Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Implementing A Federated Machine Learning Scenario Using Vantage6 and Brane

Author: Leyu Liu (2630429)

1st supervisor:	Adam Belloum
daily supervisor:	Onno Valkering
2nd reader:	Reginald Cushing

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

September 27, 2021

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Federated Learning (FL) is a distributed machine learning that enables model training on multiple devices with the coordination of a server. Recently, many FL frameworks have been developed to support FL systems. For instance, Vantage6 is one of the open-source FL frameworks that aims to apply FL for preserving medical data privacy. Brane is a framework with a user-friendly programming model that is developed at MNS group at the UvA to support the implementation of distributed applications. To investigate the possibility of using Vantage6 services in Brane for FL, we introduce the light integration of Brane and Vantage6. In this thesis project, we (1) Propose the integration[1] of Brane and Vantage6 via API function calls. The integration is performed under two scenarios based on user preference; (2) Implement a federate machine learning (ML) algorithm using two use cases for testing the applicability of the integration. Both use cases are implemented as a Federated Averaging (FedAvg) algorithm, while one is for image classification and the other is for sentiment analysis. (3) Evaluate the performance of integration on both the local machine and virtual machines. The evaluation metrics we use include running time and CPU and RAM usage.

The experimental results show that the integration successfully enables a federated machine learning workflow by implementing Vantage6 as functions in Brane. Though accessing Vantage6 services in the integration introduces some overhead, the overall performance of running a FL task in the integration is similar to standalone Vantage6.

[1] Code available on GitHub: https://github.com/621Alice/Fed-Brane-V6

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Problem Description

With the rising popularity of the big data field, more concerns are raised in the society regarding the problem of data privacy. Various companies and organizations like hospitals cannot legally reveal or expose their customers private information due to ever tightening data protection regulations (4). For artificial intelligence to function and develop, however, such private data is crucial to the researchers and engineers as to allow training ML models that satisfy the performance requirements. One solution proven effective in solving this issue has been implementation of federated learning systems(FL), which enables local training using decentralized data, minimizing risk of leakage. Typically, an FL system requires a central server and multiple worker nodes. Those worker nodes contain and process local information which is never sent to the central unit. The server distributes a global model to the nodes and then receives updates once the local computation has taken place on each of them. Through aggregation of local updates the global model then improves itself until it converges, resulting in the final solution (5).

In order to ensure development of federated learning, various open source FL frameworks have been developed. Vantage6 (2) is one of the platforms that managed to successfully implement an infrastructure supporting FL systems that allows the use of any programming language as well as operating system. Brane (6) is another framework which allows for the development of large-scale projects for domain experts. Due to its programmable and re-configurable features, Brane can be developed to support generic distributed projects, such as FL systems.

1.2 Goals

Considering the capabilities of Brane and Vantage6 frameworks and their implementation in developing federated learning and other distributed research projects, this thesis aims to examine the possibility of integration of the two. The proposed integration can be assessed by implementing a federated machine learning workflow that uses both Vantage6 and Brane services.

1.3 Research Questions

The integration of the Brane and Vantage6 raises several issues which we try to answer through the following research questions:

- RQ1: How to combine the programming models of the two systems to improve user experiences during the process of developing a ML pipeline, namely implementation, orchestration and deployment?
  - What choices do we have for the integration of the two frameworks?
  - Can we integrate Vantage6 and Brane successfully through API calls (light integration)?
  - Can we perform low-level integration between Vantage6 and Brane (deep integration)?
- RQ2: How can a particular integration between Vantage6 and Brane be made generic, and reusable for ML use cases(e.g. a particular problem that can be solved by ML)?
  - What ML use cases can we use to test the performance of a specific type of integration?
  - Can we apply different types of federated learning workflows to test a specific type of integration?

This thesis is structured as follows. After Chapter 1, some general concepts, such as FL, Docker, Vantage6, Brane, REST API and OpenAPI Specification, are introduced in Chapter 2. Chapter 3 first introduces the workflow of a FL pipeline in Vantage6. Then, we describe the design of the integration with their workflow in a FL pipeline. Two different scenarios are considered for the integration based on whether the Vantage6 infrastructure is pre-installed. In Chapter 4, we describe the implementation details of both integration scenarios. We also describe the steps to build a FL workflow in Vantage6 and how to create new FL algorithms using Vantage6 client functions. In addition, we evaluate the integration and illustrate the results of the experiments with brief discussion and

comparison in Chapter 5. Chapter 6 includes further discussion regarding the integration and possible future research directions, such as the possibility to implement other FL workflows in Vantage6 and possible further integration of Brane and Vantage6. Lastly, we summarize this thesis and provide answers to the research questions in Chapter 7.

2

Background

Before integrating Brane and Vantage6 and evaluating the integration in a FL pipeline, it is essential to first gain some understandings of the workflow of FL, the architecture of Vantage6 and Brane and some related concepts that are needed for implementation. Thus, in this Chapter, we introduce the background information that is related to my thesis, including:

- Federated Learning
- Related tools and concepts, including
  - Vantage6
  - Brane
  - Rest API
  - OpenAPI Specification
  - Docker

2.1 Federated Learning

In this section, we introduce the workflow of FL as well as its open problems.

To protect data privacy, federated learning is developed in recent years to enable decentralized model training in machine learning using multiple client devices and a central server (1). Figure 2.1 illustrates the workflow of FL. Each client device (e.g. a mobile phone) has their own local data which is not shared with other client devices or the server. During FL, each client performs local training and send model updates to server for updating the global model. The communication between the server and client device is encrypted such that the sensitive information is not leaked in the FL process (4). To ensure data privacy more effectively, previous researchers tried various federated optimization techniques to solve the existing challenges in FL. In the following paragraphs we introduce these

challenges separately, including data heterogeneity, data partitioning, privacy concerns, communication overhead and slow model convergence (1, 4, 7, 8).

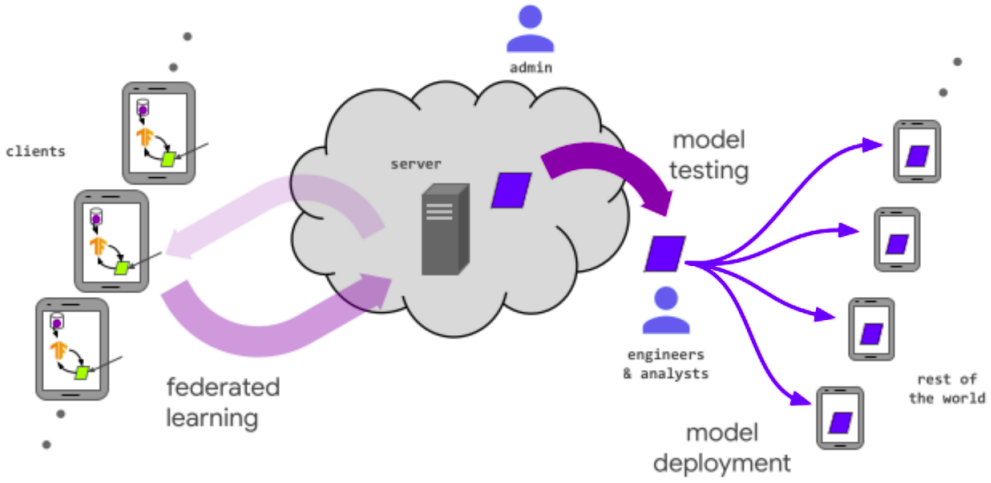

Figure 2.1: The workflow of FL (1)

Data heterogeneity Heterogeneous data refers to not independent and identically distributed (non-IID) data. With statistical difference in data, the model training process could be problematic and might cause issues in FL. For instance, because of non-IID data, the local optimum in each client is distant from the true global optimum. During gradient descent, local gradient will move to the direction of the local optimum and can lead to slow convergence of the global model. Karimireddy et al. (9) referred to such problem as client drift. They tackled non-IID data using optimization techniques in the FL process by adding a correction term to move the local gradient to the direction of true optimum.

Data partitioning FL algorithms can be classified into horizontal FL (HFL), vertical FL (VFL) and Federated transfer learning (FTL) based on data partitioning (4). Different workflows of FL have been proposed to fit different data partitioning. HFL refers to FL algorithms using horizontally partitioned dataset with overlapped features but different users. VFL uses vertical data partitioning where users can be overlapped in different local datasets but the features in each dataset is different. FTL is commonly used in situation where there is insufficient training samples (4, 10).

Privacy concerns As for privacy issues in FL, there exits many adversary attacks against ML models and their training data. To enhance data security, current FL al-

gorithms often use differential privacy mechanisms (DP) or cryptographic methods. In DP, Laplace mechanism and exponential mechanism are often used to add noise to the model gradient such that the information of any individual user will not be revealed (4). Cryptographic methods are often encryption methods that are used to encrypt the exchanged messages (e.g. model parameters) between the server and clients. It is often combined with public-private key pair and authentication to ensure privacy (11).

Communication overhead and slow model convergence Many FL optimization techniques have been proposed to reduce the communication overhead and speed up the model convergence rate. Most of the techniques are developed by improving the Federated Averaging learning (FedAvg), which is a typical example of FL algorithm that was proposed by Mcmahan et al. (5). We also implement FedAvg in this project to evaluate the performance of our proposed integration. In FedAvg, the clients receive the model from the central server and perform local training. Then, the updated model parameters will be sent back to server for averaging. The averaged parameters will be sent to clients again for another round of local training. The process of server-node communication and parameter averaging repeats until the global model is converged. Mcmahan et al. pointed out that the communication overhead between server and clients was the biggest bottleneck for FL. Therefore, they added additional computation (e.g. increase the number of training epochs) in each communication round to reduce the total number of communication rounds. However, it caused large communication cost in each round and the convergence rate in FedAvg appeared to be slow. Thus, many other researchers proposed various approaches to optimize FedAvg in terms of communication overhead and convergence rate. For instance, the use of Adam optimization and efficient compression techniques proposed by Mills et al. (12) to reduce the communication cost in each round while speeding up model convergence.

FL is getting more popular in the IT industry and the existing problems in FL are drawing attention to more researchers. As a result, the development of FL optimization techniques is getting promoted. The FL performance will be further improved and future FL systems will be more practical to be applied in real-life scenarios.

2.2 Related Tools and Concepts

In this section, we introduce the related tools and concepts we encountered during the implementation of the proposed integration, including

- Architecture of Vantage6
- Architecture of Brane
- Definition and principles of REST API
- Definition and application of OpenAPI Specification
- Architecture and application of Docker

Vantage6 Vantage6 (2) provides users a federated learning framework. Figure 2.2 illustrates the system architecture of Vantage6. As we can see, Vantage6 is composed of three components including a researcher, a central server and a set of worker nodes (clients). Each worker node has their own local data stored in the local database. A user can install either a server or a node on the local machine and configure it with a configuration file. A researcher can submit a task to the server with an algorithm image and input parameters. The algorithm image can be first implemented using Vantage6 tools and built into a docker image using a Dockerfile. After a task is successfully submitted to the server, the server will send the task information to a computing node. After a node is started, it will automatically detect the server, get the task information and execute the algorithm using local data. The intermediate results will be sent back to server for aggregation and the iterative process of FL repeats to update the global model. The final result will be sent back to the researcher when the computation is complete. Vantage6 is by design flexible and can support multiple languages and configurations. It also suits different data partitioning flexibly, including vertically partitioned dataset and horizontally partitioned dataset.

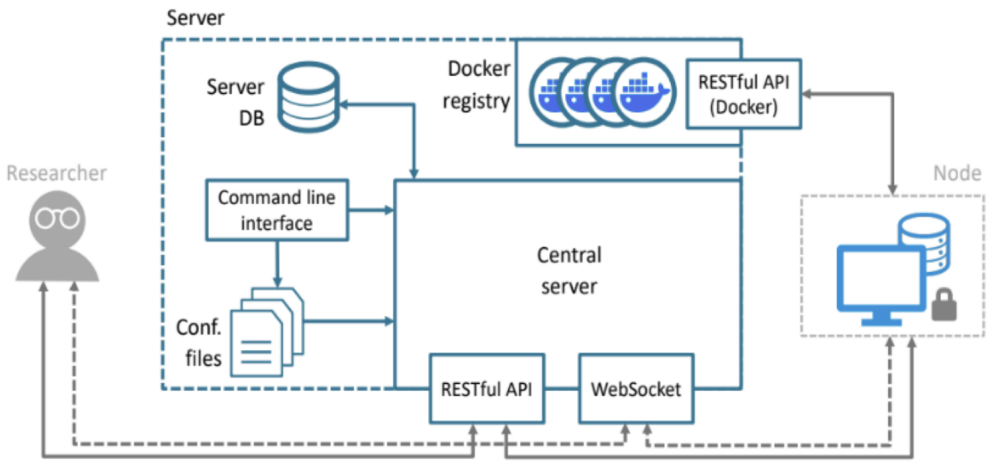

Figure 2.2: The architecture of Vantage6 (2)

Brane Brane (6) is an open-source framework for distributed learning projects. To make programming easier for domain experts, Brane is developed with a user-friendly programming model and an efficient runtime system that orchestrates the data processing pipelines. Figure 2.3 illustrates the programming model and runtime system in Brane. In the programming model, a component named Packages is used to build new functions using various builders and the data processing pipeline is created using Bakery which is a simple DSL (Domain Specific Language) while the Bakery programs can be compiled in Bytecode. Domain experts can use the Jupyter notebook in Brane to write applications and monitor their progress in real time. BraneScript is a C-like programming language that has advanced constructs. Brane repl is an interactive shell environment that takes user input from BraneScript, executes the commands and returns results. Users can simply start the repl in the terminal and use the compiled Brane functions following the Brane syntax.

As for the runtime system, it consists of four major parts: Registry, API, Relay and Vault. Registry is used to store the packages built by different package builders. After a data processing pipeline is built, a session needs to be created with runtime system through API for the execution of Bakery programs. Apart from it, Relay is used to store functions outputs while Vault is used to store secret system information, such as credentials and certificates. With different components cooperating with each other, Brane is able to provide an easy-to-use programmatic infrastructure for developing research projects.

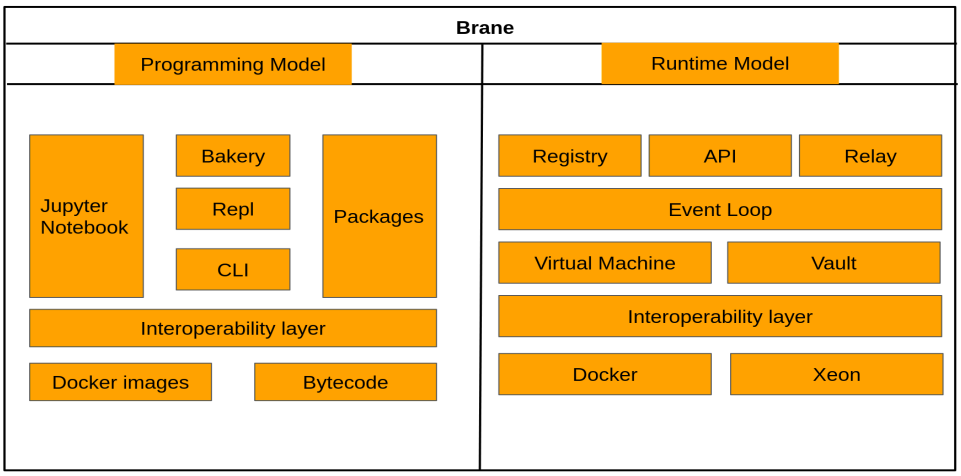

Figure 2.3: The architecture of Brane

REST API REST API (13) is used in Vantage6 server for interacting with user and nodes. It represents representational state transfer application programming interface, following 6 principles. The first principle is having a client-server system architecture. The second principle is being stateless with session state kept completely on the client side. The third principle is being cacheable and having client cache to be able to reuse response data. The fourth principle is having a uniform interface that provides identification of resources, manipulation of resources, self-descriptive messages and hypermedia. The fifth principle is have hierarchical layered system and the last principle is to allow clients to code on demand. REST is associated with HTTP resource methods, including GET, PUT, POST and DELETE methods. The resources in the REST API can be accessed through uniform resource identifiers (URIs) in combination with the resource methods. Many applications use REST API for internal communication, such as Docker.

OpenAPI Specification OpenAPI Specification (OAS) (14) defines an interface for HTTP API and we use it to build API functions in Brane. An OpenAPI document can be represented in either JSON[1] or YAML format[2]. OAS usually includes the version of the specification, metadata of the API, server objects, API path, security requirement object and component object. Component object consists of a list of reusable objects and the commonly used fields are requestBody, responses and schemas. Under each field, the properties for each object should be defined with names and data types. OAS is often used for the development of web APIs for applications such as Vantage6.

Docker Docker (3) is used in both Vantage6 and Brane infrastructure. Figure 2.4 shows the architecture of Docker. Docker is a open platform that provides users an isolated environment to run their applications, which is referred to the docker container. A user can install any packages inside a new docker container without relying on the system environment of the host machine. A docker image contains a user-defined algorithm with instructions to run a container. A user can create a docker image using Dockerfile following the specific syntax. When building an image, each instruction in the Dockerfile creates a layer to be installed in Docker for the image. When changes are made by the user and the image needs to be rebuilt, the only part that will be rebuilt is the layers that are changed. Such feature makes a docker image lightweight and easy to manage. Each docker image is built by the docker client and stored in the docker registry. Docker compose is a

[1]https://www.json.org/
[2]https://yaml.org/

special docker client that is used to deal with a set of containers for the user application to run. Docker uses the client-server architecture to manage docker containers and images. An image can be built locally or pulled from the internet. Docker daemon is a component inside docker host (server) that is used to communicate with docker client using REST API and handle the container to run an image. To run an application using Docker commands, users can have easy control over containers or images.

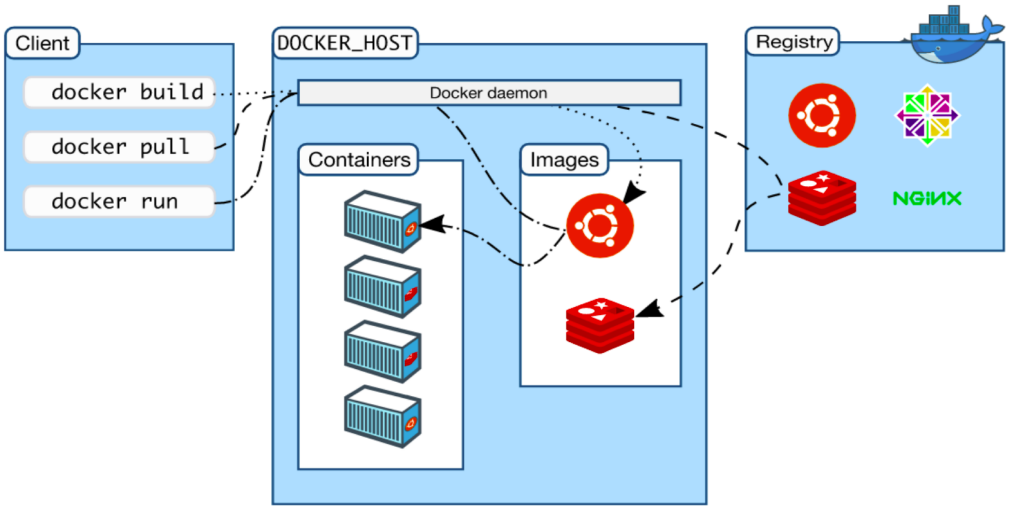

Figure 2.4: The architecture of Docker (3)

3

Design of the Integration

In this chapter, we introduce the workflow of a FL pipeline in Vantage6 and the architecture design of the proposed integration, including Scenario 1 and Scenario 2. We first describe how a FL pipeline is constructed in Vantage6 and then propose the integration. The detailed explanation of the design is shown in the following sections. The implementation details of the FL pipeline and integration are explained in Chapter 4.

As mentioned before, Vantage6 is an open-source framework for FL. To access the FL services in Vantage6, we need to build a FL pipeline. Figure 3.1 shows the main components (including server, node and researcher (user)) and their interactions in Vantage6. Figure 3.2 illustrates the workflow of a FL pipeline constructed based on the architecture of Vantage6. Starting from installation of Vantage6 packages, a server and one or more nodes are needed as the basic infrastructure to establish the FL process. A server can coordinate the FL process, distribute tasks to the nodes and collect results. A node contains non-shareable local data and can be used to execute the distributed tasks. Both server and nodes need to be configured using YAML files[1] and started using Vantage6 commands (15) before any other services can be used. Once both server and nodes are started, the connection between nodes and server can be automatically established with the right server IP address and node api key.

[1] An example of a configuration YAML file can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/tree/main/V6

3.1 Vantage6 Architecture and the FL Pipeline

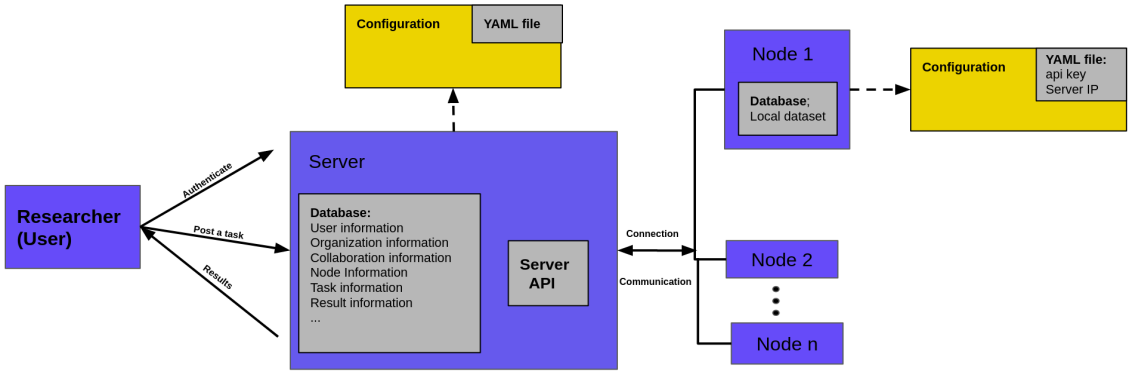

Figure 3.1: The main Vantage6 components (The blue boxes are the three main components in Vantage6: server, node and researcher. The gray boxes inside blue boxes show the main sub-modules in each component. Server component has two main sub-modules, including database and server API. Node component has one sub-module, which is the database. The yellow boxes are the actions required for a server/node to run, which is the configuration process. The gray boxes inside the yellow boxes show the modules needed for the actions, which is the YAML file required for configuration.)

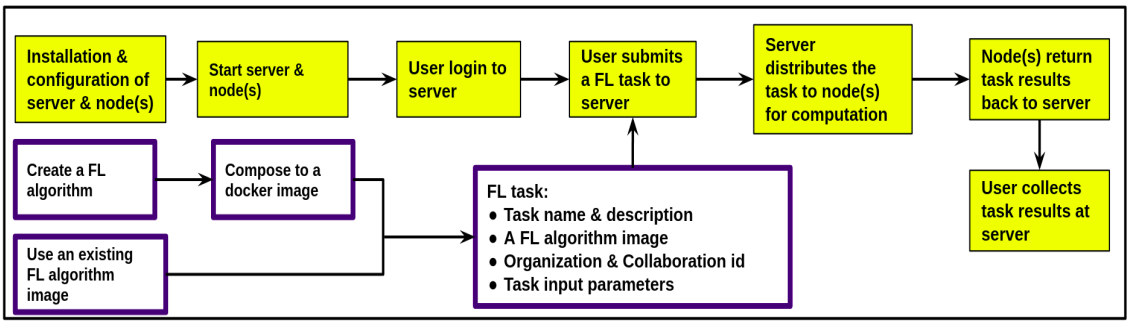

Figure 3.2: The FL pipeline in Vantage6 (The yellow boxes show the flow of a FL pipeline in Vantage6, starting from installation and configuration of the Vantage6 infrastructure to user login, submitting a FL task and collecting results. The purple-bordered boxes indicate the process of using/creating a FL algorithm in Vantage6, which is required for submitting a FL task.)

The next step in a FL pipeline is to access server as a user and submit a FL task for the actual computation. Server has a database that stores many kinds of information,

such as node information, task and result information and user information along with the organization and collaboration every user belongs to. To access a server, a user needs to connect to server using server IP address and get through the authentication process. To submit a task in Vantage6, a FL algorithm image is required as well as some user-defined task input parameters. If the user wants to use a new FL algorithm that is not currently supported by Vantage6, the FL algorithm needs to be implemented first following the documentation (details explained in Chapter 4.3) and composed into a docker image using Docker. The built image is needed by the server to distribute to each worker node for computation. The communication between node and server happens automatically in Vantage6 through server API, as well as the communication between server and the user.

The final result is returned by the node and saved in the server database and can be accessed by the user that has the permissions.

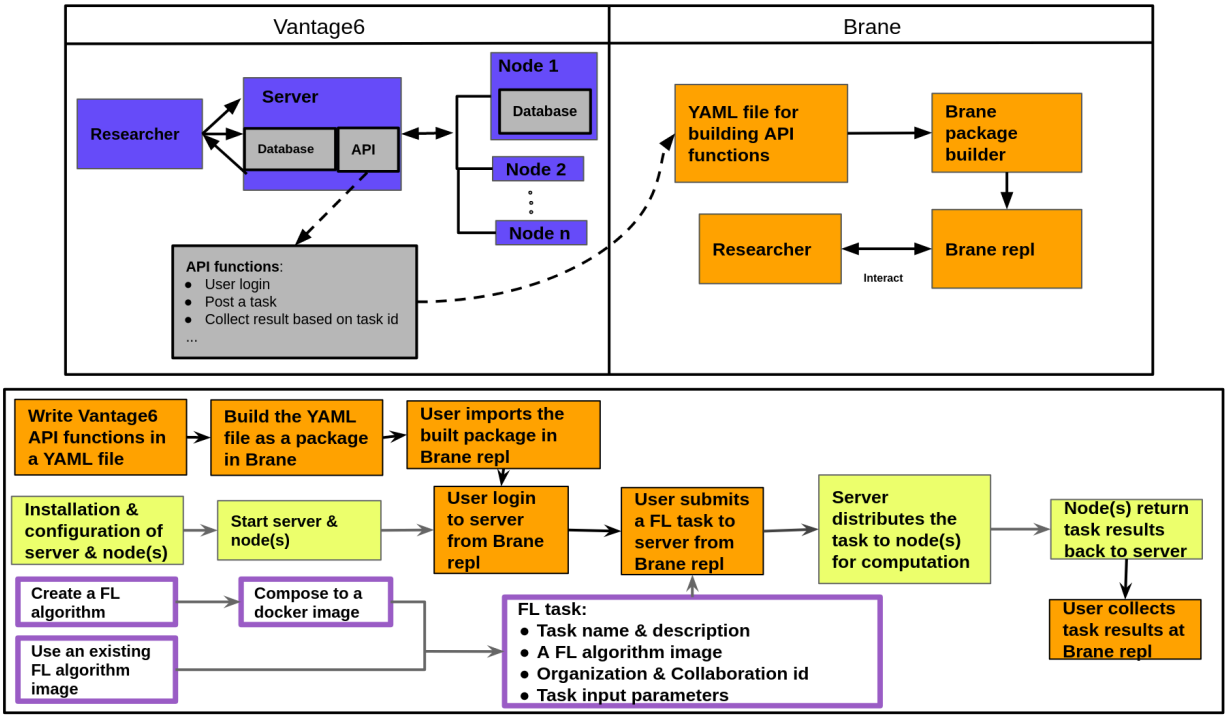

Figure 3.3: The architecture design and FL pipeline in integration Scenario 1 (The top layer of this figure shows the components of Brane and Vantage6 that are needed in the proposed integration. The bottom layer shows the flow of a FL pipeline for integration Scenario 1. The orange boxes represent the actions specific to integration Scenario 1. The light yellow boxes and light purple-bordered boxes indicate the same actions as Vantage6 FL workflow as shown in Figure 3.2.)

3.2 Integration Design and the FL Pipeline

To integrate Brane and Vantage6, we propose the light integration that makes use of API functions calls for accessing Vantage6 functions in Brane. Two scenarios are considered for the integration, including

- Scenario 1: users have pre-installed Vantage6 infrastructure, such as Vantage6 server, node and relevant dependencies.
- Scenario 2: users do not need to install Vantage6 separately but build the necessary functions inside Brane to use existing and already deployed Vantage6 services.

The details of the integration under different scenarios are described in the following.

Scenario 1 In Scenario 1, Vantage6 server and nodes are pre-installed and configured separately from Brane. To allow access of Vantage6 functions in Brane, a light integration is implemented based on the API functions calls. The details of the designed workflow in this scenario along with a FL pipeline are shown in Figure 3.3. The FL algorithm is created outside of the integration using Vantage6 functions. The integrated functions are the API functions that are used to interact with server. Originally, vantage6 server API has functions that enable user authentication, task posting, result collection and etc. In the integration, we build these API function as a package using docker YAML file in Brane so that a user can access Vantage6 functions in Brane for using FL services. The workflow of the FL pipeline in Scenario 1 is the same as the original one in Vantage6 except that the user interacts with Brane using the repl instead of Vantage6 when accessing the aforementioned functions.

Scenario 2 Similar to Scenario 1, Scenario 2 is also designed based on Vantage6 functions. However, in Scenario 2, we assume the user only have Brane installed but not Vantage6 infrastructure. Therefore, to construct a FL pipeline in this scenario, the user needs to build Vantage6 services inside Brane, including the server, the node and API functions. Figure 3.4 shows the workflow designed for Scenario 2 in a FL pipeline. As mentioned above, Vantage6 server and nodes are configured in YAML files. The same configuration files are needed by the package builder to build the corresponding function packages in Brane. The general process of building API functions in Brane is the same as Scenario 1. Then, a user can import the built packages in Brane repl and make function calls including starting server and nodes, user login, submitting a task and collecting results. In Scenario 2, server and nodes are started in Brane as containers with the same

functionalities as the original server and nodes in Vantage6. Thus, user can still access server through API function calls and use the same FL services.

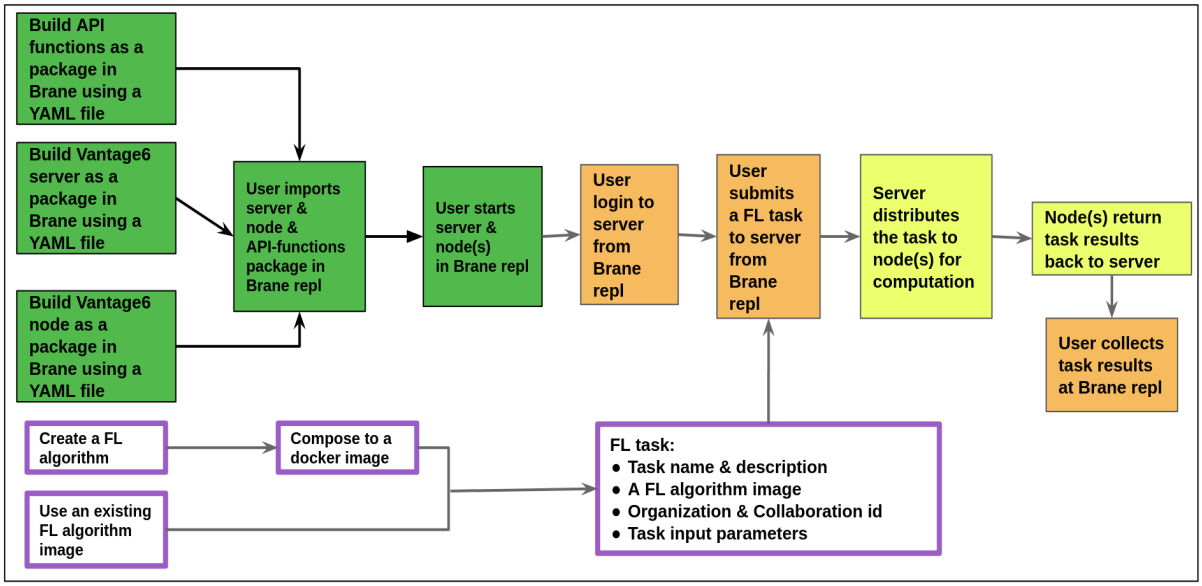

Figure 3.4: The workflow design of integration Scenario 2 in a FL pipeline (This figure shows the flow of a FL pipeline in integration Scenario 2. The green boxes represent the actions specific to integration Scenario 2. The light orange boxes, light yellow boxes and light purple-bordered boxes indicate the same actions as the integration Scenario 1 as shown in Figure 3.3.)

4

Implementation

In this chapter, based on the designed architecture in Chapter 3, we introduce implementation details of

- Building a FL pipeline in Vantage6
- The proposed integration, including
  - Integration scenario 1
  - Integration scenario 2
- Creating FedAvg algorithm in Vantage6 under two use cases, including
  - FedAvg MNIST
  - FedAvg Sentiment

To evaluate the integration of the two frameworks in a FL setting, we implement a FL pipeline for each scenario. Meanwhile, we implement the FedAvg algorithm with two use cases to test the performance of the integration for different ML use cases. The implementation is detailed in the following sections. Some technical issues encountered during the implementation is included in Appendix (8).

4.1 Building a FL Pipeline in Vantage6

As mentioned in Chapter 3, to build a FL pipeline in Vantage6, we need to setup Vantage6 server and nodes properly, have a FL algorithm and run it using Vantage6 services (see Figure 3.2). As the FL algorithm is created separately from Vantage6 node and server, we introduce the implementation details of the algorithm later in Section 4.3. In this section, we introduce

- Installation and configuration of Vantage6 infrastructure
- How to run a FL algorithm using

- API functions through Vantage6 iPython shell
- A Python script through Vantage6 Python client

4.1.1 Vantage6 server and node setup and configuration

Before using any Vantage6 services, generally, a user needs to first install and configure the required packages (e.g. Vantage6 node, server or client) on their local machine following the documentation (15). In the real-case FL scenarios, the user can either use their local machine as a server or a node.

Server configuration: a server can be configured using a server configuration file (e.g. *server.yaml*[1]) that includes information such as api path, IP address, port number, database URL, logging information and JWT[2] (JSON Web Tokens) secret key. Node configuration: a node can be configured through a node configuration file (e.g. *node.yaml*[3]) that is composed of: (1) server URL, (2) database path, (3) api key, (4) api path, (5) task directory, (6) logging and encryption information. A Vantage6 server or node can be started using the provided commands on Vantage6 documentation (15).

4.1.2 API functions through Vantage6 iPython shell

To run a FL algorithm using API functions, the user needs to access server through iPython shell, setup server database and perform a list of function calls for the execution of a FL algorithm.

Server database setup in iPython shell To access the server from iPython shell, we need to specify the server URL and port number and login to server as a user. In the server database, the default user is set to be root while there is no existing collaboration or organization in the server. A collaboration contains a list of organizations. An organization contains a list of users. Each organization inside a collaboration is assigned a node. A user is associated rules that specify what functions the user has access to. The root user has limitations to access some of functions by default, including submitting a task. A user can only submit a task to the collaboration where the user belongs to. Therefore, we need to create new users with the desired rules and add new collaborations and organizations. For convenience, a number of entities (e.g. information of a user/organization/collaboration

[1] An example of server.yaml can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/blob/main/V6/server.yaml

[2] https://jwt.io/introduction

[3] An example of node.yaml can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/blob/main/V6/node.yaml

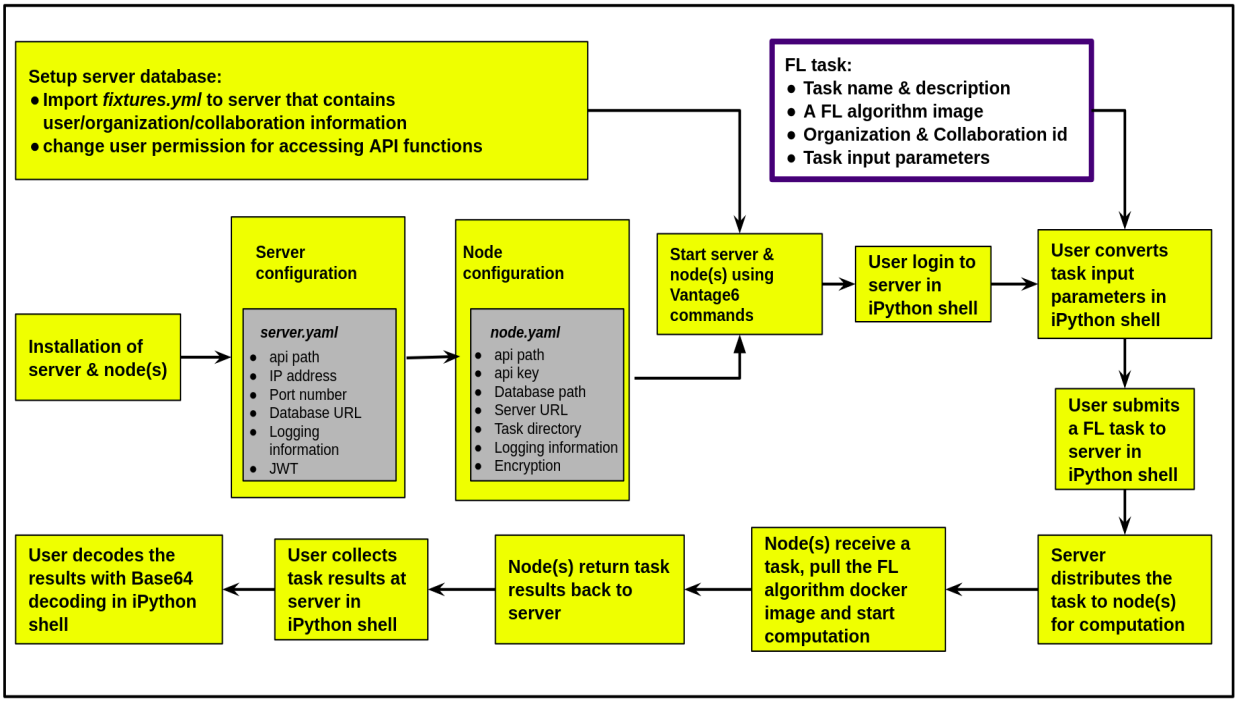

Figure 4.1: The detailed steps for constructing a FL pipeline using Vantage6 iPython shell (The yellow boxes represent the steps required for constructing a FL pipeline in Vantage6 using iPython shell. The gray boxes inside the yellow boxes show the components needed for serer/node configuration, which is the configuration YAML files. The purple-bordered box shows the FL task parameters needed for submitting a task.)

that can be stored in the server database) can be imported to the server through iPython shell using a YAML file (e.g. *fixtures.yml*[1]). The YAML file contains a list of user-defined collaborations with the organizations and users included. Then, after starting the shell, the user can log in with the predefined user name and password and assign the rules (permission to view/edit certain functions) to himself/herself, depending on which functions are needed. Every API function is accessible using GET, POST or DELETE requests and it is secured with the token generated after login.

FL task in iPython shell As shown in Figure 4.1, for running a FL task successfully in the iPython shell, we need to do the following steps:

- First, both the server and the node(s) need to be installed and configured with

[1] An example of a fixtures.yml file can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/blob/main/V6/fixtures.yml

configuration files (as described in Section 4.1.1).

- The server database needs to be setup as described in the previous paragraph.
- A node configuration file should include the full path of the local data file so that it can be detected by the node automatically for later computation. A node (organization) is also configured with either no encryption or a private key path. In the node configuration file, a user should set the private key path or disable the encryption of the collaboration of which the organization belongs to for the node to run successfully. An api key is assigned by the server and unique to a node for identification. A user needs to check the api key in the shell and configure a node with the corresponding api key.
- Both the node (s) and the server can be started with the Vantage6 command (15). After a node is started, it connects to server and authenticate itself with the api key automatically.
- To run a FL algorithm in iPython shell after user login, the user would need to submit a FL task with the required parameters, including task name, task description, image name, input parameters for initiating the algorithm, collaboration id and organization id.
- Due to the limitation of Vantage6 API, task input parameters needs to be converted to a serializable format before submitting the task (as explained in Appendix (8)).
- A running node frequently checks if a task related to the corresponding organization has been submitted to the server.
- Once a running node receives a task, it pull the docker image based on task information, start the computation using the local data automatically and send results back to server.
- All the exchanged messages between the server and the node are encrypted in Vantage6.
- Lastly, to obtain results in the end of the FL pipeline, a user needs to collect results from server and decrypt the results with Base64 decoding[1] in the shell.

4.1.3 Python Script through Vantage6 Python Client

To use the Python client functions instead of the API functions, the user needs to import the vantage6 client package and write a Python script.

[1] https://www.base64decode.org/

In the Python script[1], the user can access the server by stating the server URL and the authorization details. Similar to the process in the shell, the user will need to set up encryption for the collaboration and assign user rules before calling any other functions. For submitting a task, the user needs to list the required task properties in the "task.create" function. To collect the results, the user needs to constantly check the status of the task in a loop using the "task.get" function. The results will only be available after the execution of task is complete. The task input conversion and results decoding are performed automatically by Vantage6 client. To construct a FL pipeline using the client functions, the user can simply run the script in the terminal and the process of login, task submission and result collection can be performed accordingly.

4.2 Implementing the proposed Integration

As mentioned in Chapter 3, the proposed integration is based on Vantage6 API functions calls. In this section, we introduce how to implement the integration by calling API functions under two scenarios.

4.2.1 Scenario 1

In Scenario 1 with pre-installed Vantage6 server and nodes, we build Vantage6 API functions as a package (*vantage6* package) in Brane using a Docker YAML file (e.g. *api_ spec.yml*)[2] to perform function calls in Brane repl.

API function calls using YAML file in Scenario 1 As mentioned above, the API functions are composed in a YAML file in Brane. To make the YAML file, we use OpenAPI specifications (14). The YAML file serve to define a FL pipeline. It should at least contain functions such as user login, submitting a task and retrieving results based on task id. For the ease of testing and debugging, We also include other functions such as GET collaborations, GET tasks, GET and POST users. In the YAML file, the servers property should be first specified with server URL and the specific port that is used by the vantage6 server. For each function, the path needs to be listed before specifying the parameters for GET or POST requests. For each GET or POST request, it requires operationId, security, requestBody and responses. operationId is the function name that the user uses

[1] An example of a Python script can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/blob/main/v6-fedavg-mnist-master/v6-fedavg-mnist/client.py

[2] An example of the api_spec.yml file can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/blob/main/Brane/api_spec.yml

when calling the function in Brane repl and security scheme object is for specifying the authorization token (obtained after login) needed by the function to be accessed. In both requestBody and responses, the user can define the content in JSON format and decide whether a property is required by the function or optional. Besides, a HTTP response code (e.g. 200) needs to be specified in responses that indicates the operation is successful. The commonly used data type are usually integer and string for properties such as id, name, token, URL and description. However, for more complex properties, such as the task organization, it is essentially an array object. An array object contains array items, such as the organization id (integer) and the task input (string). The data types are usually declared directly under the property. However, for nested objects (e.g. array), sub-items (e.g. array items) can also be specified using a reference to a separately defined component scheme at the end of the YAML file.

FL pipeline for Scenario 1 Figure 4.2 shows a detailed FL pipeline in scenario 1. Similar to the process of using Vantage6 services in Section 4.1, building a FL pipeline in integration Scenario 1 also requires pre-installed and pre-configured server and nodes. The Vantage6 iPython shell is only used for disabling encryption and assigning rules to users after importing entities. To make Vantage6 API functions available in Brane, the package builder targeting Web APIs can be used to build the Vantage6 YAML file (as described above) for users to access. Due to limitations in API functions, Python functions of input conversion and decoding are both built as brane packages[1] for constructing the FL pipeline (details explained in Appendix (8)). After building all the necessary packages, the user can start brane repl and import the installed packages. The functions are callable by the pre-defined function names using Brane programming syntax. The brane functions are generally easier to use with shorter commands compared with the ones in the iPython shell. To construct the FL pipeline in this scenario, the pre-installed Vantage6 server should be started before any Vantage6 API functions are called. The Vantage6 nodes that are supposed to run the tasks need to be started as well to wait for the assigned task. Then, in the repl, the user can follow the procedure and perform the corresponding function calls, such as login, specifying the input and converting it to JSON serialized format, posting a task to the organizations in a collaboration and collecting results.

[1]Examples of input conversion and decoding package can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/tree/main/Brane

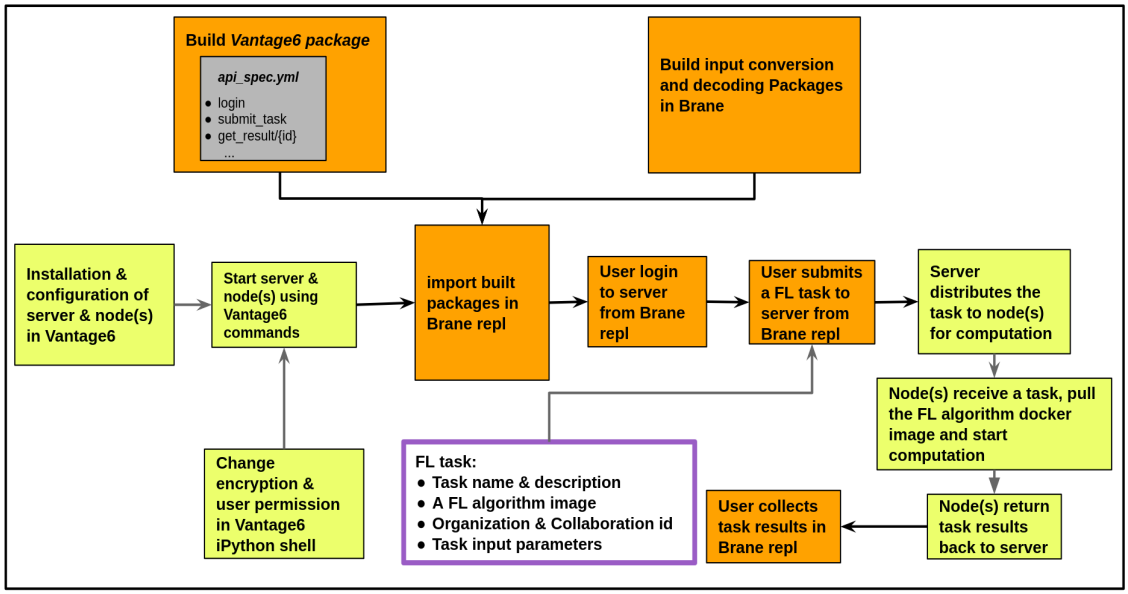

Figure 4.2: The detailed FL pipeline in integration Scenario 1 (The Orange boxes represent the actions to construct a FL pipeline that are specific to Scenario 1. The gray box inside the orange box shows a component that is needed for building API functions as a package in Brane, which a YAML file specifying each Vantage6 API function. The light yellow boxes and light purple-bordered boxes are the actions that are the same as the workflow in Vantage6 as shown in Figure 4.1.)

4.2.2 Scenario 2

As mentioned in Chapter 3, Scenario 2 has no pre-installed Vantage6 infrastructure. Thus, besides using API functions as described in Scenario 1, we need to build Vantage6 node and server as packages in Brane.

Building Vantage6 server and nodes in Brane To build a server service or node service using the Brane package builder, a package configuration YAML file (e.g. *container.yml* (see the example in Appendix (8)) is needed to specify the necessary information such as service name, entrypoint, dependencies, files needed to build the package, required installations, function input and output with the corresponding names and data types and etc. The function input refers to the real values passed from the brane repl when the function is called and output is the returned values of the function. An executable shell script (e.g. *entrypoint.sh* (see the example in Appendix (8)) is also needed for building a server or node. Such shell script is linked with the package configuration file and defines

variables that should be passed to the server or node configuration file (e.g. *configuration.yml*(see the example in Appendix (8)). The corresponding properties in the server or node configuration file should be replaced by the user-defined variable names based on the shell script.

Vantage6 server package building For building the server package (*v6_server* package), the server configuration file remains the same as the original one used in normal Vantage6 infrastructure, except that the server port is replaced by a variable. It means that when the server is started in the repl, the user can define the port number flexibly. The *container.yml* for the server specifies the function input as an integer which is the server port number. It also defines the function output which is a string that contains the remotely detected server IP address in the server container created by Brane. In the *entrypoint.sh*, the user needs to specify the variable name for the server port and specify the command to start the server with the server *configuration.yml* and the *fixtures.yml* for importing entities. The variable name in the *entrypoint.sh* should be the same as the one in *configuration.yml*. Besides, for the ease of experiment, all the Vantage6 nodes are configured without encryption. To match with this setting, the encryption for the corresponding nodes needs to be disabled in the server database. In Scenario 1, this can be done by using commands in the iPython shell. In Scenario 2, disabling encryption can also be specified in the server entrypoint.sh. However, the iPython shell is also accessible in this scenario through the server container once the server is started in the repl. Users can use the shell to change user rules and use other functions conveniently according to their preference.

Vantage6 node package building To build a Vantage6 node package (*v6_node* package) in Brane, similar to the process of building a server, the original node *configuration.yml* is used and the node *container.yml* is specified in a similar format. However, the api key, server URL, port number and database path in the configuration.yml are replaced by variables. The values of these variables are passed as function arguments by users in the repl when the node is started. Same variables are used in the node *entrypoint.sh* and *container.yml* to make sure the function arguments are passed correctly. Due to some technical issues (details explained in Appendix (8)), the task directory, proxy server address and proxy server port are explicitly specified in node *configuration.yml* as well as *entrypoint.sh* for successfully running a node in Brane. Lastly, it should be specified in the node entrypoint.sh that the node needs to be started with the node *configuration.yml*.

Then, the Brane package builder can build the node package for the user to call in the repl.

API function calls using YAML file in scenario 2 Similar to the vantage6 package in Scenario 1, a YAML file is used to build Vantage6 API functions (*v6_ client* package[1]) in Brane following the OpenAPI specifications (14). However, because the server IP is automatically detected in Brane when creating the server container, the server URL cannot be specified when building client functions. Thus, unlike vantage6 package, the server URL is not specified in *v6_ client* package. Meanwhile, the path to each function in *v6_ client* should be added with an additional api path. When a function from *v6_ client* package is called, the server URL should be passed as a function argument in addition to the originally specified input for each function.

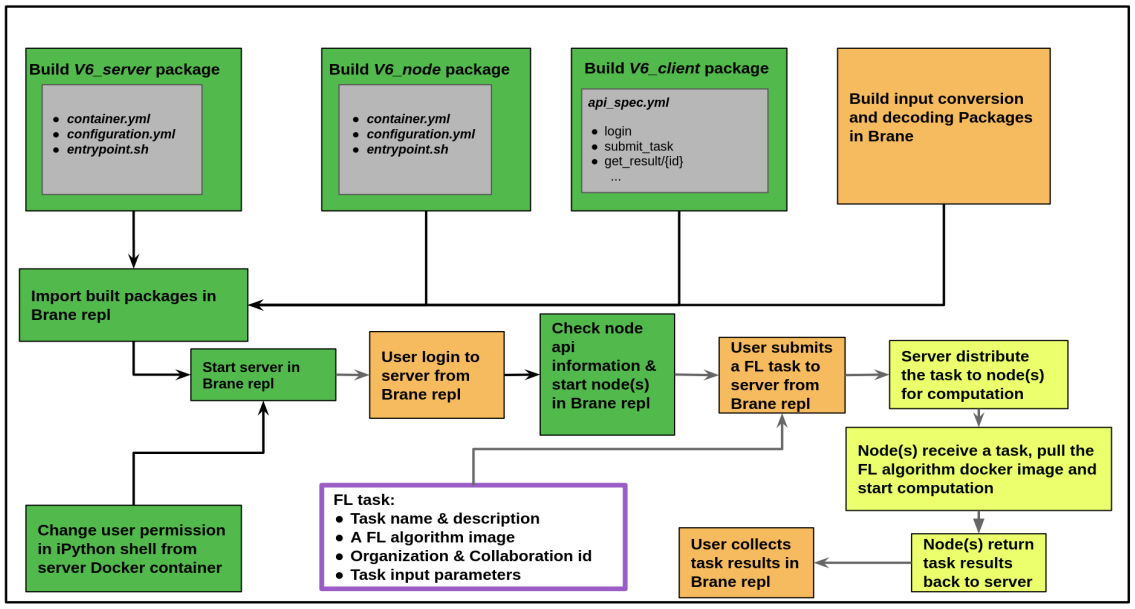

Figure 4.3: The detailed FL pipeline in integration scenario 2 (The green boxes are the actions for constructing a FL pipeline that are specific to Scenario 2. The light orange boxes, light yellow boxes and light purple-bordered box indicate the actions that are the same as Scenario 1 as shown in Figure 4.2.)

FL pipeline for Scenario 2 Figure 4.3 illustrates a detailed FL pipeline in Scenario 2. In contrast to Scenario 1, Scenario 2 does not have pre-installed server or nodes. To con-

[1] An example of V6_client package can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6/tree/main/Brane/V6_server_node_deployment/v6_client

struct a FL pipeline, the user needs to first import the *v6_server*, *v6_node* and *v6_client* package (the details of the packages are described above) and start a server with the user-defined port number. The input conversion and decoding packages are also needed in this scenario. Then, the iPython shell needs to be started using the server container ID. As mentioned above, the user needs to change permission rules to a certain user in the shell for acquiring access to the desired functions in repl. After that, the user needs to log in the server in the repl and start the corresponding nodes based on the api key information specified in the server database (this can be accessed in the repl by calling *get_node* function). While the server and nodes are running successfully, similar to Scenario 1, the user can perform a series of function calls (e.g. task submission and results collection) to finish constructing the FL pipeline.

4.3 Implementing FL Algorithms in Vantage6

As mentioned in chapter 3, a FL pipeline in Vantage6 requires the execution of an FL algorithm. In this section, we introduce

- The general process of creating a new FL algorithm in Vantage6
- Details of implementing FedAvg algorithm in two use cases
  - FedAvg MNIST
  - FedAvg Sentiment

To create a new FL algorithm in Vantage6, a specific package structure needs to be followed[1]. Under the project folder, at least a Dockerfile, a *setup.py*, an algorithm package (folder) containing a *_init_.py* are needed for the Docker image to be built. The Dockerfile includes the information of some commonly used dependencies, name of the algorithm package, the installation commands for building the image and which Docker wrapper is being used for the algorithm. The setup.py specifies the required dependencies for the algorithm as well as some basic descriptions (e.g. name and version). In *_init_.py*, a master method is needed to orchestrate the workflow and one or more RPC methods are needed for the nodes to perform the computation. Every RPC method gives the node instructions on the computation steps of the algorithm, starting from data processing to model training and testing. An RPC method will be triggered by the master method to start executing the algorithm and return the output to the master method after execution. The master method sends out tasks to each node (calls the specific RPC methods) with

[1]An example of a FL algorithm in Vantage6 can be found on GitHub: https://github.com/621Alice/Fed-Brane-V6

user-defined input parameters. Then, it will wait for the returned output from the RPC methods, average the collected results and send them back to the RPC methods for another round of execution until the algorithm converges. During the execution of the algorithm, the master method will create a master Docker container where the final result will be returned. Each task that the master method sends to the node will create a Docker container for itself to return intermediate results.

Both of the use cases are implemented using PyTorch[1] based on the structure of the simplified FedAvg example on Vantage6 documentation (15) and further developed to suit the specific dataset. Due to the limitation of current version of Vantage6 (details explained in Appendix (8)), we combine training dataset and test dataset in one CSV (comma-separated values) file specifically for each node to execute the algorithm. More details for each use case are described in the following.

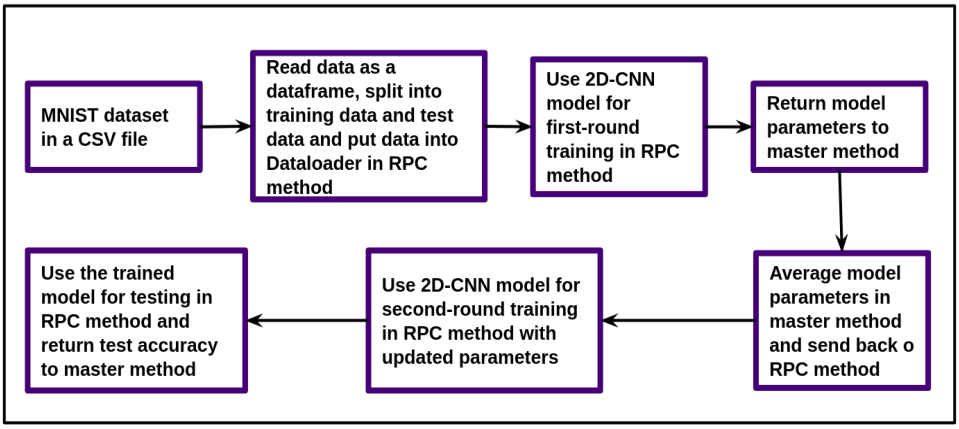

Figure 4.4: The workflow of FedAvg MNIST (Following the arrows, the purple-bordered boxes represent each step needed for implementing a FedAvg algorithm for MNIST image classification.)

4.3.1 FedAvg MNIST

FedAvg MNIST uses MNIST dataset[2] for 10-class image classification. MNIST dataset consists of grayscale images of hand-written digits from 1 to 10. We use the MNIST CSV data files that represent images using pixel values. Each line in the data file is composed of the image label (in number) and 784 pixel values (ranging from 0 to 255) of an 28 X 28 image. There are in total 70,000 data samples included in the dataset. I split the

[1] https://pytorch.org/
[2] https://www.python-course.eu/neural_network_mnist.php

dataset equally for two nodes where each node has 35,000 data samples. The image data is partitioned horizontally with same features (label and pixel values) but different images.

The workflow of FedAvg MNIST is shown in Figure 4.4. For image classification, we use a simple convolutional neural network(CNN) model that consists of two 2D convolution layers, two dropout layers and two linear layers. To train the model using PyTorch, we first separate the image labels and pixel values, normalize the pixel values and transform the data into tensors. The PyTorch Dataloader[1] is used to prepare the datasets for model training and testing. During model training, the stochastic gradient descent (SGD) optimizer is used with a learning rate of 0.01. The loss function used in the training process is the PyTorch negative log likelihood loss which is useful for classification problems. The master method calls the RPC method for model training. The model is trained on each node for a certain (user-defined) number of epochs. The updated model parameters will be returned to the master method for averaging. The master method will call the RPC method again with the averaged parameters. Then, the node will train the model for a second round with the updated parameters and use the resulting model for local testing.

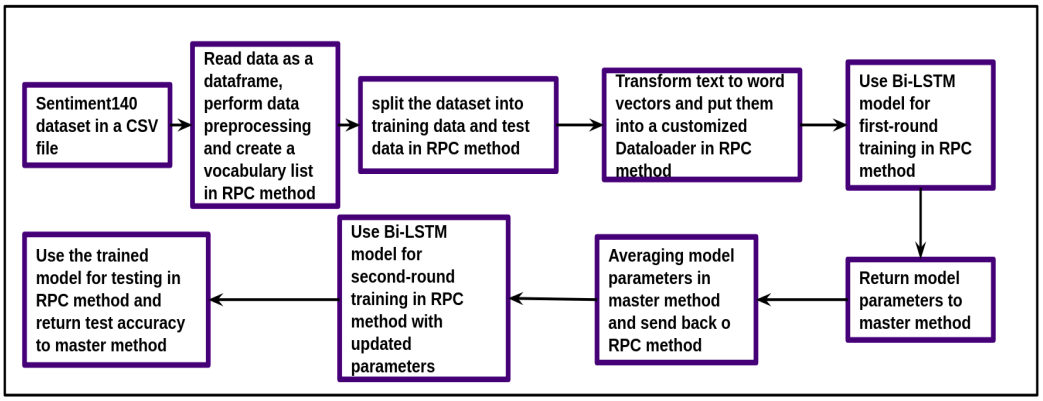

Figure 4.5: The workflow of FedAvg Sentiment (Starting from processing the Sentiment140 dataset, the implementation process of a FedAvg algorithm for sentiment classification is shown in the purple-bordered boxes.)

4.3.2 FedAvg Sentiment

FedAvg Sentiment uses Sentiment140 dataset[2] for binary sentiment classification. Sentiment140 is a dataset of tweets that are collected from Twitter API[3] and automatically annotated based on emoticons (16). It includes information such as sentiment label, user

[1]https://pytorch.org/docs/stable/data.html
[2]http://help.sentiment140.com/for-students
[3]https://developer.twitter.com/en/docs/twitter-api

name, tweet text, tweet ID, date and etc. Only the sentiment label and tweet text are preserved for model training and testing while the rest of the unnecessary information is removed. Each tweet is either labelled as positive (with number "4") or negative (with number "0") in the processed dataset. We use the same number (70,000 for two nodes and 35,000 for each node) of data samples as the FedAvg MNIST for the ease of testing.

The workflow of FedAvg sentiment is shown in Figure 4.5. In the RPC method for model training and testing, the tweet data is first processed by making text lowercase, removing irrelevant tags (e.g. "html" and "@"), English stop words and punctuation. The processed text is split by space to create a vocabulary list for the whole dataset before splitting the data for training and testing. The vocabulary list is used to create a word dictionary that matches each word with an integer index. Each text is transformed into input word vectors using the word dictionary before being processed in the Dataloader. Inside the Dataloader, an additional collate function is used to pad all the inputs based on the longest tensor and combine the padded sequences into each batch. The neural network model used to classify sentiment is composed of an embedding layer, three bidirectional long short-term memory (Bi-LSTM) layers, three recurrent neural network (RNN) layers and one linear layer. During the model training, Adam optimizer is used combined with the cross-entropy loss function. The learning rate is relevant to the batch size. Similar to the FedAvg MNIST classification, the master method calls the RPC methods twice for model training, resulting in two communication rounds. In the last round, model testing is performed using the updated model on the local test set.

5

Experiment Design and Results

In this chapter, we introduce the experiment design and results for

- Evaluation of the FL algorithm
- Evaluation of the integration and Vantage6, including
  - Local testing
  - Remote testing

During the experiments, the implemented FL algorithm, FedAvg, is evaluated under two use cases to test the effectiveness of the applied ML model. The proposed integration is evaluated in a FL pipeline for each scenario using two use cases of FedAvg. The implementation details of the integration and the FL algorithm are included in Chapter 4. To test the efficiency of the integration, the performance of the standalone Vantage6 is used as the baseline for comparison. A list of experiments is summarized in Table 5.1. The details of different experiment setup are described in the following.

Experiment design and evaluation metrics for the FL algorithm The FL algorithm FedAvg is evaluated using the two proposed use cases, including FedAvg MNIST and FedAvg Sentiment. The experiments are performed on Vantage6 mock client. We chose model accuracy as the evaluation metric for the FedAvg algorithm in both use cases. Mock client is a pseudo Vantage6 client that simulates the FL process for the purpose of testing FL algorithms locally. The evaluation of the algorithm is designed to test the effect of number of epochs and the number of nodes on the model performance. Thus, we designed 6 experiments for each FL use case, where each use case is evaluated using 2, 10 or 20 epochs on 1 or 2 nodes respectively (Table 5.2 and Table 5.3).

Experiment	Client	Testing Method	FedAvg MNIST	FedAvg Sentiment	Number of Node(s)
Model accuracy evaluation	Mock client	local testing	✓	✓	1 & 2
Vantage6 performance evaluation	Mock client	local testing	✓	✓	1 & 2
Vantage6 performance evaluation	Real client	local testing	✓	✓	1 & 2
Integration scenario 1 performance evaluation	Real client	local testing	✓	✓	1 & 2
Integration scenario 2 performance evaluation	Real client	local testing	✓	✓	1 & 2
Integration scenario 1 performance evaluation	Real client	Remote testing	✓	✓	1 & 2
Vantage6 performance evaluation	Real client	Remote testing	✓	✓	1 & 2

Table 5.1: List of Experiments

Experiment design and evaluation metrics for the integration As shown in Table 5.1, both scenarios of the integration are evaluated and compared with the performance of standalone vantage6. The evaluation for standalone Vantage6 and each integration scenario is performed using the corresponding FL pipeline described in Chapter 4. Before starting an experiment, the Vantage6 server and corresponding Vantage6 node(s) for computation need to be started. In each experiment, a FL task is submitted using either FedAvg MNIST algorithm image or FedAvg Sentiment algorithm image. The execution of a FL task can be performed on either one or two Vantage6 nodes. Each node holds the equal amount of data that is needed to train the corresponding FL task. For instance, when using FedAvg MNIST algorithm image, each node participating the task execution is configured with a local dataset of 35000 MNIST data samples. Similarly, when running FedAvg Sentiment, each node needs to be re-configured with the data path to the Sentiment140 data samples. The evaluation metrics for the integration include the running time, CPU and Memory that are used for running a FL task.

- The CPU usage is shown in percentage in the experiment results, indicating how much of the processor's capacity is used by the running process/container.
- The memory usage is measured by either MiB(Mebibyte) or GiB(Gibibyte) to represent the amount of memory used during the task execution.

For the ease of testing, the number of epochs to train each FL algorithm is set to be 2 for

each experiment. To verify the effectiveness of integration for FL process, the experiments are conducted not only locally on a local machine but also remotely using virtual machines (VMs). The details of the two types of testing are illustrated in the following.

Local testing setup and evaluation Local experiments are conducted on a local machine with a 8-core CPU and 20GB of RAM. As shown in Table 5.1, 8 experiments are conducted for local testing to compare the performance of standalone Vantage6, integration Scenario 1 and Scenario 2.

The experiments are performed on either mock client or real client with one or two nodes. During the task execution on each node, the main node container (where the task is posted to) will automatically generate a master container. The master container will control the flow of the algorithm execution. Based on the task input parameters (e.g. organization ids for executing the algorithm), an algorithm container will be generated for each participating organization/node to execute the algorithm. Both the server and nodes are hosted on the same local machine and they are running regardless of the existence of a FL task. A node does not compute the algorithm on the node container but its sub-containers, so the execution is mainly relying on the master container and algorithm container(s). Besides, the amount of CPU used by each vantage6 node (less than 4%) and the master container (less than 1%) is negligible. Thus, the performance of each experiment is evaluated based on the CPU and memory usage of the algorithm container and the memory usage of the master container. Each one-node experiment has one master container and one algorithm container. The performance is evaluated based on the maximum amount of CPU and memory that can be used by the corresponding containers during the task execution. For experiments with two nodes as shown in Table 5.7 and Table 5.9, each experiment has one master container and two algorithm containers (for each node). The resulting maximum CPU or memory usage for the algorithm container is measured based on the algorithm container that has the largest usage.

Remote testing setup and evaluation Three machines are used for the remote testing. Each machine has a 8-core CPU. The Vantage6 server and node 1 are installed in two virtual machines (located in the research lab) where each has 16GB of RAM. Node 2 is installed on the machine with 20GB of RAM, which is the same machine used in local testing. Mock client is only designed for local environment, thus, the experiments for testing Vantage6 on mock client is not included in remote testing. Besides, due to limited time and possible limitations of Vantage6 and Brane (e.g. possible undetectable task directory for

node container), integration scenario 2 is only implemented to fit local testing and further implementation on VMs is not included in the current work. Thus, the experiments on the VMs are only designed for standalone Vantage6 and integration Scenario 1.

Similar to local testing, node 1 and node 2 are configured with the corresponding dataset based on the FL use case. Both the server and nodes should be configured and started on the corresponding machines before the experiments. Node 1 and node 2 will automatically connect to server based on the server IP address. After a task is posted to the server, node 1 and node 2 will receive the task and start computation. The one-node experiments are conducted on node 1 where a master container and an algorithm container will be generated for running every task. The two-node experiments are performed on both node 1 and node 2, where the node 1 is the main node that has the master container. Each node will start a algorithm container for executing the FL task. The performance evaluation in remote testing is the same as local testing.

In the following sections, we introduce the experiment results along with some discussions.

5.1 Evaluation Results of the FL Algorithm

In this section, we discuss the experiment results for the evaluation of the FL algorithm FedAvg on two use cases: FedAvg MNIST and FedAvg Sentiment. Each experiment is performed on mock client and evaluated on one or two nodes. Each node has 35,000 data samples. The results are described in the following.

FedAvg MNIST As shown in Table 5.2, the model accuracy is increasing steadily as the number of epochs increase. Meanwhile, the two-node experiments show clear increase of around 2-3% on model accuracy compared to the one-node experiments. It indicates that averaging model parameters from multiple nodes that hold different datasets is an effective way in FL to improve model performance.

FedAvg Sentiment Table 5.3 shows similar growth pattern to FedAvg MNIST in model accuracy when the number of epochs/nodes increases. However, the general model accuracy for FedAvg is substantially lower (e.g around 20% lower with the same number of nodes and epochs). With the same number of epochs but different NN model and dataset, the relatively low accuracy on FedAvg Sentiment could mean that the Bi-LSTM model converges more slowly on the sentiment140 dataset compared with FedAvg MNIST with

2D-CNN. The accuracy score also decreases slightly when training using two nodes with 20 epochs. The reason for the decreased accuracy score could be related to not well-tuned training parameters or unsuitable optimizer for the dataset.

Based on the experiment results, the FedAvg algorithm is effective on both use cases and the model accuracy can be improved by increasing the number of nodes and epochs in general.

NO. of Epochs	No. of Nodes	Model Accuracy
2	1	89.38%
2	2	91.51%
10	1	94.28%
10	2	97.89%
20	1	95.84%
20	2	98.70%

Table 5.2: Model Accuracy for FedAvg MNIST

NO. of Epochs	No. of Nodes	Model Accuracy
2	1	64.79%
2	2	67.92%
10	1	68.17%
10	2	70.83%
20	1	70.63%
20	2	69.67%

Table 5.3: Model Accuracy for FedAvg Sentiment

5.2 Evaluation Results of the Integration and Vantage6

In this section, we describe the experiment results of the proposed integration. We include the following experiments

- Local testing
  - Standalone Vantage6 on mock client
  - Standalone Vantage6 on real client
  - Integration scenario 1 on real client
  - Integration scenario 2 on real client
- Remote testing
  - Standalone Vantage6 on real client
  - Integration scenario 1 on real client

As mentioned before, the evaluation for the performance of stand-alone Vantage6 is included as the baseline for comparison. Each experiment is performed by constructing a FL pipeline with the two FL use cases separately. The details are shown in the following.

5.2.1 Local Testing

For the local experiment results of both integration scenarios and standalone Vantage6, there is no major difference in performance for executing the same FL use case except the running time. As we can see in Table 5.4, Table 5.6 and Table 5.7, the time used for running FedAvg MNIST on two nodes is around 1.5 times larger than what takes to run the same use case on one node. When using FedAvg Sentiment for evaluation (Table 5.5, Table 5.8 and Table 5.9), the time difference between on-node and two-node experiments is even larger. Besides, the CPU usage for each node is slightly lower in general for two-node experiments compared with one-node experiments. As for the difference of using the two FL use cases, FedAvg MNIST usually takes more memory (e.g. around 900MiB) but less running time (e.g. 1m for one node and 2m for two nodes). However, FedAvg sentiment requires less memory (e.g. around 300MiB) but significantly more running time (e.g. around 9m for one node and 18m for two nodes). Moreover, since mock client do not use real node containers for calculation, the time required for starting containers are saved for mock-client experiments. Thus, the experiments on mock client are usually faster than the ones on real client.

Overall, the evaluation results between Vantage6 and integration (including Scenario 1 and Scenario 2) for local testing are similar when running the same FL use case and the results can vary based on the use case.

5.2.2 Remote Testing

For remote testing experiments with FedAvg MNIST (Table 5.10 and Table 5.11), the evaluation results is similar to the results in local testing in terms of running time and memory usage. However, the CPU usage is considerably different (e.g. around 400% for local testing and 800% for remote testing) because of different CPU specifications between the VM and local machine. In addition, as shown in Table 5.12 and Table 5.13, the FedAvg Sentiment experiments in remote testing is significantly slower than the local testing when using the same FL use case. Moreover, when running FedAvg Sentiment on node 1 alone (Table 5.12), the running time is similar to the performance of running with two nodes. However, when running the same task on node 2 alone (same as local testing as shown

in Table 5.8), the time it takes for the execution is a lot shorter than running on Node 1 alone. The results indicate that the the machine used by node 1 is slower than the one used by node 2 when executing heavy tasks. Meanwhile, the running time for two-node experiments is largely affected by the slower machine. Moreover, in terms of evaluation comparison between integration scenario 1 and Vantage6, there exits no major performance difference in remote testing when executing the same task under the same experimental setup.

To summarize, the evaluation results of different integration scenarios and Vantage6 demonstrate comparable performance in terms of running time, CPU and memory usage under the same experimental setup (e.g. equal number of nodes and same FL task). Nevertheless, the performance for a certain integration or Vantage6 can still be affected by the choice of FL algorithms/use cases and the system configurations of the host machines.

Number of Node(s)	Running Time	Max CPU Usage/ Limit	Max Memory Usage/ Limit
One node	59s	396%/800%	1.09GiB/ 19.42GiB
Two nodes	1m41s	400%/800%	1.13GiB/ 19.42GiB

Table 5.4: Vantage6 Experiment Results of FedAvg MNIST on Mock Client

Number of Node(s)	Running Time	Max CPU Usage/Limit	Max Memory Usage/ Limit
One node	7m41s	400%/800%	0.33GiB/ 19.42GiB
Two nodes	17m19s	400%/800%	0.37GiB/ 19.42GiB

Table 5.5: Vantage6 Experiment Results of FedAvg Sentiment on Mock Client

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	1m29s	401%/800%	863MiB/19.42GiB	517MiB/ 19.42GiB
Integration scenario 1 performance evaluation	1m16s	402%/800%	859MiB/19.42GiB	506MiB/ 19.42GiB
Integration scenario 2 performance evaluation	1m11s	407%/800%	828MiB/19.42GiB	506MiB/ 19.42GiB

Table 5.6: Local Testing(one node): FedAvg MNIST Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	2m24s	374%/800%	890MiB/19.42GiB	508MiB/ 19.42GiB
Integration scenario 1 performance evaluation	1m56s	382%/800%	865MiB/19.42GiB	506MiB/ 19.42GiB
Integration scenario 2 performance evaluation	2m04s	387%/800%	922MiB/19.42GiB	505MiB/ 19.42GiB

Table 5.7: Local Testing(two nodes): FedAvg MNIST Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	9m35s	406%/800%	285MiB/19.42GiB	185MiB/ 19.42GiB
Integration scenario 1 performance evaluation	9m18s	403%/800%	284MiB/19.42GiB	185MiB/ 19.42GiB
Integration scenario 2 performance evaluation	8m44s	406%/800%	281MiB/19.42GiB	215MiB/ 19.42GiB

Table 5.8: Local Testing(one node): FedAvg Sentiment Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	19m3s	386%/800%	287MiB/19.42GiB	232MiB/ 19.42GiB
Integration scenario 1 performance evaluation	17m13s	391%/800%	279MiB/19.42GiB	231MiB/ 19.42GiB
Integration scenario 2 performance evaluation	17m23s	403%/800%	282MiB/19.42GiB	231MiB/ 19.42GiB

Table 5.9: Local Testing(two nodes): FedAvg Sentiment Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	1m6s	787%/800%	965MiB/15.67GiB	522MiB/ 15.67GiB
Integration scenario 1 performance evaluation	1m1s	777%/800%	933MiB/15.67GiB	520MiB/ 15.67GiB

Table 5.10: Remote Testing(one node): FedAvg MNIST Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	1m54s	784%/800%(node 1) & 404%/800%(node 2)	904MiB/15.67GiB(node 1) & 906MiB/19.42GiB(node 2)	517MiB/ 15.67GiB
Integration scenario 1 performance evaluation	2m5s	786%/800%(node 1) & 403%/800%(node 2)	936MiB/15.67GiB(node 1) & 860MiB/19.42GiB(node 2)	524MiB/ 15.67GiB

Table 5.11: Remote Testing(two nodes): FedAvg MNIST Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	19m44s	689%/800%	387MiB/15.67GiB	195MiB/ 15.67GiB
Integration scenario 1 performance evaluation	19m32s	698%/800%	374MiB/15.67GiB	200MiB/ 15.67GiB

Table 5.12: Remote Testing(one node): FedAvg Sentiment Experiment Results

Experiment	Running Time	Max CPU usage (Algorithm Container)/Limit	Max Memory Usage (Algorithm Container)/Limit	Max Memory Usage (Master Container)/Limit
Vantage6 performance evaluation (real client)	21m	703%/800%(node 1) & 405%/800%(node 2)	380MiB/15.67GiB(node 1) & 274MiB/19.42GiB(node 2)	246MiB/ 15.67GiB
Integration scenario 1 performance evaluation	20m58s	709%/800%(node 1) & 404%/800%(node 2)	342MiB/15.67GiB(node 1) & 277MiB/19.42GiB(node 2)	252MiB/ 15.67GiB

Table 5.13: Remote Testing(two nodes): FedAvg Sentiment Experiment Results

6

Discussion

In this chapter, we show:

- Benefits and shortcomings of the proposed integration
- Possible future research directions for further integration
- Other FL algorithms that can be combined with Vantage6

6.1 Discussion of the Proposed Integration

In this section, we make comparisons between standalone Vantage6 and the integration in terms of the performance of running a FL task and the ease of use. The shortcomings and benefits of the integration are summarized as follows.

Benefits According to the experiment results in Chapter 5, the integration of Brane and Vantage6 can be used for implementing a FL workflow efficiently. The performance difference between the integration and stand-alone Vantage6 is minimal, proving no additional overhead is needed for running FL tasks in the integration. Besides, integration Scenario 2 includes all the necessary Vantage6 functions inside Brane. It allows Brane users to use FL services without setting up Vantage6 separately. Additionally, the pre-built functions are easily callable in the repl due to the user-friendly programming model in Brane. A user only needs to provide function input parameters in the repl instead of implementing a Python script with relatively more complicated Vantage6 client functions. Lastly, Brane repl also provide users an interactive platform that allows real-time monitoring for each step in the FL pipeline.

Shortcomings Though Brane functions are user-friendly, additional steps are required for the initial setup and task posting in the integration. To post a task using Vantage6 Python client, the user can simply run a Python script that has the user authentication and task information. However, the integration Scenario 1 is implemented based on the Vantage6 API functions as described in Chapter 4. Thus, the user needs to install the function packages in Brane first and input commands manually in the repl for each API function. For integration Scenario 2, Vantage6 commands are not needed, instead, server and nodes are started in the repl by using the pre-built Brane packages. The process of posting a task in Scenario 2 is similar to Scenario 1. Therefore, additional time for entering commands to call API functions are needed in both integration scenarios compared to using Vantage6 Python client.

Overall, the integration is user-friendly and effective for implementing a FL pipeline using Vantage6 functions in Brane with negligible overhead.

6.2 Further Integration

Based on the current integration, we discuss the possibilities to further deploy Vantage6 functions in Brane.

For instance, the integration Scenario 2 is currently only supporting local testing and it is possible to apply it for multi-node testing on multiple machines. However, during local testing, there are many issues regarding running tasks on node containers (details explained in Appendix (8)). They are caused by certain limitations of Vantage6 and Brane. These problems in Scenario 2 are solved only based on a local environment with a specific local directory and a static localhost IP address, which might not be suitable in a multi-node environment. To make it work, we could try using similar approach in a multi-node environment and modify the installed Vantage6 node package and Brane package to solve related issues if necessary.

Though the light integration has similar performance to standalone Vantage6, it still requires users to create the FL algorithm using Vantage6 client functions outside of Brane. Meanwhile, the Vantage6 API is not fully implemented in the current version which requires more steps to construct a FL pipeline than the Vantage6 Python Client (as described in Chapter 4), thus , it introduces overhead. To make it possible for users to create a FL algorithms and control its flow easier in Brane, we can implement the deep integration which does not make use of the Vantage6 API functions like the proposed light integration but builds Vantage6 client functions inside Brane. To implement deep integration, similar

to the Vantage6 node and server package in Scenario 2, it is possible to build Vantage6 client functions as packages in Brane as well. Inside the original *Vantage6-client* package, there are multiple small functions that control the implementation and workflow of the FL algorithm. To make this package available in Brane, we can divide the whole package into individual functions and build each function using the Brane Python function builder. Due to the limitation of the current version of Vantage6 (e.g. no user control on the communication when creating a FL algorithm which might be needed for building related functions in Brane) and limited time, the deep integration is not included in the current work. However, there is going to be better communication control in the future release of Vantage6, thus, the deep integration would be possible in the future work.

6.3 Other FL Algorithms

In this thesis, we implement FedAvg algorithm with two ML use cases to test the integration. There also exist many other FL workflows that can be implemented in Vantage6. For instance, VERTIGO (17) is a FL algorithm that uses logistic regression on vertically partitioned data, which can be implemented using Vantage6. The key idea of how to realize VERTIGO locally is briefly demonstrated on Vantage6 GitHub[1] but not fully implemented, thus, it would be possible to implement the full algorithm locally following the demonstration and even extend it for remote testing. Another FL algorithm that can be implemented in Vantage6 is Krum (18). Krum is a federated fusion algorithm that optimizes the original federated averaging with a proposed aggregation rule. It allows computation of good unbiased gradient estimates and is resilient to Byzantine (arbitrary behaviours) attacks. To implement it in Vantage6, we can modify the implemented FedAvg algorithm and add the gradient aggregation rule to the parameter averaging process. Besides, with the future release of Vantage6 (version 3) that will allow using external libraries and node-to-node communication control, it will be easier to implement various FL workflows in combination with other FL libraries (e.g. PySyft[2]) in the future.

[1] https://github.com/IKNL/vertigo
[2] https://github.com/OpenMined/PySyft

7

Conclusion

To conclude, in this thesis, we proposed an integration of Brane and Vantage6 under two scenarios and evaluate the integration by implementing a FL pipeline with FedAvg algorithm and two ML use cases. The experiment results demonstrate the effectiveness of the integration with insignificant additional overhead compared to standalone Vantage6. The answers to our research questions are summarized in the following paragraphs.

Regarding how to implement a ML pipeline using Brane and Vantage6 in RQ1, we proposed the light integration of the two systems and implemented a FL pipeline for the integration. The light integration was based on Vantage6 API function calls. We considered two scenarios for the integration, including scenario 1 with pre-installed Vantage6 infrastructure and scenario 2 that allowed users to build Vantage6 server and node as packages in Brane. Both scenarios were evaluated using a FL pipeline with the standalone Vantage6 as the baseline. We performed the experiments on local machine and also VMs, where the evaluation metrics for the performance included running time, CPU usage and RAM usage. As for more low-level integration, we do no include it in the current work but we introduce the possibility of implementing deep integration with the support of the future version of Vantage6 in Chapter 6.

RQ2 concerns the applicability and reusability of the proposed integration. To answer it, we implemented a FedAvg algorithm with two ML use cases and evaluated the integration in both use cases. The experiment results show that the integration can be applied to both use cases successfully with slight performance difference based on the dataset and NN model. It indicates that our proposed light integration is applicable to federated machine learning and can be reusable in different ML us cases.

8

Appendix

Vantage6 and Brane are still new research projects that are under active deployment. Thus, many issues/errors could exist on the current version of Vantage6 or Brane. In the Appendix, we introduce the issues we encountered during implementation and experiment.

8.1 Implementation issues

General issues Though Vantage6 documentation was nicely structured, there are sometimes missing/outdated details leading to technical errors. For instance, when importing entities to server, the "vserver import" command on the documentation did not work and an extra "–drop-all" flag was needed for the database tables being successfully built in the server. Besides, for posting a FL task in iPython shell, the documented format for task specification was outdated. Additional parameters, such as "id" and "input", should be be specified under "organizations" for a task to be posted on server successfully. Due to the limitation of the current version, Vantage6 docker wrapper that handles communication between nodes and the algorithm only supports CSV files and database files (with .db extension) as local data files on the nodes. In order to use the aforementioned datasets (e.g. MNIST and Sentiment140), I chose to work with CSV data files in both use cases and both datasets are partitioned horizontally in the nodes. Besides, current version of Vantage6 only supports one file as database for each node. Thus, each node contained only one CSV file with both training data and testing data during experiments. An extra step was added during the execution of the algorithm to use only 70% of the data for training while the rest was for testing. Moreover, there are certain functions that are built in the Python client but not available in the Vantage6 API, including input data type conversion. The data type conversion of task input parameters is required for the success of posting a task.

Thus, an input conversion function was implemented to convert the input parameters into JSON serializable format before adding it to the task information. Besides, the results for a task should be decoded into human-readable format, which is another function that is not available in iPython shell. The input-conversion and results-decoding functions should be made accessible in Brane for the integration. Thus, both of them was first implemented using Python script. Then, the functions were customized with descriptions of all the meta data, files and dependencies explicitly in a configuration YAML file. The brane package builder was used to install each function using a configuration YAML file in Brane as a self-contained Docker image. After successful installation, these python functions became accessible in brane repl. The required JSON string in input-conversion function does not allow single quotes, but double quoted string is not supported in Brane repl. Thus, the string was specified using single quotes in the repl first and an extra step was added to the function to convert it to double-quoted string. Lastly, for building the API functions in Brane, additional support for function input parameters as well data types (e.g. array and nested object) was added to Brane binary package.

Issues specific to Scenario 2 As for the integration Scenario 2, the server package was built using an older Python version to make it run successfully. Besides, when a Vantage6 node is started, it needs to find the specified task directory based on the *configuration.yml* to create a Docker data volume to execute a task. The node container creates sub-containers when executing a task and the task directory path to the sub-container is an auto-detected and fixed path in Vantage6. Since the node container is created by Brane, its sub-containers do not detect the task directory inside Brane and thus, require the task directory to be specified before starting starting the node in Brane. So we specified the path to the task directory in the node *configuration.yml* and *entrypoint.sh* files specifically to be the same as brane data directory for its existence to be detected. We also made the required local dataset available in the Brane data directory for the node container to use. In addition, a proxy server is automatically started once a task is received by a running node. However, when building the node container in Brane, the proxy server cannot detect the host IP address. Thus, the proxy server IP was manually specified in the entrypoint.sh based on the host IP of the local machine. Since Vantage6 only setup containers automatically when the server and nodes are configured using the regular Vantage6 commands, the proxy port that a node needs (by default 8080) in Scenario 2 was occupied, leading to the failure of starting the node. Thus, the proxy port number was reset to be randomly generated in a range where there was no occupied port. To import entities using a YAML

file in Scenario 2, an older version of Vantage6 server docker image was needed while the actual server should be running on the new version. Thus, both server images were installed in Brane while the older version was kept in a virtual environment for importing entities only. The socket.io error was another error appeared, leading to the failure of a node. It was solved by adding the correct dependency versions of SocketIO to the node *container.yml*. The details of *configuration.yml*, *entrypoint.sh* and *container.yml*for both server and node are shown in Figure 8.1, Figure 8.2, Figure 8.3, Figure 8.4, Figure 8.5 and Figure 8.6 correspondingly.

```yaml
application: {}
environments:
  acc: {}
  dev: {}
  prod:
    allow_drop_all: 'True'
    api_path: /api
    description: Vantage6
    ip: 0.0.0.0
    logging:
      backup_count: 5
      datefmt: '%Y-%m-%d %H:%M:%S'
      file: vantage6.log
      format: '%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s'
      level: DEBUG
      max_size: 1024
      use_console: true
    port: 'PLC_SERVER_PORT'
    uri: sqlite:///default.sqlite
  test: {}
```

Figure 8.1: The server *configuration.yml* in integration scenario 2

8.2 Experiment issues

When testing the real Vantage6 client locally, the internal local IP address was unable to be detected on the local machine (Linux) by the node container because the "localhost" was bind to the server. This problem was resolved on the Vantage6 side by resolving the localhost to the actual IP address (we re-installed Vantage6 using this branch: git+https://github.com/iknl/vantage6.git@DEV). During the remote testing, an server attribute error was encountered when after node 2 finished task execution and tried to connect to server to register the results. It only happened when the newest server image was used. To make the experiments run successfully, I chose to specify an older server docker image (2.1.1) when starting the server and the error no longer exists. Another error that was related to Vantage6 was duplicate node networks with the same name. Such error existed because the same node was started before with the same node network name and

the network was not removed properly when the node was stopped. It was solved by using a docker command "docker prune" to clear the duplicated networks.

```yaml
application:
  directories:
    data: /
  api_key: PLC_API_KEY
  api_path: /api
  databases:
    default: PLC_DATA_PATH
  encryption:
    enabled: false
    private_key: ''
  logging:
    backup_count: 5
    datefmt: '%Y-%m-%d %H:%M:%S'
    file: v6.log
    format: '%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s'
    level: DEBUG
    max_size: 1024
    use_console: true
  port: "PLC_SERVER_PORT"
  server_url: http://PLC_SERVER_HOST
  task_dir: /usr/local/share/vantage6/node/v6
environments:
  acc: {}
  dev: {}
  prod: {}
  test: {}
```

Figure 8.2: The node *configuration.yml* in integration scenario 2

```bash
#!/usr/bin/env bash
set -euo pipefail

sed -i "s/PLC_SERVER_PORT/$SERVER_PORT/g" configuration.yml

# Run import inside virtual environment with Vantage6 v1.2.3 installed.
pipenv run vserver-local import --config ./configuration.yml --drop-all fixtures.yml

# Disable encryption for existing collaborations.
pipenv run python -c "
from vantage6.cli.context import ServerContext;
from vantage6.server.model.base import Database;
ctx = ServerContext.from_external_config_file('./configuration.yml', 'prod', True);
Database().connect(ctx.get_database_uri());
import vantage6.server.db as db;
for c in db.Collaboration.get(): c.encrypted = False; c.save();
"

# Start the server.
vserver-local start --config "./configuration.yml"

echo "~~>output: done"
exit 0
```

Figure 8.3: The server *entrypoint.sh* in integration scenario 2

```
#!/usr/bin/env bash
set -euo pipefail

export LC_ALL=C.UTF-8
export LANG=C.UTF-8

sed -i "s/PLC_API_KEY/$API_KEY/g" configuration.yml
#sed -i "s/PLC_DATA_PATH/$DATA_PATH/g" configuration.yml
sed -i "s/PLC_DATA_PATH/${DATA_PATH//\//\\/}/g" configuration.yml
sed -i "s/PLC_SERVER_HOST/$SERVER_HOST/g" configuration.yml
sed -i "s/PLC_SERVER_PORT/$SERVER_PORT/g" configuration.yml

# Generate random port number for the proxy.
export PROXY_SERVER_PORT=$(shuf -i 2000-65000 -n 1)
#set the data volume to be the same for brane and v6
export DATA_VOLUME_NAME=brane_data
#set v6 proxy server to be the same as host IP
export PROXY_SERVER_HOST='192.168.0.157'

vnode-local start --config "./configuration.yml"

echo "~~>output: done"
exit 0
```

Figure 8.4: The node *entrypoint.sh* in integration scenario 2

```yaml
name: v6_server
version: 1.0.0
kind: compute

base: ubuntu:18.04
docker: true

entrypoint:
  kind: service
  exec: entrypoint.sh

dependencies:
  - python3
  - python3-pip

environment:
  LC_ALL: C.UTF-8
  LANG: C.UTF-8

install:
  - pip3 install pipenv
  - pipenv install vantage6-server==1.2.3
  - |-
    pip3 install \
      Flask-SocketIO==4.2.1 \
      Flask==1.1.1  \
      PyJWT==1.7.1 \
      python-engineio==3.10.0 \
      python-socketio==4.4.0 \
      socketIO-client==0.7.2 \
      vantage6-server==2.1.1 \
      websocket-client==0.57.0
files:
  - configuration.yml
  - entrypoint.sh
  - fixtures.yml

actions:
  start_server:
    command:
      args: []
      capture: prefixed

    input:
      - name: server_port
        type: integer

    output:
      - name: output
        type: string

```

Figure 8.5: The server *container.yml* in integration scenario 2

```yaml
name: v6_node
version: 1.0.0
kind: compute

base: ubuntu:18.04
docker: true

entrypoint:
  kind: service
  exec: entrypoint.sh

dependencies:
  - python3
  - python3-pip

install:
  - |-
    pip3 install \
      Flask-SocketIO==4.2.1 \
      Flask==1.1.1  \
      PyJWT==1.7.1 \
      python-engineio==3.10.0 \
      python-socketio==4.4.0 \
      socketIO-client==0.7.2 \
      vantage6-node==2.1.0 \
      websocket-client==0.57.0

files:
  - entrypoint.sh
  - configuration.yml

actions:
  start_node:
    command:
      args: []
      capture: prefixed

    input:
      - name: api_key
        type: string

      - name: data_path
        type: string

      - name: server_host
        type: string

      - name: server_port
        type: integer

    output:
      - name: output
        type: string

```

Figure 8.6: The node *container.yml* in integration scenario 2

References

[1] Peter Kairouz et al. **Advances and Open Problems in Federated Learning**. *Foundations and Trends in Machine Learning*, **14**(1–2):1–210, 2021. [Online]. Available: http://dx.doi.org/10.1561/2200000083. iii, 5, 6

[2] Arturo Moncada-Torres, Frank Martin, Melle Sieswerda, Johan van Soest, and Gijs Geleijnse. **VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange**. In *AMIA Annual Symposium Proceedings*, pages 870–877, 2020. iii, 1, 8

[3] **Docker: Empowering App Development for Developers**. `https://www.docker.com/`. iii, 10, 11

[4] Chen Zhang, Yu Xie, Hang Bai, Bin Yu, Weihong Li, and Yuan Gao. **A survey on federated learning**. *Knowledge-Based Systems*, **216**:106775, 2021. [Online]. Available: https://doi.org/10.1016/j.knosys.2021.106775. 1, 5, 6, 7

[5] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. **Communication-Efficient Learning of Deep Networks from Decentralized Data**. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017. arXiv:1602.05629. 1, 7

[6] **The Brane Framework**. `https://docs.brane-framework.org/`. 1, 9

[7] Qinbin Li, Zeyi Wen, Zhaomin Wu, Sixu Hu, Naibo Wang, Yuan Li, Xu Liu, and Bingsheng He. **A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection**. 2019. arXiv:1907.09693. 6

[8] Deng Ting, H. Hamdan, K. A. Kasmiran, and R. Yaakob. **Federated Learning Optimization Techniques for Non-IID Data: A Review**. *International*

Journal of Advanced Research in Engineering and Technology (IJARET), **11**:1315–1329, 2020. [Online]. Available: http://dx.doi.org/10.34218/IJARET.11.12.2020.125. 6

[9] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. **SCAFFOLD: Stochastic Controlled Averaging for Federated Learning**. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, **119** of *Proceedings of Machine Learning Research*, pages 5132–5143. PMLR, 13–18 Jul 2020. [Online]. Available: https://proceedings.mlr.press/v119/karimireddy20a.html. 6

[10] Dinh C. Nguyen, Ming Ding, Pubudu N. Pathirana, Aruna Seneviratne, Jun Li, and H. Vincent Poor. **Federated Learning for Internet of Things: A Comprehensive Survey**. *IEEE Communications Surveys & Tutorials*, page 1–1, 2021. [Online]. Available: http://dx.doi.org/10.1109/COMST.2021.3075439. 6

[11] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. **Practical Secure Aggregation for Privacy-Preserving Machine Learning**. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1175–1191, New York, NY, USA, 2017. Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/3133956.3133982. 7

[12] Jed Mills, Jia Hu, and Geyong Min. **Communication-Efficient Federated Learning for Wireless Edge Intelligence in IoT**. *IEEE Internet of Things Journal*, **7**(7):5986–5994, 2020. [Online]. Available: https://doi.org/10.1109/JIOT.2019.2956615. 7

[13] **REST API Tutorial**. `https://restfulapi.net/`. 10

[14] **OpenAPI Specification v3.1.0**. `https://spec.openapis.org/oas/v3.1.0`. 10, 23, 27

[15] **Vantage6**. `https://docs.vantage6.ai/`. 13, 20, 22, 29

[16] Alec Go, Richa Bhayani, and Lei Huang. **Twitter sentiment classification using distant supervision**. *Processing*, **150**:1–6, 01 2009. [Online]. Available: http://www.stanford.edu/ alecmgo/papers/TwitterDistantSupervision09.pdf. 30

[17] Yong Li, Xiaoqian Jiang, Shuang Wang, Hongkai Xiong, and Lucila Ohno-Machado. **VERTIcal Grid lOgistic regression (VERTIGO)**. *Journal of the American Medical Informatics Association*, **23**, 11 2015. [Online]. Available: http://dx.doi.org/10.1093/jamia/ocv146. 45

[18] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. **Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent**. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 118–128, Red Hook, NY, USA, 2017. Curran Associates Inc. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf. 45