functionality to speed up their progress.

Figure 5.12 shows the CPU and memory utilization of experiment (4). It is immediately clear how much emulation slows down multi-platform projects compared to the base case. A simple multi-platform project takes around 45 minutes. The FL-actor image build stage takes 27 minutes, which is a 4.5-fold increase, and the multi-platform trained-model image build takes 13 minutes, which is a fivefold increase. This stark difference is especially visible in Figure 5.13, which shows the stage durations of experiment (4). Other metrics are also lightly affected by this change. More disk space is needed. Experiment (4) uses up 17.5 GB instead of 14 GB due to the extra image layers.

This increase also influences the net-IO where (4) requires sending out 12 GB and receives 10 GB instead of (1)'s 9 GB and 5.5 GB. In addition, the net-IO accumulated more traffic over the longer project run-time. In conclusion, these findings prove that FLOps supports working multi-platform images. However, this approach significantly increases build times when run on single machines instead of dedicated build machines with matching native architectures.

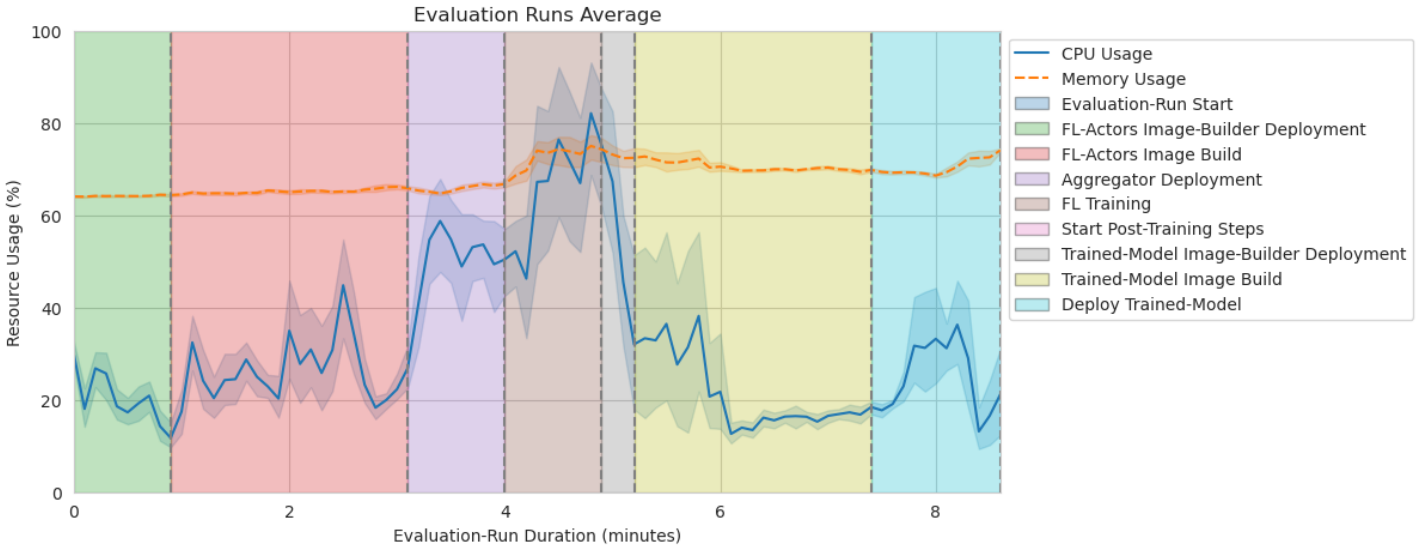

Figure 5.10: Experiment 3: CPU & Memory Utilization

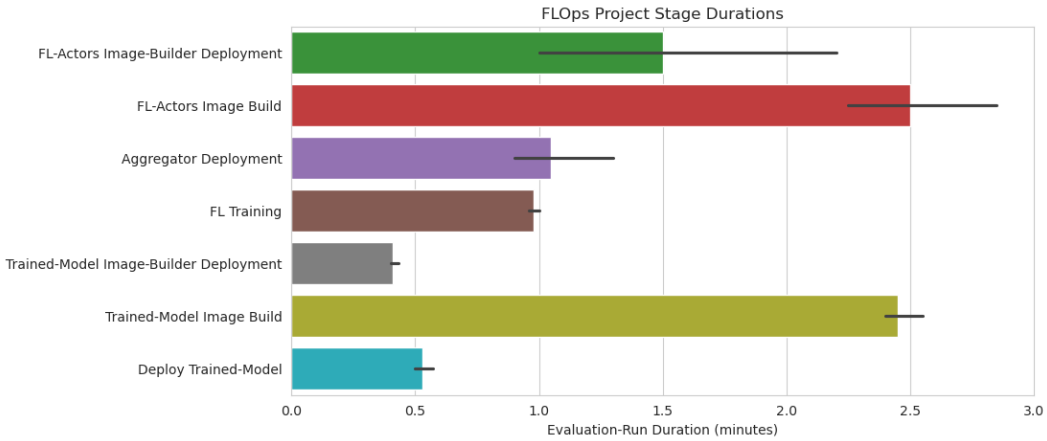

Figure 5.11: Experiment 3: Stage Durations

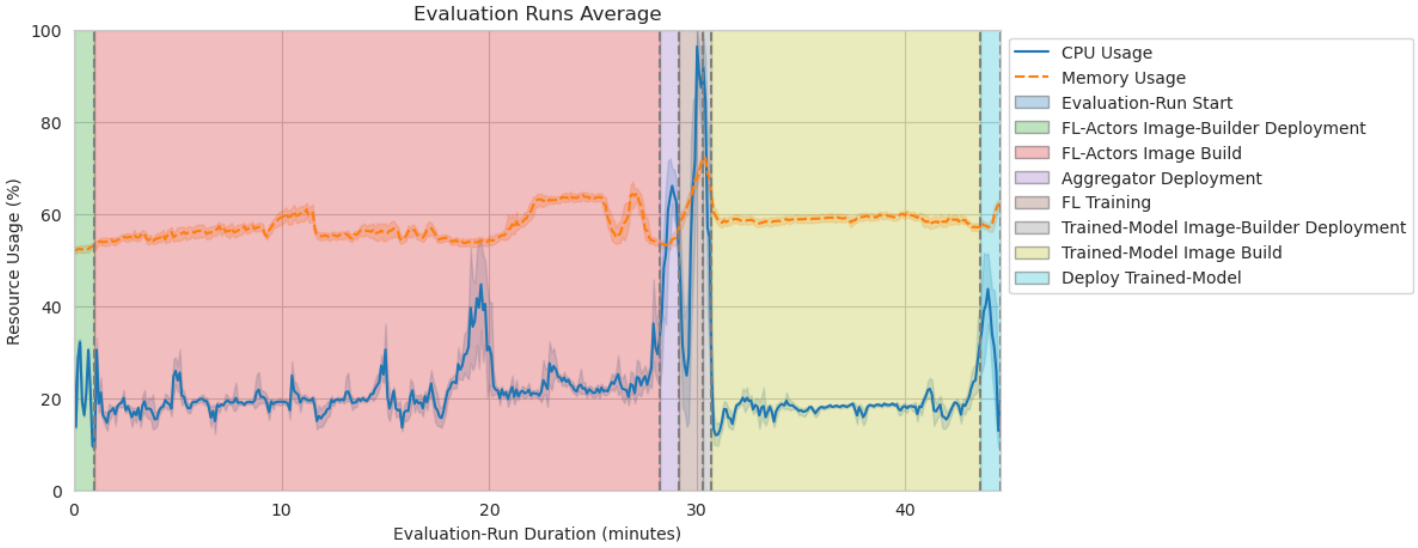

Figure 5.12: Experiment 4: CPU & Memory Utilization

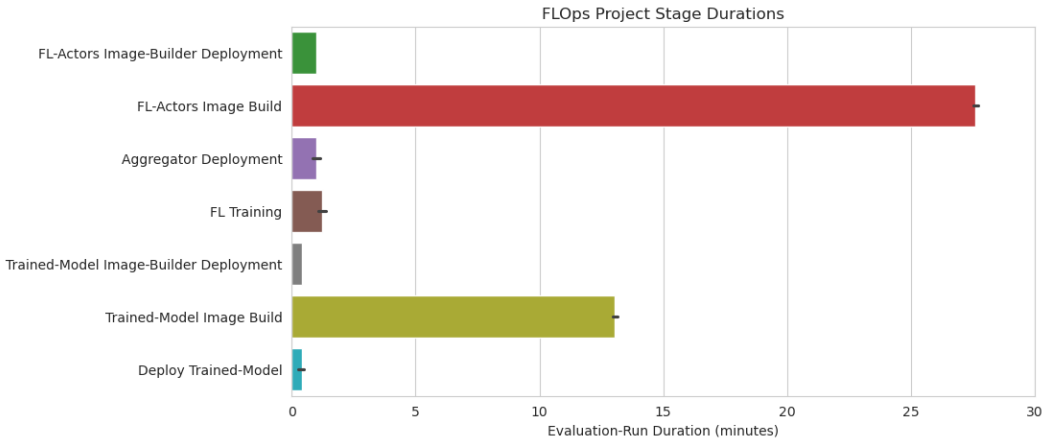

Figure 5.13: Experiment 4: Stage Durations

5.3.3 Fundamentally Different Projects

Longer Training Rounds with more Learners

Figure 5.14 shows the CPU and memory utilization of experiment (2). Because this project configuration uses more learners and training rounds, it is logical that the FL

training stage will last longer. Note that this larger example is still relatively small compared to ML/FL training periods that take multiple hours or days to complete. As a result, the CPU utilization during training is less spread than for (1) and concentrates in the high 90-100% range. Memory shows a similar shift from the low 60s to the low 70s. The disk space and net-IO stay similar to (1). The resulting accuracies are now all in the mid-80s instead of below 80%, as in (1). Extended training periods lead to better results in this case.

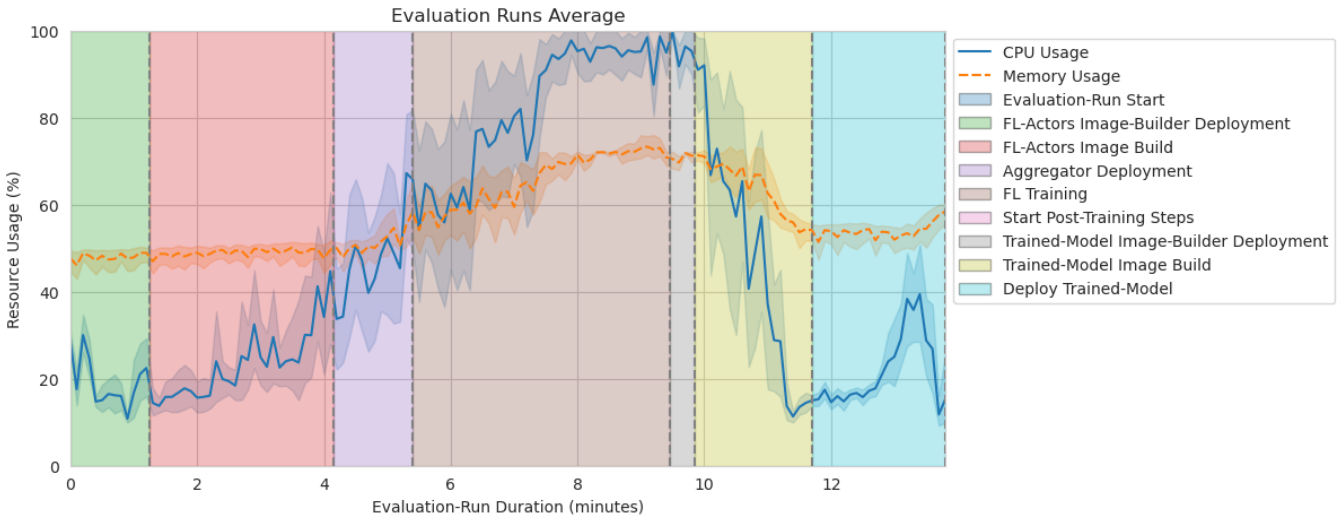

Figure 5.14: Experiment 2: CPU & Memory

Different ML Repository, Framework, and Dataset

Figure 5.15 shows the CPU and memory utilization of experiment (5). The first noticeable difference to (1) is the almost threefold project duration, mainly due to build times. Pytorch is a more heavy-weight ML library than Scikit-learn. Thus, configuring its larger dependencies takes longer. (5)'s CPU behavior is similar to (1). The FL training in (5) requires less memory (mid 50s) than in (1) (low 60s).

Figure 5.16 shows the remarkable influence of the chosen ML framework and its dependencies for FLOps. Compared to the relatively lightweight Sklearn base-case example with a total used disk space increase of 14 GB, this simple Pytorch example takes up approximately 35 GB in the end. During build times when dependencies are pulled, the peak extra disk space utilization reaches 60 GB. These heavy dependencies are also visible in the network IO in Figure 5.17. The garbage collection is strongly present and visible in this case. The tiny number of learners and training rounds lead

to a small accuracy of 40% of the final models.

As a result, these experiments prove that FLOps can handle different FL training configurations using various ML frameworks, repositories, and datasets. They also unveil that using popular classic ML libraries might be unrealistic for resource-constrained edge devices. Thus, libraries that are dedicated to restricted devices should be analyzed and tried out as future work.

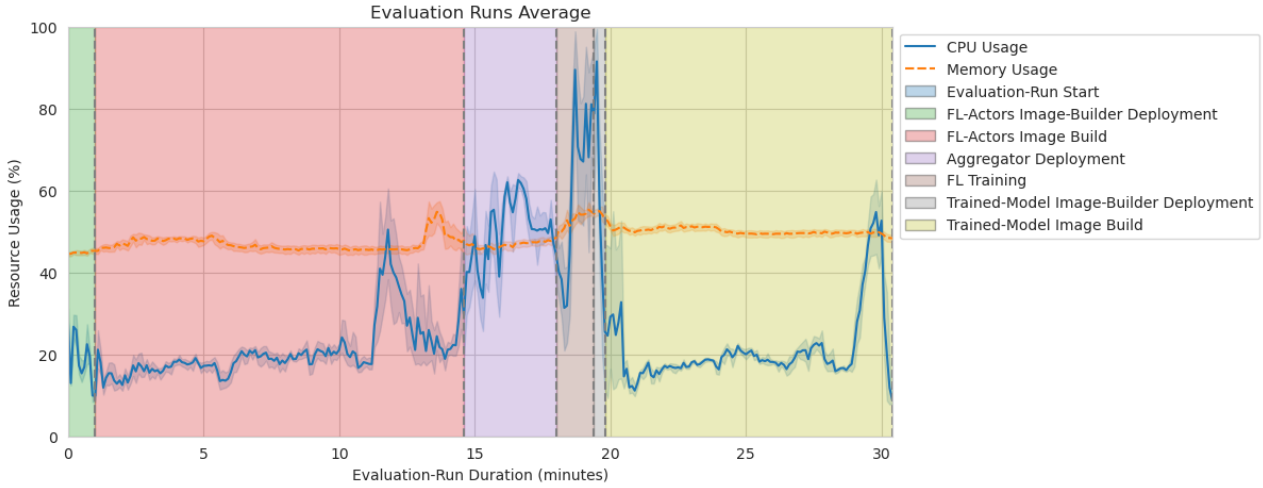

Figure 5.15: Experiment 5: CPU & Memory

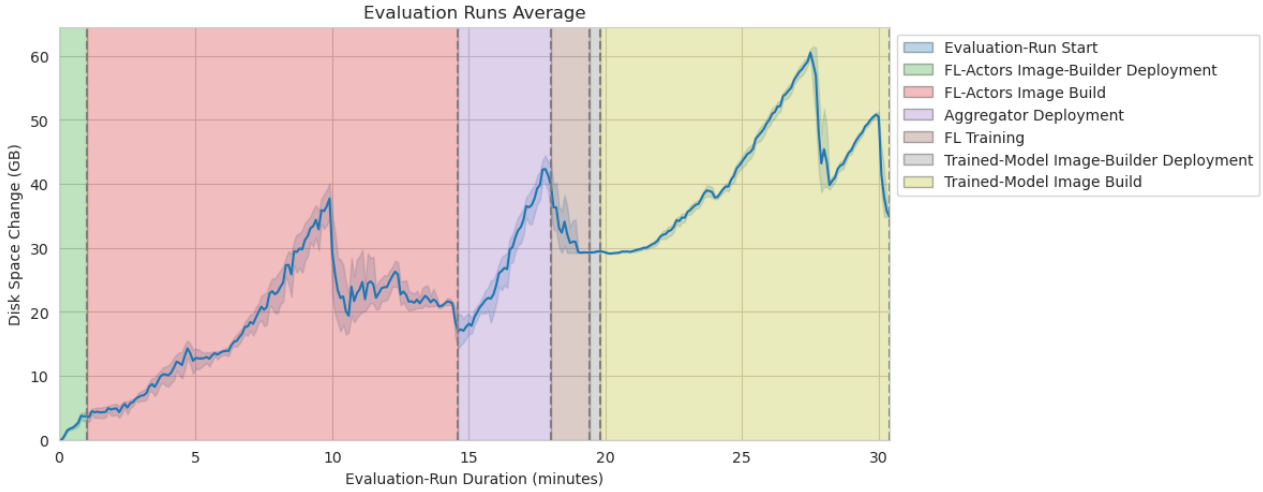

Figure 5.16: Experiment 5: Disk Space

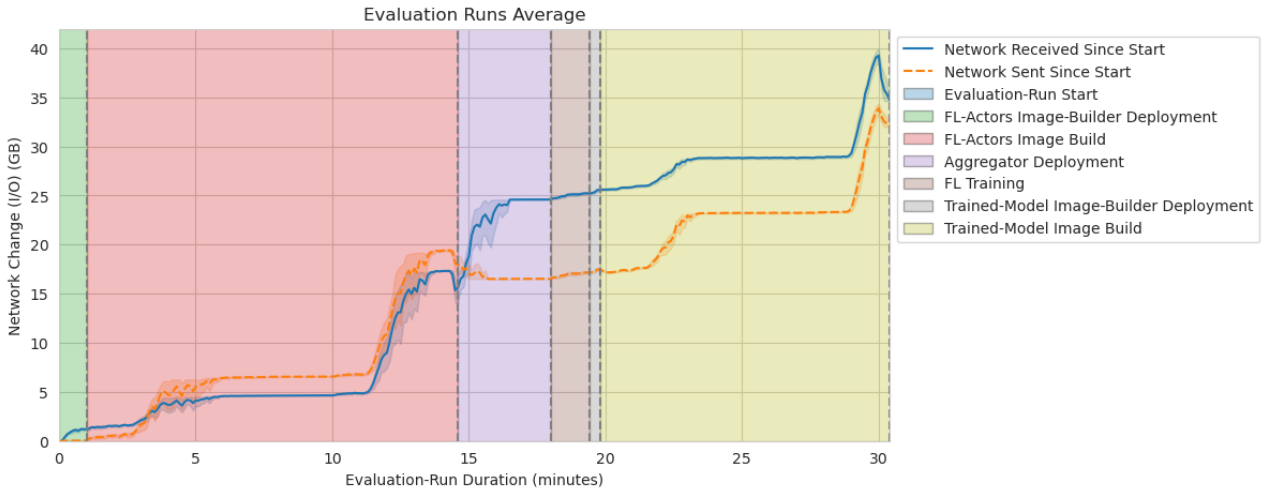

Figure 5.17: Experiment 5: Network IO

5.3.4 Multi-cluster & HFL

Classic FL on the Multi-Cluster Setup

Before analyzing how real HFL performs on multiple clusters, we need to verify that the second setup is capable of classic FL and compare it to the monolith setup. Observing the aggregated sum of metrics from all three devices leads to flat and less insightful plots. For this reason, the following graphs will depict the metrics by device. Figure 5.18 shows the CPU utilization of the cluster setup during experiment (7). It depicts how the control plane root device only manages components but does not perform any heavy operations as intended. Its CPU utilization is constantly at a very low level. The orchestrator selected and deployed the image builder service on cluster B. That is why it is the only busy device during the initial image build phase. Notably, this scheduling decision seems to be deterministic. The orchestrator selected Cluster B for every single run. FLOps does not tell the orchestrator to use the same cluster for the image builders. Once FL actor deployment and training start, CPU utilization gets distributed among both cluster nodes. Figure 5.19 shows the memory utilization of the devices. The memory stays stable when no workloads are performed on a device, which is the case for the root at all times and for cluster-A while cluster-B is building the images. The reason why the root device has a higher memory usage then the two clusters is because the root hosts the control plane. This includes Oakestra's root orchestrator and all FLOps management components.

Figure 5.20 shows the increase in disk space for the devices. It shows how cluster-B's

disk space is increasing during the build process due to the dependencies and layers that are pulled and build. There is a drop after the build process finishes, and the builder service is undeployed. In the middle of the build process, cluster B pushes the built base image to the root that hosts the image registry. Cluster-B pushes the FL-actor images without a significant increase because the common base layers are reused. The disk space of cluster-A only starts increasing when the FL actors are deployed on it. Figures 5.21 and 5.22 show the received and sent network changes on the devices, matching the disk space changes.

The FLOps stages of the experiment (7) do not show any remarkable outliers and resemble the base case. The only difference is that the project and its stages take longer due to the weaker hardware. The final training results are equivalent to (1).

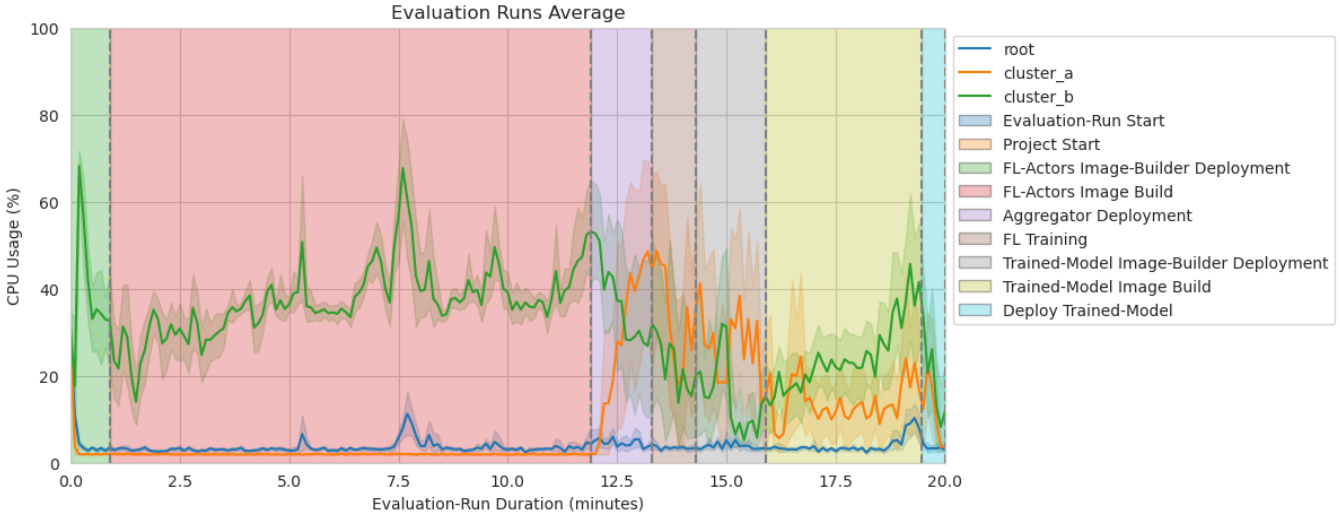

Figure 5.18: Experiment 7: CPU Utilization

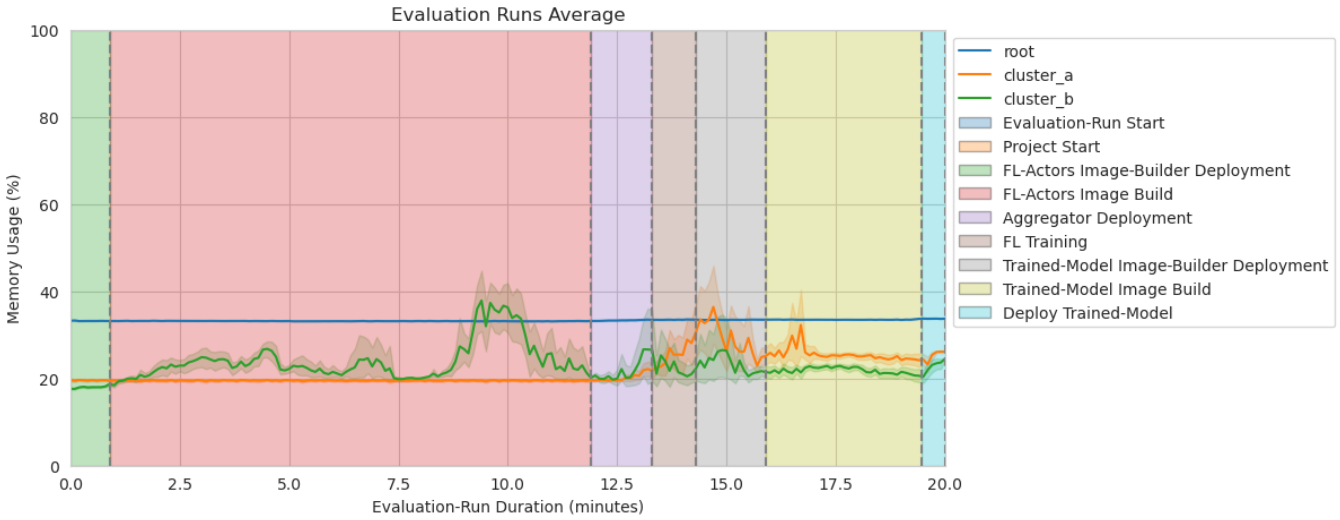

Figure 5.19: Experiment 7: Memory Utilization

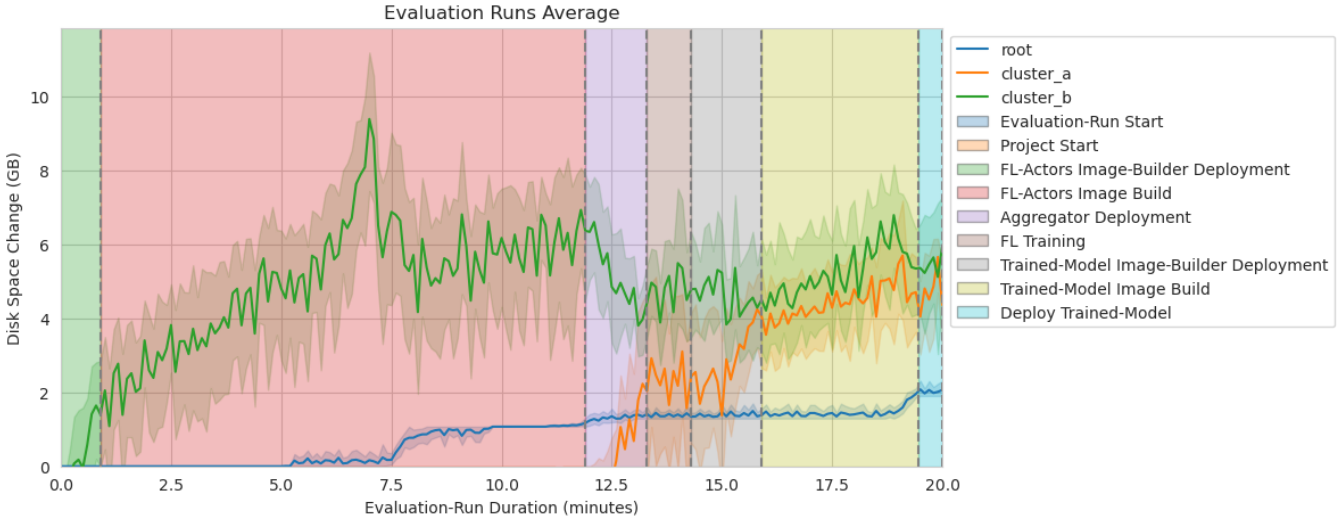

Figure 5.20: Experiment 7: Disk Space

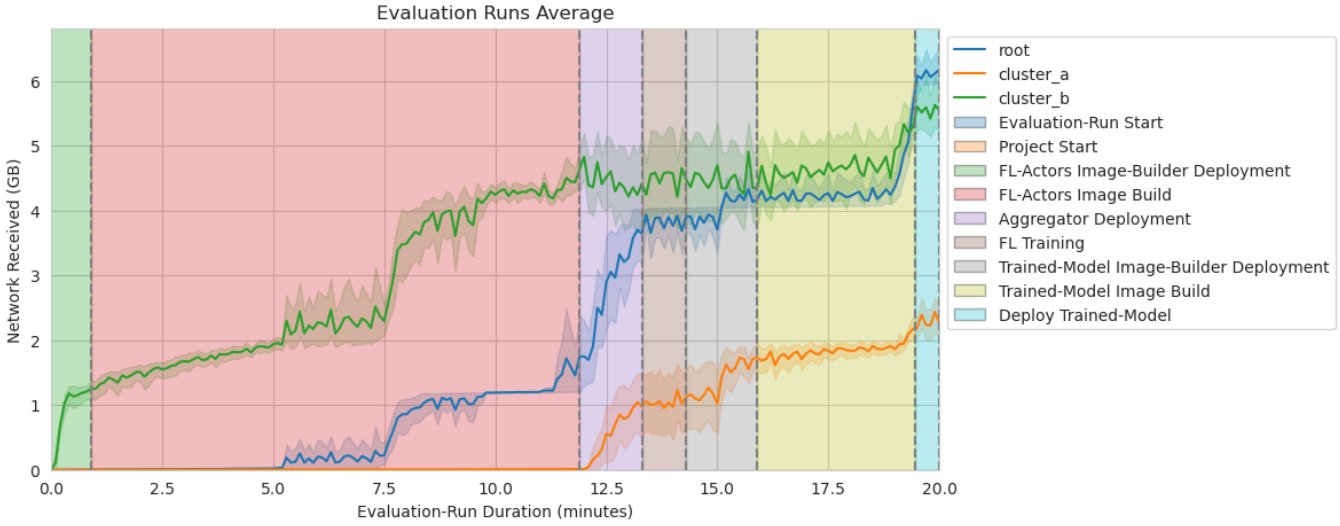

Figure 5.21: Experiment 7: Received Network

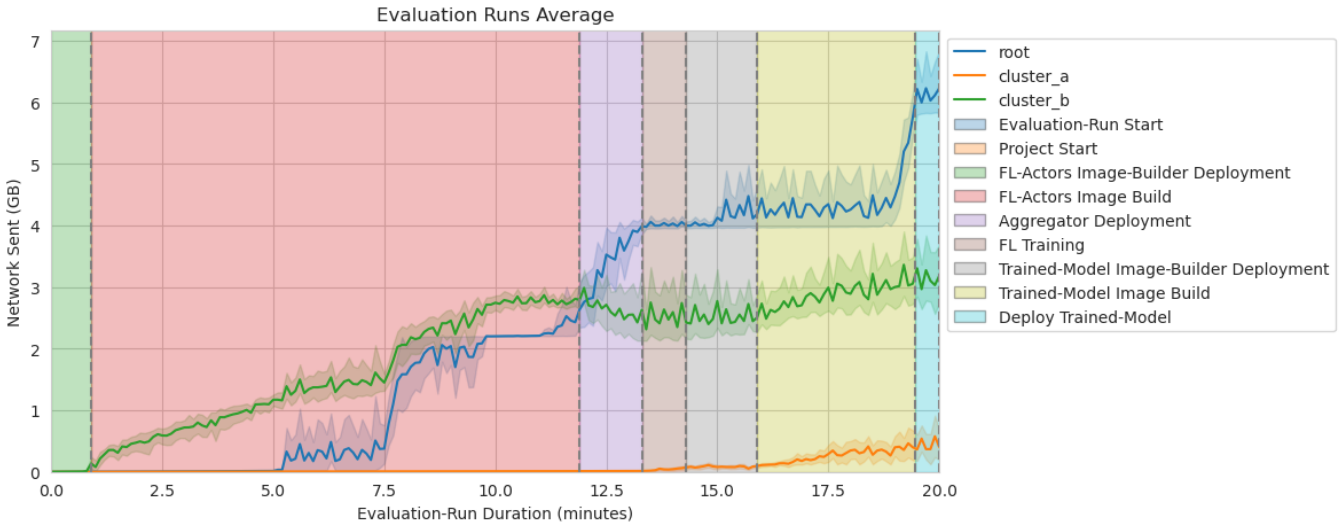

Figure 5.22: Experiment 7: Send Network

Minimal HFL base-case on a monolith Cluster

Figure 5.23 shows the CPU and memory utilization of experiment (6). Its purpose is to be a minimal HFL base-case that only uses a single cluster. In additon, this experiment

proofs that FLOps' custom clustered HFL solution works. Compared to (1) the project duration is a bit longer due to the longer training stage. Every other metric stay similar to (1). This includes the stage durations, disk space, net IO, and training results. CPU and memory utilization are slightly increased in the deployment and training stages because of more FL actors.

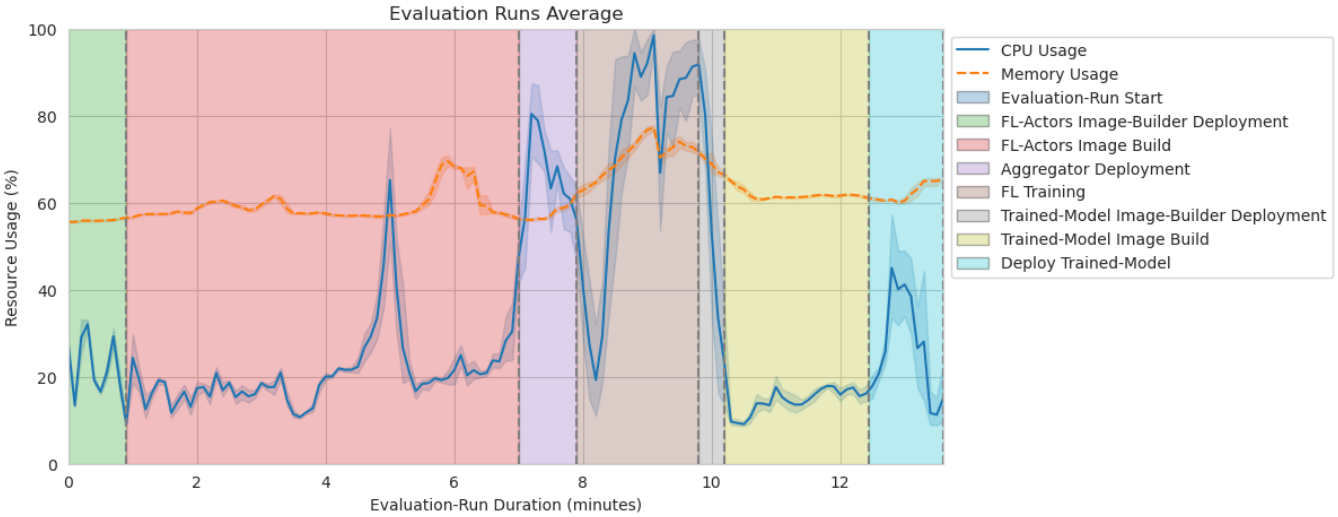

Figure 5.23: Experiment 6: Monolith HFL CPU & Memory

HFL on the Multi-Clustered Setup

Figure 5.24 shows the CPU utilization of the multi-cluster setup devices during HFL. The image build seems to be no longer handled by a single cluster but is more distributed among both. In (7), learners were randomly deployed on clustered. In (8), the figure clearly shows that the CPU is utilized more and distributed more equally among the clusters. This is the case because each cluster gets its own cluster aggregator and set of learners. This more homogeneous utilization also applies to memory, as Figure 5.25 shows. All other metrics show a similar change from (7) to (8) as from (1) to (6). This includes disk space, net IO, stage durations, and training results. These findings prove that FLOps can perform clustered HFL on distributed multi-cluster devices.

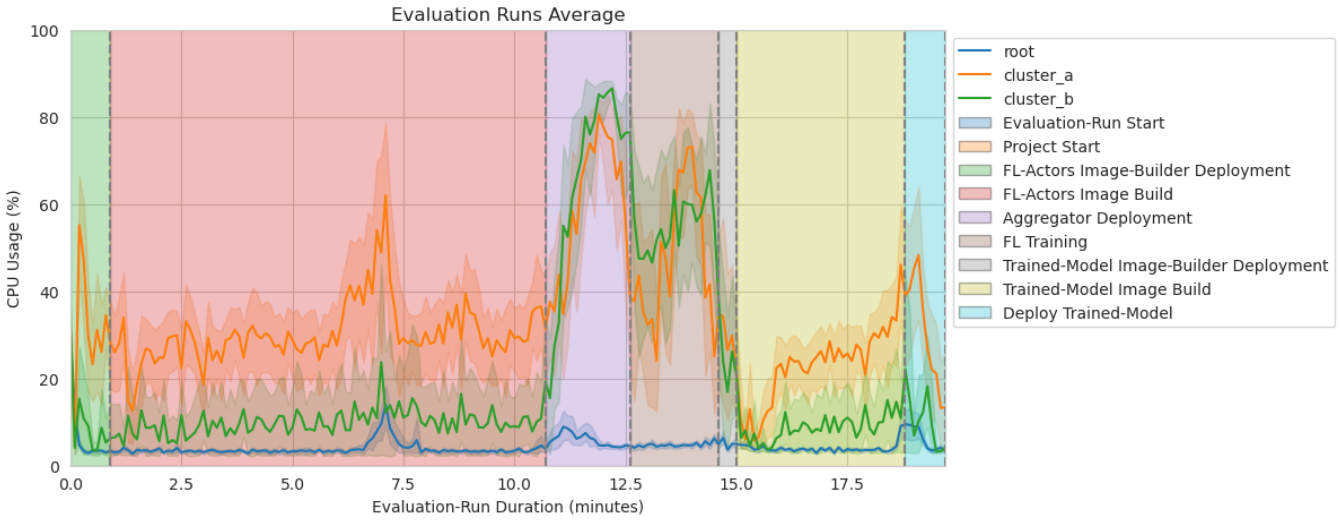

Figure 5.24: Experiment 8: Multi-Cluster HFL CPU Utilization

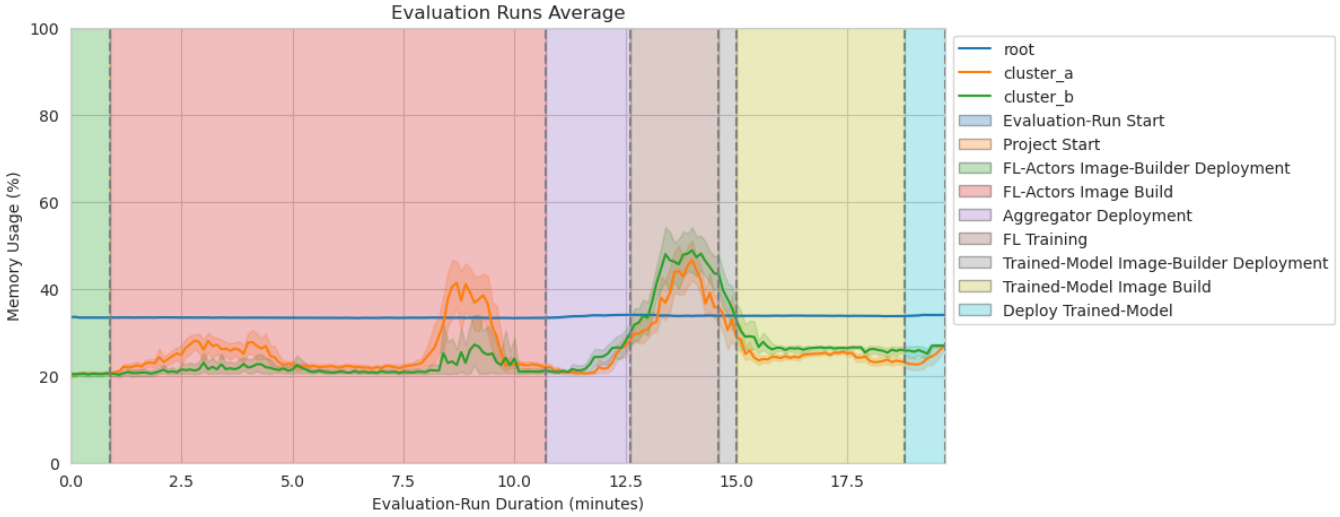

Figure 5.25: Experiment 8: Multi-Cluster HFL Memory Utilization

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] *Alpine Multi-Platform Image*. Accessed: 2024-08-31. URL: https://hub.docker.com/layers/library/alpine/latest/images/sha256-eddacbc7e24bf8799a4ed3cdcfa50d4b88a32context=explore.

[4] *Anaconda Documentation*. Accessed: 2024-08-29. URL: https://docs.anaconda.com/.

[5] *Apache Arrow Flight Documentation*. Accessed: 2024-08-29. URL: https://arrow.apache.org/docs/format/Flight.html.

[6] *Apache Arrow Flight Introduction*. Accessed: 2024-08-29. URL: https://www.dremio.com/blog/an-introduction-to-apache-arrow-flight-sql/.

[7] *Apache Arrow Repository*. Accessed: 2024-08-29. URL: https://github.com/apache/arrow.

[8] *Apache Parquet Documentation*. Accessed: 2024-08-29. URL: https://parquet.apache.org/docs/overview/.

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

[10] 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.

[11] 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].

[12] 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.

[13] 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.

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

[15] *Buildah and Podman Relationship*. Accessed: 2024-08-30. URL: https://podman.io/blogs/2018/10/31/podman-buildah-relationship.

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

[17] 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].

[18] 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.

[19] 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.

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

[21] *Click Repository*. Accessed: 2024-09-03. URL: https://github.com/pallets/click.

[22] *CNCF Distribution Registry Documentation*. Accessed: 2024-08-30. URL: https://distribution.github.io/distribution/.

[23] *Conda Documentation*. Accessed: 2024-08-29. URL: https://conda.io/projects/conda/en/latest/index.html.

[24] *Conda documentation about libmamba solver*. Accessed: 2024-08-29. URL: https://conda.github.io/conda-libmamba-solver/user-guide/.

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

[26] J. L. Dem. *The columnar roadmap: Apache Parquet and Apache Arrow*. Accessed: 2024-08-31. 2018. URL: https://www.youtube.com/watch?v=dPb2ZXnt2_U.

[27] *Docker Buildx Documentation*. Accessed: 2024-08-31. URL: https://docs.docker.com/reference/cli/docker/buildx/build/.

[28] *Docker Manifest Documentation*. Accessed: 2024-08-31. URL: https://docs.docker.com/reference/cli/docker/manifest/.

[29] *Docker Multi-Platfrom Image Builds Documentation*. Accessed: 2024-08-31. URL: https://docs.docker.com/build/building/multi-platform/.

[30] *EMNIST Dataset*. Accessed: 2024-08-29. URL: https://www.nist.gov/itl/products-and-services/emnist-dataset.

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

[32] *flask-openapi3 Framework*. Accessed: 2024-08-29. URL: https://github.com/luolingchun/flask-openapi3.

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

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

[35] *Flower Container Images*. Accessed: 2024-08-30. URL: https://hub.docker.com/u/flwr.

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

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

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

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

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

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

[42] *Flower Next API Announcement*. Accessed: 2024-08-30. URL: https://flower.ai/blog/2024-04-03-announcing-flower-1.8-release.

[43] 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/.

[44] 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.

[45] 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.

[46] *HDF5-FEMNIST Project*. Accessed: 2024-08-29. URL: https://github.com/Xiao-Chenguang/HDF5-FEMNIST.

[47] 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.

[48] *Hugging Face Datasets and Arrow Documentation*. Accessed: 2024-08-29. URL: https://huggingface.co/docs/datasets/about_arrow.

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

[50] *Image Manifest Versions and Schemas Documentation*. Accessed: 2024-08-31. URL: https://distribution.github.io/distribution/spec/manifest-v2-2/.

[51] 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].

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] A. Lamb. *Building InfluxDB 3.0 with Apache Arrow, DataFusion, Flight and Parquet*. Accessed: 2024-08-31. 2024. URL: https://www.datacouncil.ai/talks24/building-influxdb-30-with-apache-arrow-datafusion-flight-and-parquet?hsLang=en.

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

[58] J. Lenoy. *Apache Arrow Flight SQL: High Performance, Simplicity, and Interoperability for Data Transfers*. Accessed: 2024-08-31. 2022. URL: https://www.youtube.com/watch?v=OLsXlKb_XRQ.

[59] 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.

[60] *Linux Kernel FUSE Documentation*. Accessed: 2024-08-30. URL: https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[61] 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].

[62] 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.

[63] *Mamba Documentation*. Accessed: 2024-08-29. URL: https://mamba.readthedocs.io/en/latest/.

[64] 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.

[65] *Micromamba Documentation*. Accessed: 2024-08-29. URL: https://mamba.readthedocs.io/en/latest/user_guide/micromamba.html.

[66] *Miniconda Documentation*. Accessed: 2024-08-29. URL: https://docs.anaconda.com/miniconda/.

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

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

[69] *MLflow Documentation about Docker Build Commands*. Accessed: 2024-08-30. URL: https://mlflow.org/docs/2.12.2/cli.html?highlight=docker%20image#mlflow-models-build-docker.

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

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

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

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

[74] 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].

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

[76] *Oakestra Dashboard*. Accessed: 2024-09-03. URL: https://github.com/oakestra/dashboard.

[77] *OCI Image Index Specification*. Accessed: 2024-08-31. URL: https://github.com/opencontainers/image-spec/blob/main/image-index.md.

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

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

[80] 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.

[81] 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.

[82] *Podman Documentation*. Accessed: 2024-08-30. URL: https://docs.podman.io/en/latest/.

[83] *Poetry Documentation*. Accessed: 2024-08-29. URL: https://python-poetry.org/docs/.

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

[85] *Python argcomplete Documentation*. Accessed: 2024-09-03. URL: https://kislyuk.github.io/argcomplete/.

[86] *Python argparse Documentation*. Accessed: 2024-09-03. URL: https://docs.python.org/3/library/argparse.html.

[87] *QEMU Documentation*. Accessed: 2024-08-31. URL: https://www.qemu.org/documentation/.

[88] 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.

[89] *Red Hat explains Buildah*. Accessed: 2024-08-30. URL: https://www.redhat.com/en/topics/containers/what-is-buildah.

[90] *Red Hat explains Podman*. Accessed: 2024-08-30. URL: https://www.redhat.com/en/topics/containers/what-is-podman.

[91] *Rich Repository*. Accessed: 2024-09-03. URL: https://github.com/Textualize/richr.

[92] 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.

[93] 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.

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

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

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

[97] *SetupTools User Guide*. Accessed: 2024-08-29. URL: https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html.

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

[99] *Typer Repository*. Accessed: 2024-09-03. URL: https://github.com/fastapi/typer.

[100] *uv Python package and project manager*. Accessed: 2024-08-29. URL: https://github.com/astral-sh/uv.

[101] *Waitress WSGI Server*. Accessed: 2024-08-29. URL: https://github.com/Pylons/waitress.

[102] 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.

[103] 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.

[104] 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.

[105] 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].

[106] 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.

[107] 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.

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

Appendices