

Building a cloud-based testing platform for Federated Learning

David-Marian Buzatu — 2174320

Supervisor: Shuo Wang

B.Sc. Computer Science

Word count: 9978

April 8, 2022

Contents

Contents	1
1 Introduction	3
1.1 Project aim	3
1.2 Background	3
1.3 Related work	5
2 Technical background	7
2.1 Training methodologies and architectures in ML	7
2.2 Google Cloud Platform	8
2.2.1 Compute Engine	8
2.3 Terraform	9
2.4 PyTorch	10
2.5 Flask	10
2.6 Postman	10
2.7 NodeJS	11
2.7.1 ExpressJS	11
2.8 Database	11
3 Design	13
3.1 Key objectives	13
3.1.1 Key high-level functionalities	13
3.1.2 Key high-level non-functionalities	13
3.1.3 Constraints and assumptions	15
3.2 Development plan	15
3.2.1 Detailed functional requirements	16
3.2.2 Detailed non-functional requirements	16
4 Implementation	19
4.1 Architecture	19
4.2 Components	21
4.2.1 Instance	22
4.2.2 Main Controller	24
4.2.3 Back-end server	25
4.2.4 Front-end	28
5 Testing and appraisal	30
5.1 Appraisal	30
6 Conclusion	35
6.1 Future additions	35
7 Bibliography	36
A GitHub repository	39

B	NNLayerFactory	40
C	NNConvolutionLayerFactory	41
D	Example neural network JSON	42
E	Example training parameters JSON	43
F	Modals	44
G	Environment states update	47
H	List of Acronyms	48

Chapter 1

Introduction

This project provides a cloud-based platform to perform Federated Learning (FL) algorithms testing within a distributed infrastructure with built-in simulation capabilities. Moreover, the solution is made available as a Web Application with a Graphical User Interface (GUI) that makes the interaction code-free, and enables testing to be targeting real-life situations, rather than coding.

1.1 Project aim

The aim of this project is to ease and improve the work with FL algorithms by automating the process required for testing FL. FL has been introduced in [30] as an alternative learning approach where privacy of data is insured while training Neural Network (NN) models. Although there have been numerous performance and architectural improvements in FL, a prevalent issue is the lengthy process of developing architecture for testing such algorithms. This task is rather complex and brings a lot of difficulty as one needs to share data according to specific distributions, simulate hardware and software constraints, while also building Neural Networks (NNs) and training them in a distributed environment. Therefore, our goal is to simplify this process by removing the necessity of writing code, having the option of simulating devices with various configurations, and be able to train models in a FL setting by interacting with a GUI.

1.2 Background

The term Artificial Intelligence (AI) has emerged more than 60 years ago, when John McCarthy hosted the first academic conference on the subject. Since then, AI has undergone periods of ignorance through the 1970s, but has since become one of the most studied and groundbreaking subject in the field of Computer Science. Its applications are limitless, ranging from improving search engines and building recommendation systems, to facial and image recognition, just to name a few. AI has thus been fervidly studied by researchers, and its popularity has just been increasing [27].

NNs have initially captivated the interest of researchers as a means of simulating human-like neurons with the use of electric circuits [28]. By associating them with the human neural system, they have been modelled to behave in a similar way - i.e., signals are passed from neuron to neuron, which in turn process the input and forward the resulting output. In artificial NNs, electrical signals are replaced by linear and non-linear activation functions that 'turn-on' specific neurons based on the associated weights between each connection. Thus, bits of information are learnt by sets of neurons, resulting in complex connections and structures that are able to learn patterns from information.

The discovery of Back-propagation [42] opened up the possibility of learning models that had a tremendously higher accuracy and could work for more complex problems. With the accomplishments achieved in the studies of Geoffrey et al. [16, 17] Deep Neural Networks (DNNs) have been first introduced, and demonstrated state-of-the-art learning capabilities at the cost of more computational resources, time for training and amount of data needed.

Ever since, **DNNs** have enabled outstanding advancements in areas such as computer vision, recommendation systems, speech recognition, data generation and more. Moreover, improvements in computational power, data availability and software engineering [5] have also been essential to making such developments, and have simultaneously evolved to meet the required industry demands.

However, this ever increasing usage of **DNNs** to solve problems in our lives came with other issues. For example, with deeper networks, the memory and computational complexity increase significantly, getting to the point where hundreds of machines are required to learn and store a model. Furthermore, the availability and quantity of data has brought up issues pertaining to its usage due to privacy concerns. An article [15] has shown the implications of data usage, where a teenager was identified as being pregnant solely using their online shopping behaviour. Therefore, moving raw information to a centralised server may result in immediate or latent information leakage [24].

As such, an interest in finding ways for learning in a privacy-friendly environment has received momentum. Advances in cloud-computing have proposed learning models in a distributed way, where numerous machines work simultaneously on the same learning task, and result in an aggregated model [8]. This way, the need of constructing single machines capable of learning has been mitigated, and a faster, more accurate and scalable environment for training models has emerged. Nonetheless, the issue of privacy of data is still a concern in distributed machine learning due to the collection of data in a centralised place, with direct access to the underlying data.

The aforementioned developments have made possible the coining of **FL** as a distributed learning algorithm that keeps data private. Introduced in [30], **FL** allows learning a shared model in a collaborative mobile environment, where each device keeps their data locally, and only share the learnt model with a centralised server that aggregates the collection of models - see Figure 1.1. Moreover, **FL** enables learning smarter models using lower latency and less power, while also ensuring privacy of data [30]. Data privacy is not the only issue that **FL** tries to solve. It is also expected to handle Non independent

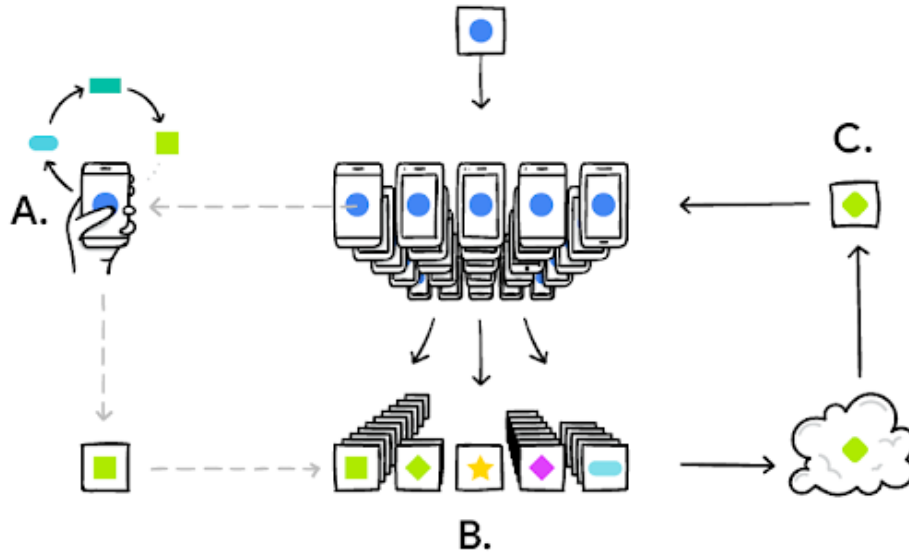


Figure 1.1: Source: [31]. The mobile device adapts the model to its local data (A). It then sends the model to a centralised server that aggregates all the models received (B) to form a new, generalised model (C) that gets shared as the ground-truth model for all devices.

and identically distributed (**non-i.i.d**) data and space features that vary from device to device [29, 24]. This can happen in cases where data varies across regions, or information is gathered from multiple organisations. Finally, **FL** takes advantage of security measures, such as encryption, to ensure privacy of data during transmission and processing, whereas other classical approaches neglect this aspect [24].

The learning process involves multiple learning rounds. In each round, devices are elected by a centralised entity based on their availability. In general, availability is achieved by accomplishing three requirements:

- the device is idle
- it is charging
- has a wi-fi connection [30].

We adopt the notation from [47] to further explain **FL** and provide mathematical insights into our approach. Thus, we define N as the number of data owners, and we note the data owners as the set $\{F_1, F_2, \dots, F_N\}$. Each data owner F_i wants to train the model using its local data D_i . In a **FL** approach, the users train a model M_{FED} where the underlying data D_i for a user F_i is hidden from others. Finally, we note the accuracy of the model as V_{FED} .

The concept of **FL** is further categorised into three main areas, based on the type of data used [47]:

- **Horizontal FL**: datasets share the same feature space, but different sample space. For example, two regional banks may have different users, as it is unlikely to have the same users frequenting both banks, however the type of data that is collected is similar, thus the feature space is akin [47].
- **Vertical FL**: the datasets share the same sample ID space, but differ in feature space. This is the case when users are common to two companies, but the features of the data they entail are different. For example, a local grocery store and a bank may share a vast amount of users, while the data each of them collects is comprised of different things. Thus, an aggregation of the features is done by creating a model that is learnt from both datasets, while preserving each party’s private data.
- **Federated Transfer Learning (Hybrid FL)**: used when both user and feature space are different. A common representation of the two feature spaces is learnt using limited sample sets, which can then be applied to make predictions for samples with only one type of feature.

In our work, we have targeted **Horizontal FL**, however our approach may be adapted to other categories of **FL**.

1.3 Related work

In the extensive study on the currently open issues in **FL**, the authors of [19] identify numerous concerns. Platform and deployment challenges are thoroughly studied, and it is stated that **FL** is different than centralised training due to:

1. Automated testing not being able to cover the whole spectrum of heterogeneous devices
2. Monitorisation and debugging being difficult or impossible due to no physical access to devices
3. Performance and stability of devices not being affected by training.

Monitorisation of key metrics such as crash rates, memory usage, latency and other app specific metrics is difficult to be tested, as it requires code deployment to physical devices and building architecture that allows these interactions with the physical devices. It can take weeks to roll out - i.e. to deploy - a **FL** model because of the issues that might arise with each device.

In the same study, the difficulty of reproducing problems in a controlled environment is also mentioned as an undergoing issue in **FL**. Induced biases are difficult to identify and react to, thus having

the option of testing for such cases is essential.

Finally, system parameter tuning is an open issue in FL, as this type of learning has to handle:

- convergence speed
- throughput - e.g., number of rounds and devices, amount of data
- model fairness, privacy, robustness
- resource usage on server and client.

Tension between the parameters make it hard to find the right combination. For example, throughput increase might introduce bias by preferring highly performing devices with little or no data. Maximising low loss by increasing complexity hurts less performing devices. Moreover, such biases are hard to detect because the evaluation phase will use the same type of devices to validate the model [19].

Therefore, we have identified that building a testing platform for FL will provide researchers and developers with an innovative, time-efficient and easily usable system to perform preparations for future deployments of FL algorithms. However, we note that this process is difficult due to the required flexibility of working with any learning task, any type of NN, any type of dataset, and any type of device. Our solution does not cover all cases, and therefore aims at providing a minimal approach that covers image datasets, Convolutional Neural Networks (CNNs) and simulating devices on the cloud.

In the work of Tim Krask et al. [22], a novel way of performing Machine Learning (ML) tasks in a distributed environment has been coined. The proposed solution helps users by providing a declarative language for performing ML tasks, making working with ML available to non-expert users, or saves development time for advanced researchers. Moreover, the system offers ML solutions in a readily-scalable environment, further easing the work with sophisticated distributed algorithms. Our solution was inspired by their approach to ease working with distributed ML, and we took the idea of parsing information about data and algorithms in a declarative language and adapted it to JavaScript Object Notation (JSON).

Numerous studies tackle the issue of distribution of data in FL environments. Some propose novel ways of handling distribution by changing the underlying learning implementation [33], while others provide insights into federated averaging algorithms [29]. Thus, we placed specific emphasis on the available options for data distribution and federated averaging algorithms. Our aim is to provide a flexible mechanism for testing different distributions of data for a given learning task, while also providing distinct averaging algorithms.

While there are plenty of open issues related to security and communication in FL and solutions are proposed [19, 21, 29, 47], we do not place emphasis on these particular topics. Our solution is, in its current stage, aimed at providing testing environments for working with FL algorithms with the focus on data distribution and environment simulation capabilities.

In terms of the architecture adopted, we have adhered to a centralised FL approach. Although studies of edge-based FL [46] and client-edge-cloud architecture [25] have shown different architectures that learn high accuracy models with reduced communication and computational cost, we chose not to follow such approaches due to limitations in the available resources and development time.

Chapter 2

Technical background

In this chapter we introduce the learning process in [ML](#) tasks on which our solution is based on. We also include different libraries, cloud providers and frameworks used throughout our approach. Only the crucial technologies that are key in our implementation are described here. However, other tools are mentioned in the description of our implementation - see [Chapter 4](#).

2.1 Training methodologies and architectures in ML

The process of [ML](#) is capable of identifying patterns in data such that models can be built and used in a variety of tasks [\[9\]](#). [ML](#) models are usually represented as a graph structure composed of processing units and parameters; together with hyper-parameters, the model learns the structure of the data [\[24\]](#). Using inputs, the model generates an output based on the type of task it was trained for: classification, prediction, generation, etc.

From the different processes of [ML](#), we have focused our implementation on supervised learning. In this type of learning, the training data is comprised of features and labels. During the training process, the model learns to predict outputs similar to the labels. We have narrowed down our implementation to supervised learning tasks due to the type of data we are able to handle - i.e., images - and the types of models we are able to create - i.e., [DNNs](#) created using PyTorch [\[36\]](#).

By [\[1\]](#), an [ML](#) implementation can be done in four stages:

1. Data pre-processing: to perform learning, the task has to be identified and datasets prepared for processing during training.
2. Network architecture: a network architecture has to be chosen for the task such that it is able to perform as expected. There are numerous architectures that can be chosen - e.g., [CNNs](#) for working with image recognition - and experimentation has to be done to find the right fit.
3. Model training: After the data has been processed and the model's network constructed, the model is trained using machine learning frameworks - in our case, PyTorch.
4. Model testing: The resulting model is tested on unseen data to measure performance.

Although these stages still apply when using huge amounts of data, classical training approaches on single machines are not viable anymore, as the quantity of data may be too big, or distributed on clusters [\[45\]](#). Distributed [ML](#) accelerates the training process and accommodates for datasets that are huge or distributed across clusters of servers. In a distributed learning environment, data is split across various machines, which individually perform training using their data partition, and a pipeline combines the results from the different machines into a single model [\[24\]](#).

Although distributed [ML](#) solves the issue of working with terabytes of data and makes the usage of resources more efficient, it does not solve the issue of privacy of data. Despite that in a distributed environment data is not centralised, the working servers still have direct access to the underlying data

that has been collected.

With FL [31], the data that is used during training and inference is kept on the participant’s device, thus the model can be trained collaboratively in a distributed environment, while also keeping the data private from a centralised server [1, 4] - see Figure 2.1 for comparison. This mechanism works by training models locally on each device, sending the resulting models to a centralised server, and then aggregating them to form a general model.

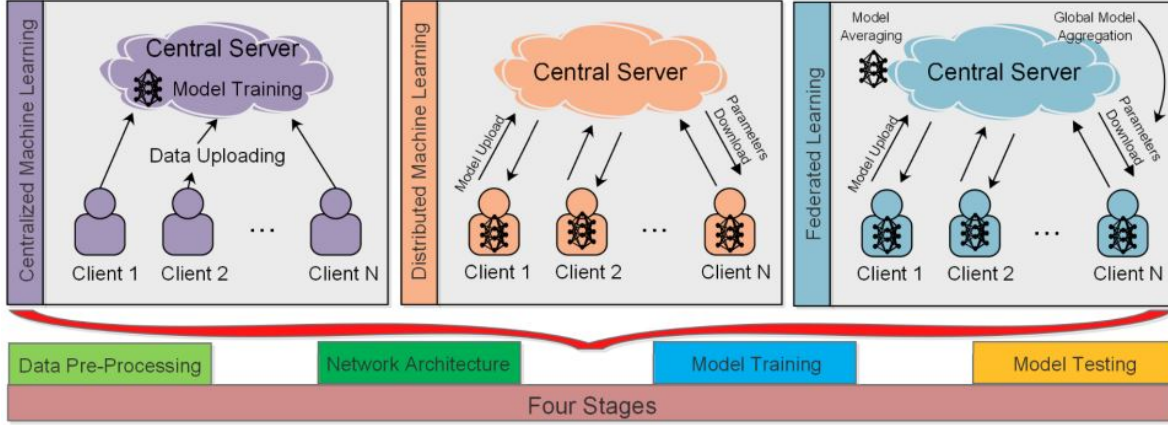


Figure 2.1: Source: [1]. In the figure above, Muhammad Asad et al. describe three types of model classifiers. Centralised ML is the classical approach, where data is collected and trained on a centralised server. In the distributed ML approach, training is done using the same methodology as the Centralised ML, except models are trained on the clients in a synchronised approach and the central server averages the received models. In Federated Learning, clients are independent of each other and perform their training with local parameters. The central server computes the aggregate of the models and sends the next global model for further training or inference on each client.

2.2 Google Cloud Platform

Google Cloud Platform (GCP) is a suite of cloud computing services that Google uses internally and that are available for use by end-users [12]. Their approach to cloud-based services enables companies and individuals to integrate their services into the cloud with as much flexibility as needed in managing networking, firewalls, data, logging and computing instances. They also provide advanced data analytics services and machine learning capabilities, however our attention was focused on cloud development infrastructure using the Compute Engine (CE) service.

We have decided to use GCP during development due to its lower cost, flexibility in integrating clusters of machines, ease of access to the networking services used by the clusters, and the selection of available boot images for CE instances.

2.2.1 Compute Engine

The CE is a service that enables the creation of Virtual Machines (VMs). Compared to other providers, we found a greater variety of machine configurations, available images, ease of use, and lower pricing.

CE instances can be built to accommodate various needs in terms of computation, high memory usage, general purpose machines, scale-out workloads, and Graphics Processing Unit (GPU) intensive applications - e.g., ML tasks - [10]. Our solution allows users to select the type of configuration they need, thus they can aim for more processing power or memory for each simulation.

We have created our own boot images - i.e., systems that run docker containers with the instance code - to be used by [CE](#) instances to further improve the performance of our solution. In terms of networking, we used the default networks available in the eu-west region, and made changes to the firewall to allow connections to the machines.

2.3 Terraform

It is an Infrastructure as Code ([IaC](#)) tool that allows building, maintaining and versioning infrastructure [14]. Using Terraform's high-level configuration language it is possible to describe the infrastructure using humanly-readable code. Terraform builds an execution plan before applying changes and requires approval.

The infrastructure code can be utilised by making use of the Terraform Command Line Interface ([CLI](#)) that provides a suite of commands to plan, create and destroy environments. The typical workflow for working with Terraform is:

1. Write - Create infrastructure as code. This is done using the configuration language and defines the providers, resources and variables to be used during the build process.
2. Plan - Before building any infrastructure, Terraform allows to see the results of applying the build command.
3. Apply - Produce the requested infrastructure. This step connects to cloud providers and builds the requested infrastructure.
4. Destroy - If the infrastructure needs changes, or it is malfunctioning, the destroy command allows to remove the infrastructure and all of its resources.

The main purpose of the language is to write resources which represent infrastructure objects. They are given a resource type - e.g., **google_compute_instance** - and a name - e.g., **instances**. Inside the resource block, there are configuration arguments needed for the given resource type:

EXAMPLE FROM OUR IMPLEMENTATION

```

# resource type          # name
resource "google_compute_instance" "instances" {
  count          = var.nr_instances # number of instances
  name           = "instance-${var.user_id}${count.index}" # name of instances
  machine_type   = var.machine_type # type of machine to use

  boot_disk {
    initialize_params {
      image = var.instance_image # what image to use during creation
    }
  }
  network_interface {
    network = "default"
    access_config {
      # necessary to allow external connection
    }
  }
}
```

Blocks are containers for content. They can have zero or more labels and are comprised of a body which can have an arbitrary number of arguments and nested blocks.

```

<BLOCK TYPE> <BLOCK LABEL> <BLOCK LABEL> {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Arguments assign a value to a name. Expressions represent a value, either by reference or combination.

A module is a collection of **.tf** and **.tf.json** files kept together in a directory.

Our approach uses Terraform to build the instances used to train **FL** models, and it is also used to create the infrastructure for the controllers of the instances.

2.4 PyTorch

PyTorch [36] is an open-source framework for working with Deep Learning [20]. Compared to other options, PyTorch has been chosen due to its workflow with Python, its flexibility and ease in creating models, and overall performance. It works with tensors - i.e., multidimensional rectangular arrays of numbers - that can be accelerated via **GPU**, and provides an automatic differentiation system to support **DNNs**.

In our approach, we have abstracted away the construction of models by creating Factory classes that use the PyTorch's **nn** module to construct models based on the given options. The **nn** module contains the building blocks for **NNs** in PyTorch, and it allows working with a variety of layers, including, but not limited to Convolution, Pooling and Transformer.

2.5 Flask

Flask [35] is a micro web framework used to build web services in Python. It is a micro framework due to its minimal core that does not impose a database abstraction layer, form validation or authentication, but it allows developers to add such functionalities as needed by installing libraries.

We have extensively used Flask in our implementation, on both the instances and controllers. Due to its minimal core, creating a server that can listen and respond to Hypertext Transfer Protocol (**HTTP**) requests is done in less than ten lines of code, and leaves the developer the freedom to add complexity as needed. To create a dummy server that listens on the default port 5000, one can write the following five lines of code:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "Hello World"
```

The **@app.route** is a Python decorator used to inform Flask about available route paths in the **HTTP** server. The function defined below the decorator is used to handle the incoming request. Moreover, in the decorator developers can specify the allowed methods for the end-point - e.g., **GET** - to further limit the types of accepted requests.

2.6 Postman

Postman is an Application Programming Interface (**API**) platform for building and using Application Programming Interfaces (**APIs**) [37]. With Postman, developers can target **HTTP** routes in their application, store and catalogue requests, automate testing, document and mock **APIs** and even allow collaborations across organisations.

We have made use of Postman to create collections of **API** requests within our platform to perform end-to-end tests and to allow for tracking of changes.

2.7 NodeJS

NodeJS is a JavaScript asynchronous runtime for building scalable web applications [6]. NodeJS abstracts working in a concurrent environment through its internal engine. It helps developers by tackling the intricacies of concurrent requests handling, thus mitigating the issue of dead-lock processes, and improves the performance by reducing the number of I/O operations.

2.7.1 ExpressJS

However, to properly make use of NodeJS and its capabilities, we have used ExpressJS, which is an application framework for NodeJS that has been developed to build web applications and APIs [43]. We have used ExpressJS due to our previous experience with it, and also the necessity for working with asynchronous requests in our back-end server and proxies in our gateway.

2.8 Database

Databases are the core of any application. They provide the skeleton for the data used throughout the application, providing means of storing and querying the data. From the most prominent characteristics, databases focus on how to store the data, the speed of access, the flexibility of the data model, scalability and consistency.

There are two main types of databases: Structured Query Language (SQL) and Not only Structured Query Language (NoSQL) databases. SQL is a good fit for situations when data transactions are essential to be consistent. Thus, SQL provides documentation, simplicity and data integrity before anything else [13]. However, SQL was not developed with distribution in mind, thus due to its structure and complexity of querying, SQL is more fitted to scale vertically - i.e., by increasing a machine's resources - rather than horizontally - i.e., being distributed on clusters of machines.

NoSQL targets this exact horizontal scaling issue and flexibility in structuring the data. NoSQL was developed thinking about structural changes, scalability, distribution and ease of use from developers' perspectives [18]. In SQL databases, data is structured in a tabular way, with data points spreading on rows, having attributes stored as columns. On the other hand, NoSQL databases provide document, key-value, wide-column and graph storage to provide the aforementioned benefits.

For our project we have used MongoDB for a few reasons. Firstly, the project has been developed in an AGILE manner. Thus, feature changes have occurred as the project advanced, and there was a constant need of being flexible with how the data is stored and used. Secondly, the scope of the project is to be a standard cloud-based provider for testing FL algorithms, meaning that the architecture has to be scalable and provide options for distribution and constant growth of the stored data. Finally, the interaction with MongoDB is consistent among all the programming languages and libraries we have used, thus it eased the work with the database and accelerated development.

We have initially considered storing datasets in the database and serve them to instances as needed. This would have been beneficial in improving the performance of updating datasets, but most importantly it would have made sharing datasets among environments a matter of targeting the same database. Moreover, with the consent of users, datasets could be shared, the database providing a centralised space where datasets could reside and could be accessed by any user's instance.

However, we have identified a few crucial points that worked against this idea. Firstly, datasets are rarely going to change, resulting in a minor number of writes for this operation in the database. Secondly, reads may quickly become expensive. Storing datasets in the database will incur at least a request from each instance to get the corresponding dataset - we say at least because caching could be put in place to save datasets locally. However, caching also comes with a few issues:

- Caching datasets will introduce the need for cache invalidation.

- Having many instances targeting the database to get datasets, which have at least sizes of hundreds of Megabytes, will incur a bottleneck for the whole platform.
- Data will be replicated along each instance, thus incurring an increase in the memory used.
- Storing datasets in documents has to be formatted to handle any type of data, which comes with implementation difficulties and less flexibility to changes.

Finally, storing datasets into the database will also come with more security measures to protect datasets from being accessed by malicious users.

Therefore, we have decided to use local data storage for datasets. Datasets would be sent over the network to each designated instance, where they will be accessed through local access by the learning algorithms. In case of changes, a delete command will be issued to delete the dataset and a new one will be added to the instances.

Chapter 3

Design

Our approach to building a testing platform for FL algorithms targets the assumptions and conditions required for FL to be implemented. Firstly, as stated by [31], devices need to meet certain criteria to be used in a training session, i.e., they need to be charging, connected to a wi-fi connection and be idle. Secondly, the type of data on each device is *non-i.i.d.*, i.e., the feature space varies from device to device, and it cannot be known what kind of distribution of data may get to be used [1]. Finally, the amount of data available for each participant is different, thus there needs to be careful consideration as to how the data should be used [1].

Our approach targets these conditions through its functionalities, and provides a modern and state-of-the-art approach to testing FL algorithms. In this section, we present our solution’s functionalities, how it relates and uses aforementioned relevant research in its approach, and how we prioritised and developed our solution.

3.1 Key objectives

The core functionality of our approach can be summarised as follows:

- Key high-level functionalities - related to the core capabilities of the platform.
- Key high-level non-functionalities:
 - Minimising the required coding for working with FL algorithms.
 - Ease of accessibility in setting up a development environment for FL through a GUI.
 - Providing flexibility in testing FL algorithms in terms of the number of devices, the distribution of the dataset, the aggregation method used, and the performance and behaviour of devices.
- Constraints and assumptions.

3.1.1 Key high-level functionalities

Our solution must create environments for training NN models in a federated process. Moreover, the platform must allow users to change datasets used by each device, change training distribution of data and provide training logs. Finally, the platform must allow users to configure the hardware components of devices, set a failure chance for each instance, and construct a variety of NN models.

3.1.2 Key high-level non-functionalities

Minimising coding

Working with ML may be done with a library such as PyTorch. However, using libraries comes with the responsibility and time-consuming process of learning to write code. Thus, our approach aims to help those researchers that may not be as acquainted with writing code, but also the proficient users

advance their work faster by removing the need for coding.

Our solution was to encapsulate the majority of classes and functions from PyTorch into Factory classes that can abstract away the creational process and allow flexible implementations of **NNs** and training methods. We provided in Figure 3.1 the simplified class diagram for the NNModel class used in our approach.

Finally, to communicate the design choices for the networks, training, hyper-parameters and data distributions, we have made use of **JSON**. This allows our servers to inter-communicate efficiently and enables the web application to provide the information from the user input as a standard format that can be processed by all servers. Thus, users only have to provide their options using **GUI** elements - i.e., buttons or input fields - which are then converted into **JSON** objects and sent to servers to process them and create the architectures and training methods requested.

Ease of accessibility

An important factor in our approach is making working with **FL** accessible to all interested researchers, with or without coding knowledge. Thus, we adhered to use a **GUI** web application to allow communication with our servers. We have decided that a command line tool may be too ambiguous, and would make the interaction complicated, resulting in users finding themselves lost in the low-level interface.

Therefore, we built a web application using NextJS [38], a JavaScript framework for building user interfaces in React [32]. We have adhered to using NextJS due to its performance, accelerated development, ease of use and prior experience with React. NextJS helped us build an interface based on reusable components, and made it easy to set-up routes, caches and prepare deployment artefacts by its working nature.

We have abstracted complex interactions - e.g., creating an environment - by making use of input boxes, buttons and lists. We believe that our simplified approach allows for an easier interaction between users and environments, while still preserving the essential mechanisms of **FL**. We present our interface in 3.2 for creating environments.

Device simulation

Our approach focused on two of the issues identified in the study done by Kairouz et al. [19], which are building architectures for **FL** algorithms and real-life simulation of devices. In terms of the architectural development, our approach used cloud-based platforms to provide working devices, while also including modern approaches to building cloud-infrastructure with **IaC**. We have adhered to using automated infrastructure building tools to ease our development process, allow for collaboration and history tracking for the infrastructure - which is almost impossible to achieve without **IaC** tools - and to provide dynamic infrastructure to our users. Therefore, we have made our users focus on the development of **FL** algorithms and evaluation metrics, rather than researching cloud-based solutions, building distributed architectures and long periods of coding.

For simulating devices, we needed computational power, memory resources and networking capabilities. We currently use a Virtual Machine (**VM**) - i.e., an **CE** instance - to simulate a single device. Thus each individual machine can simulate particular behaviours without affecting other devices.

We adapt the notion of **instance** to denote a device in our system. Instances are the lowest level of processing in our architecture and have the tasks of receiving and handling dataset partitions, to build **NNs**, to train and share models, and to update their simulation parameters. The notion of **environment** denotes a set of instances pertaining to a user.

Environments encapsulate the necessary devices for testing a **FL** algorithm, and are independent of each other. Once an environment has been created, users can distribute data as intended and start the learning process. They can test how the distribution of data on each instance affects the outcome of the aggregated model and can change training parameters without any code interaction.

The specific parameters that can be changed for an instance are:

- data distribution
- probability of failure during training
- resources

Data distribution

Distributing data in [FL](#) tasks is essential. In numerous studies [4, 1, 47] data distribution has been studied and tackled with various solutions. There have been issues related to geographical localisation of data - i.e., data coming from different parts of the world -, unbalanced distribution of the data on devices, and prioritisation of devices such that only specific distributions get to influence the model training. With our approach, we want researchers to be able to test how their model is affected by aforementioned circumstances and allow for changing the distribution of data on instances to reflect these situations.

Probability of failure during training

As stated in [31], devices are considered eligible for a training round if they meet certain requirements. To simulate these requirements, we have implemented a mechanism for failing during training on each instance that is based on a given probability rate. Thus, we believe instances may reflect better the real-life chance of a device to stop meeting the training conditions during a training round.

Resources

Due to our reliance on [GCP](#) to host devices, we are able to allow specific configurations [11] for instances based on the available options from our provider. Therefore, we made it possible to configure instances with specific numbers of CPUs, GPUs and RAM to be able to reflect real-world device's performance.

3.1.3 Constraints and assumptions

Our approach has focused on supervised learning due to the intrinsic complexities that come with unsupervised and reinforcement learning, that may not work with our current model representation. Furthermore, we have constrained datasets to image data, as handling text datasets must be done with more control over the fields, data pre-processing and storage. Moreover, we have constrained our instances to run on [GCP](#) due to the cost implications, however we may support other providers as well. Finally, we have assumed that model evaluation will be done separately by our users, and thus our solution does not provide functionalities related to this.

3.2 Development plan

Our project was developed with an AGILE [3] mindset. We have developed our solution in an incremental process, where iterations helped us refactor parts of the code, improve the quality over time, and also enabled us to meet our goals. We started by aiming for a Minimum viable product (MVP) [40, 44] that would allow researchers to do the essential tasks with [FL](#). The importance of simulating real-life simulations in [FL](#) contexts is a tedious process that depends on the aforementioned functionalities to be working as expected.

In terms of prioritisation, we have started with researching our architectural approach, the technical stack that we will use, and our limitations. To achieve our MVP, we have first implemented the devices to handle [FL](#) tasks, then built the servers that control the process and the back-end server, constructed the [GUI](#) and only then began integrating real-life simulation. Because our approach was an iterative one, we have constantly went back to some previous components and improved our work, or added new functionality.

3.2.1 Detailed functional requirements

Due to our iterative approach, we have focused our functional requirements on basic needs that our system must meet as an MVP. The following functionalities were crucial in our approach:

- the system must be able to create environments based on user needs.
- the system must be able to train a model in a FL approach using a user environment.
- the system must allow users to create and manage accounts.
- the system must allow users to manage their environments.
- the system must allow users to update the data distribution on instances.
- the instances must be configurable.
- the instances must be independent of each other.
- the system must be able to destroy environments.
- the system must provide a GUI to interact with the environments.

3.2.2 Detailed non-functional requirements

Our non-functional requirements targeted the interaction between users and our system. We have not focused on providing innovative architectural solutions to FL training, such as the studies of [4, 25, 48] have shown, neither did we target to test the security of the data in FL tasks. Our approach focused on providing an interface for easily setting up testing environments for working with FL algorithms, and also allow for real-life testing of these methods.

Therefore, our non-functional requirements can be listed as:

- the users must be able to create environments only with button clicks and text inputs.
- the system must be able to handle at least 2 instances running in an environment.
- the NNs architectures must be diverse and allow for complex networks to be created.
- the system must handle errors during training without stopping execution.

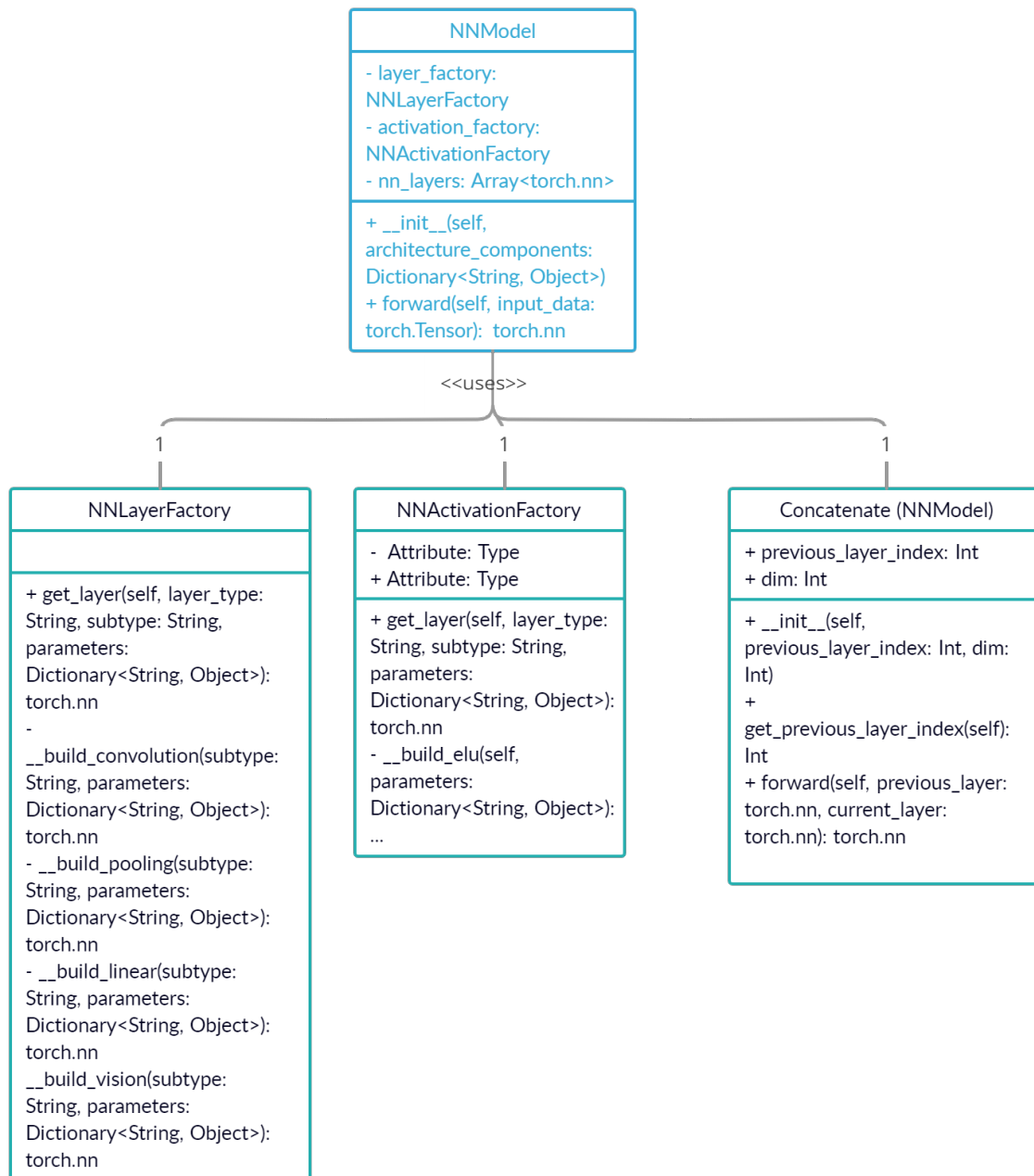


Figure 3.1: Class diagram for the NNModel class. The NNModel class is used to create the required [NN](#) architecture. It uses other Factory classes - i.e., LayerFactory and ActivationFactory - to build different types of layers and activations. We have omitted all the functions used and sub-classes due to space constraints.

Final Year Project

Environments

Datasets

Logout

Create environment

Machines configuration

Number of instances

1

GENERAL-PURPOSE

COMPUTE-OPTIMIZED

Machines series

e2

Please select your machine series

Machines type

e2-micro (2vCPU, 1GB memory)

Please select your machine type

Environment options

Instance number

Probability of failure

%

REMOVE

ADD OPTIONS

CREATE

CANCEL

Figure 3.2: Environment creation page. Users simply insert the number of instances, select their machine type and set instance specific parameters given an index. The index is used to randomly select an instance to send the specified probability of failure. After the process finishes, users have an environment ready to be used for training.

Chapter 4

Implementation

In this section, we elaborate on the intricacies of our implementations for the development of the system. We start with a thorough analysis of a viable architecture for our solution, then we continue to describe each component from the architecture, why it is necessary and how was it implemented.

To begin our implementation, we have primarily inspected the approaches adapted in [31, 48, 47, 4], and created system flows to further help us understand the needed architecture. As mentioned in Chapter 3, we adapt the notation of environment to denote a cluster of instances designed for a particular FL task, where instances represent devices. We now introduce the notion of **controllers**, which denote a set of working servers that are responsible to handle the interaction with environments.

In Figure 4.1 we show a simplified flow for creating an environment. Using Terraform [14] we create environments and once they have been instantiated, save their corresponding Internet Protocol (IP) addresses necessary for communication.

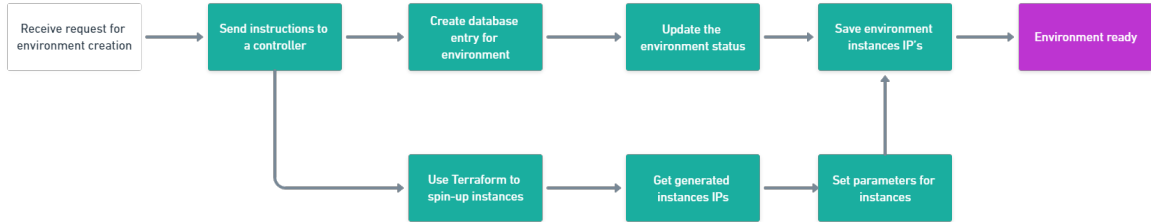


Figure 4.1: System flow for environment creation

Figure 4.2 represents the set of actions that can be used to update an environment. The ability to change an instance parameters - i.e., data distribution, probability of failure during training - without having to re-create the environment, allows us to provide complex testing capabilities in an accelerated and easier way. Finally, we also identified the need to have the option of changing the NN model used by instances.

4.1 Architecture

Based on the identified essential flows in our system, we further analysed various types of architectures that could accommodate these requirements. We present our thorough analysis of different solutions, what are their strong points and weaknesses, in our GitHub repository A. Therefore, we further de-

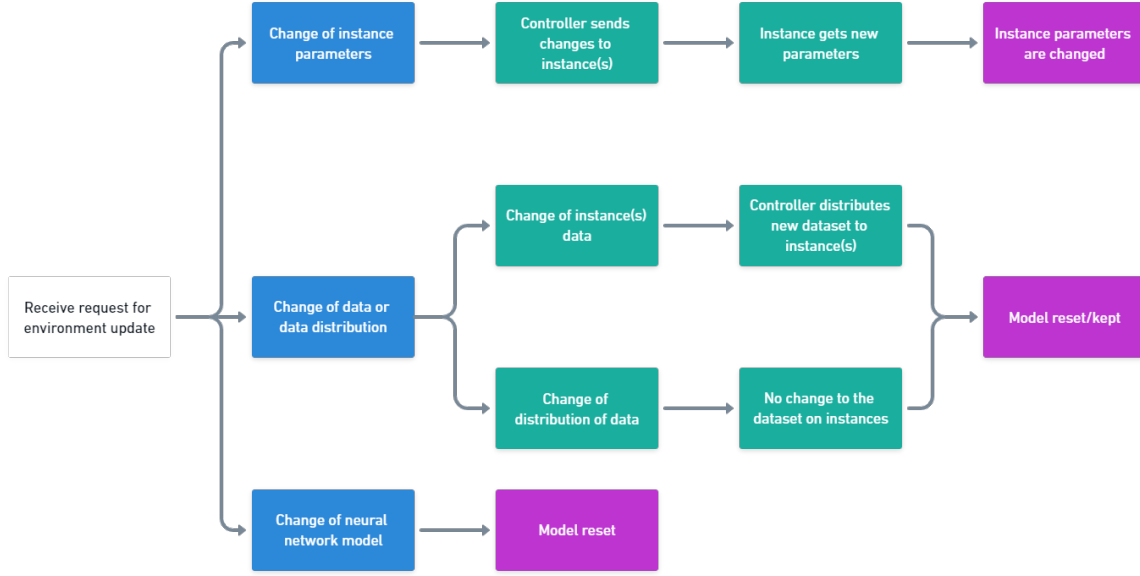


Figure 4.2: System flow for updating environments

scribe the final architecture.

Our ideas for the architecture took inspiration from the protocol endorsed in [4] and shown in Figure 4.3. Thus, we identified three main essential pieces of working servers:

- instance - represents a device in the protocol
- controller - represents the central server in the protocol
- back-end server - our own additional server needed to handle controllers and interactions with environments.

We have started our architecture from the intuition that users would be using our system to either interact with an environment for training purposes, or perform Create/Read/Update/Delete (CRUD) operations on them. Therefore, we have identified the need of having a back-end server that would bridge the user requests to the specific environments. However, we quickly realised that a back-end server would be overwhelmed with requests and may even crash or underperform if numerous training rounds would be carried out, as tens or even hundreds of HTTP requests would have to be kept open with each instance for the whole duration of the training process.

Furthermore, the back-end server would have too many responsibilities to handle if it would be the only component between users and environments. Thus, we have recognised the need of another component that sits between the back-end server and environments, and we called that **main controller**. With the controller in place, we have also found a better solution for encapsulating environments. As stated in our thorough analysis in A, by having a set of independent, stateless servers - i.e., main controllers - be responsible for the management of environments, we eliminated a single-point-of-failure in environments - i.e., a controller specific for that environment - while also improving the performance and scalability capabilities of the platform. Any main controller can pick a request and fulfil it without any prior connection to an environment. This approach resembles a Master-Slave approach, where the main controllers act as masters, and the environment's instances as slaves.

Once we introduced multiple controllers, we saw the need of balancing the load to them. We have made use of Nginx [34] because it is one of the industry standards for load balancing, it was easy to set-up and has a lot of built-in flexibility when it comes to how the load is balanced - i.e., it offers multiple algorithms for balancing the load. Whenever a command will be issued, a main controller

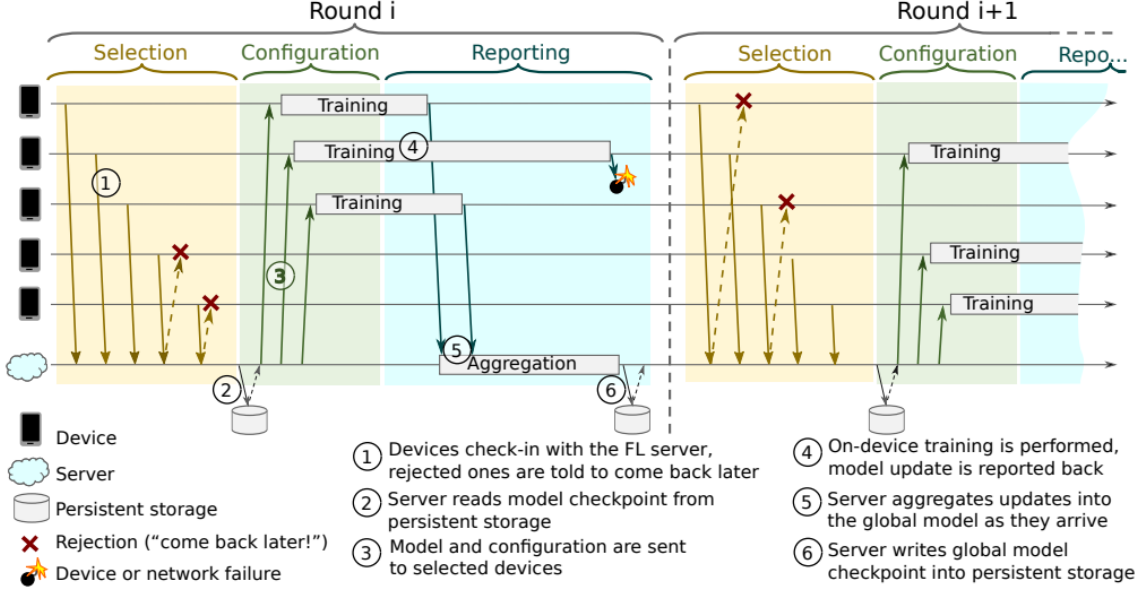


Figure 4.3: Source: [4]. In the protocol, devices share their availability with the server in the 'selection' window. Then, the server shares the common, aggregated model with each device and sends over the configurations for training in the 'configuration' window. During the 'reporting' stage, on-device training is performed and the model reported back to the centralised server. The central server aggregates the model and writes the global model to a persistent storage. These three stages repeat for as many rounds as specified.

will be selected to handle the request and will target the right environment.

Another important aspect we have focused on was user authentication. We had to choose between passing authentication tokens through each component, for every request, or create a service that could handle authentication for us, and ensure the rest of the components about authenticity. Therefore, we have added a Gateway component to our architecture that is responsible for authenticating incoming requests, and only allowing authenticated requests to pass. Moreover, we have implemented a reverse-proxy inside the gateway to forward the requests to the back-end server once authentication checks have passed.

Due to the blocking nature of HTTP requests, we identified an issue in the way requests would be handled by the back-end server. With blocking, requests for training would still lock the back-end server just as if the back-end would interact directly with environments. This is due to the request being waited to be fulfilled. We have thus changed the way the back-end works, and adhered to an event-driven architecture for it, which added another local database to our architecture - i.e., Redis. We will further explain this implementation in 4.2.3.

Finally, we have made use of MongoDB to store various data about interactions with the environment and user data. A complete picture of the architecture can be seen in Figure 4.4.

4.2 Components

We continue to elaborate on each of the components of the architecture, with the mention that we do not describe each available endpoint on our servers - as that can be seen in our repository A - but focus on the intricacies of each piece, and arguing for the decisions made during development. We also

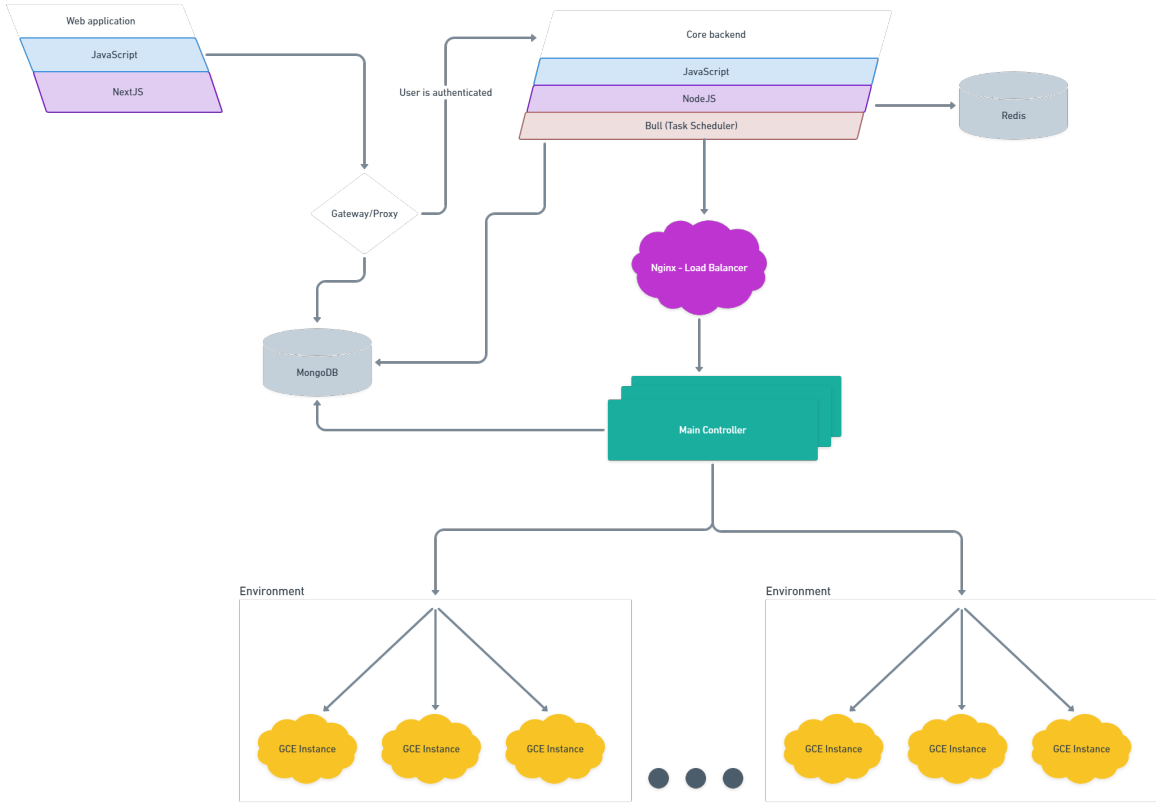


Figure 4.4: Architecture high-level view

do not describe the gateway, load-balancer and database as they have been previously discussed.

4.2.1 Instance

An instance functionality is represented by the diagram in Figure 4.5. As stated before, an instance is equivalent with a device in general FL terminology. Instances can best be described as computational resources that can be configured for specific learning tasks, and consequently have no interaction with the database and are independent from the rest of the architecture. Therefore, the instance is delegated to perform operations on NN models, update the data that is used for various model operations, and also simulate real-life devices through instance operations - e.g., setting the probability of failure.

To fulfil these requirements, we decided to use Python for the instances and rely on HTTP protocols to interact with them. The use of Python was essential, as the language has a plethora of libraries and frameworks that work best with performing ML tasks. Moreover, using Python enabled us to use Flask [35] as a web framework to handle HTTP requests.

Performing updates on the dataset is a matter of using system calls to delete a folder or a set of files, which could have been done in other programming languages as well. Accordingly we do not emphasise this aspect.

The most crucial and difficult aspects that the instance has to fulfil are the creation and training of a model. To allow users to provide JSON inputs that describe the type of network to be built and the types of loss and optimisation functions to be used during training, we have adhered to using the **Abstract Factory** and **Factory method** design patterns. This allowed us to be dynamic with the objects created at runtime, while also allowing new classes to be instantiated with the addition of a new method inside a Factory, or by adding a new Factory class to our family of factories.

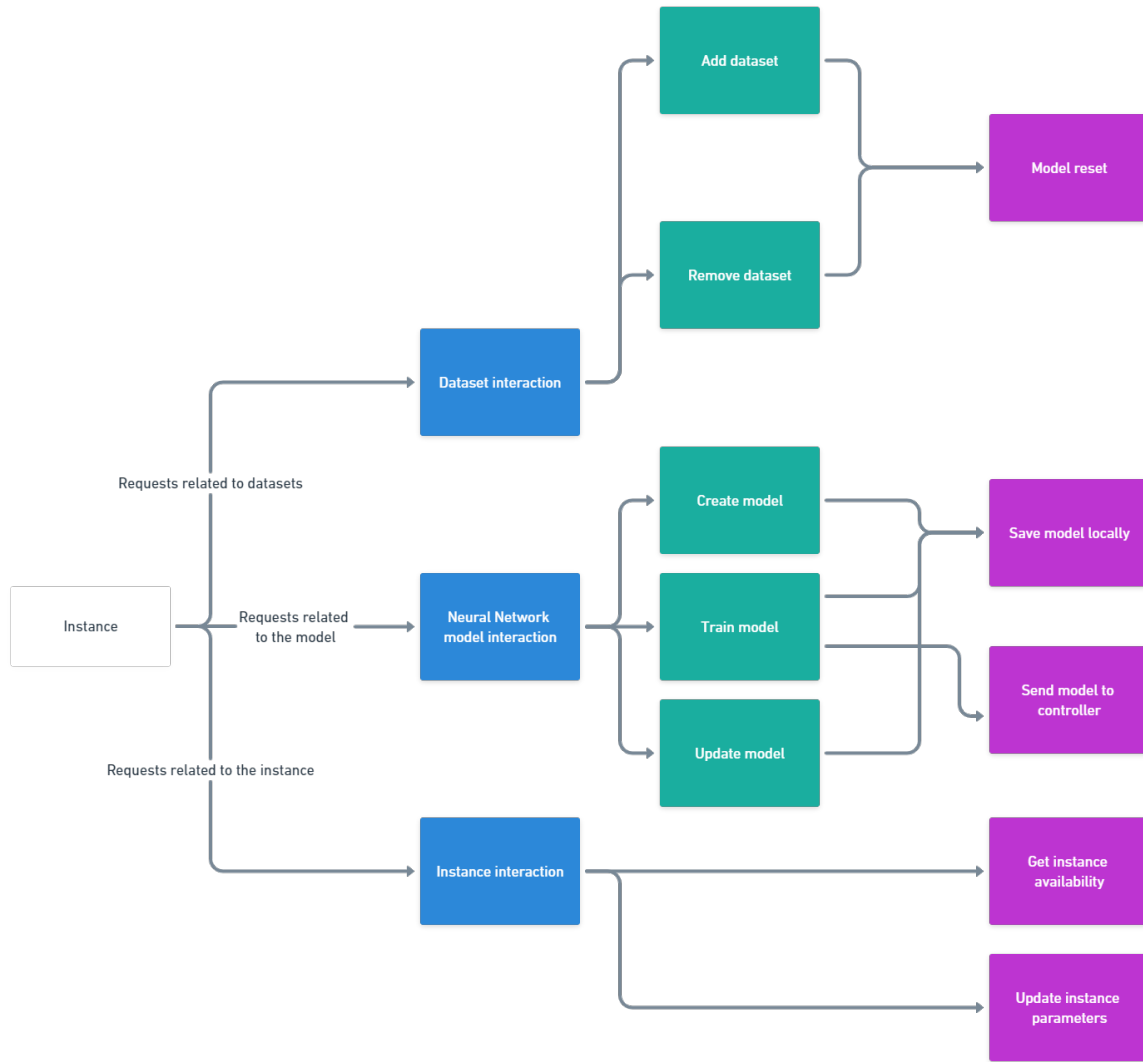


Figure 4.5: System flow for instances

As shown in Figure 4.6, a factory class has a getter method that can be used to build a specific type of class from that factory. To instantiate classes, the factory has ‘`__build`’ methods that are specialised for a particular case - e.g., convolution layers. However, due to the complexity of networks, we needed to add factory classes for different categories of layers: convolution, pooling, linear and vision. Thus, the **NNLayerFactory** class became an Abstract Factory that uses Factory classes to build **NN** layers.

One can find a code snippet for the **NNLayerFactory** in [B](#) and **NNConvolutionLayerFactory** in [C](#) to understand how those classes are built. We also mention that whenever a class cannot be created due to its type not being recognised, or its list of parameters being insufficient, our factories are able to handle the error cases by throwing exceptions.

This design choice allows instances to receive **JSON** strings that can be parsed to create the necessary classes. An example of such **JSON** is given in [D](#). The same style applies to training parameters as well, where other factory classes use **JSON** strings to build the optimisers and losses by matching on specific PyTorch class names, and looking for sets of parameters pertaining to that class. An example of a **JSON** used for training parameters is given in [E](#).

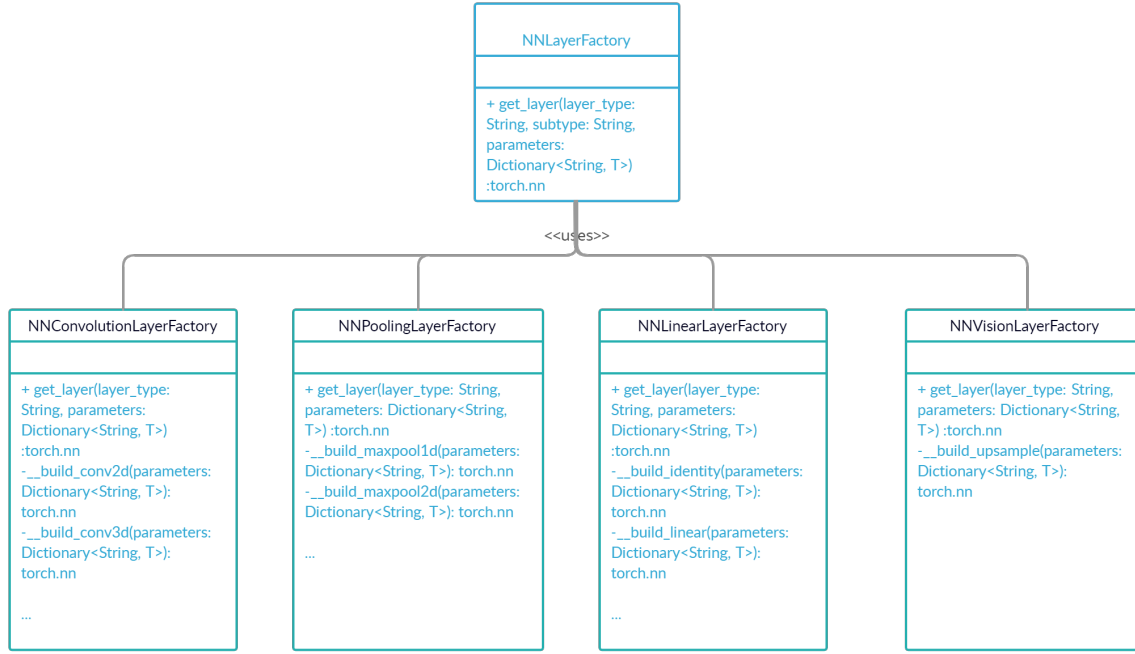


Figure 4.6: Class diagram for layer factory. We have omitted the majority of build methods due to space constraints and all sharing the same function arguments.

When it comes to performing the [FL](#) task, instances provide their availability to the main controller, which then decides which instance to use for the training rounds. Instances will be given training parameters by the main controller and start training. During a training round, on each epoch, using its probability of failure, the instance checks if it should stop training - i.e., simulate failure - by drawing a random integer from the $[0, 100]$ range, and checking if it is greater than the probability of failure. Once their learnt model was saved locally, it will be shared with the main controller at end of each complete training round on the instance.

The main controller is responsible to aggregate the models from all instances, and will issue a model update on each instance, which will result in overwriting the local model with the aggregated one. This process repeats for as many rounds as the main controller has been instructed to do. We believe that this type of handling for the training rounds reflect real-life situations closely, while also preserving the functionality of [FL](#).

4.2.2 Main Controller

The main controller is our adaptation for the centralised server, with the addition that any controller can work with any environment. Controllers are the only components that interact with the instances and control their life-cycle.

Due to controllers having the role of the centralised server in [FL](#) settings, they have also been implemented using Python and Flask. This was mandatory, as aggregation of models is dependent on the library used to compute the models on the instances. Unfortunately, this also meant that a copy of the factory classes had to be present on the controllers, as PyTorch requires these classes to be able to load and update a model.

The endpoints in the controllers are available in [A](#). As previously mentioned, controllers are stateless, which allows us to assign any controller to any task related to environments. We continue our discussion on how the controller creates environments and performs the training, and consider that

transmission of datasets and parameters to the instances are evident, as they only pass the requests and files to the target environment.

Creating environments

Before constructing the **VMs**, the controller creates an entry in the MongoDB database for the environment to be created. This insertion asserts the status of the environment to be 'creating', thus the front-end application is capable of showing progress. Afterwards, it prepares a payload - i.e., a set of objects - to be used by Terraform to build infrastructure as requested.

This payload contains data related to the number of instances, type of machines to use, and user information. Terraform uses these to spin-up **VMs** using **CE** instances dynamically. Behind the scenes, Terraform uses **gcloud** commands to interact with Google Cloud (**GC**), and abstracts away the long list of commands it issues on the behalf of the owner of the **GC** project.

Once instances have been built, the controller saves their **IP** addresses in the database, and continues to send the instance specific configurations to each instance. When instances have been configured, the state is updated to 'Created' and the process finishes.

Training models

For the process of training models - see Figure 4.7 -, the controller first parses the **JSON** parameters for training. Following the protocol explained in 4.3, the controller selects the suite of instances to be used in the training round by making a request to all instances in an environment and getting their availability. Training can stop here if the process exceeds a maximum number of re-trials or insufficient devices are found.

Next, the controller instructs each instance to perform training. Due to the number of instances, we made the requests asynchronous such that multiple devices can train at the same time, and we can maximise the utilisation of the controller. We have used the concept of **futures** [7] from the standard Python library.

Once all instances responded - either with success or failure - the controller continues to aggregate the models, or stops the training if all instances have failed. Our current implementation always performs an average of the received models. However, our approach may easily change the type of aggregation by sending the aggregation type in the **JSON** for training. Once the model is aggregated and saved locally, it is shared with each device.

This process repeats for the given number of rounds, or until a critical error - e.g., no devices are available - occurs. The final model is sent back to the back-end server, which we continue to discuss next.

4.2.3 Back-end server

The back-end server has been developed to fulfil the need of a coordinator inside our architecture. As previously mentioned in 4.1, due to the blocking period resulting from training a model, we have decided to implement this server with an event-driven architecture.

Event-driven architectures are based on the concept of tasks and executional queues. Whenever a task is created, it is added on the task queue and waits until its execution order comes. If the task cannot be finished in the given period, its state will be saved, and added back to the queue to be processed in the future. This process is similar to how Central processing unit (**CPU**)s handle processes, and thus is not trivial to be implemented. Therefore, we have studied available libraries in JavaScript and Python, and decided to use JavaScript due to the library's simplicity, flexibility in handling tasks, and prior experience.

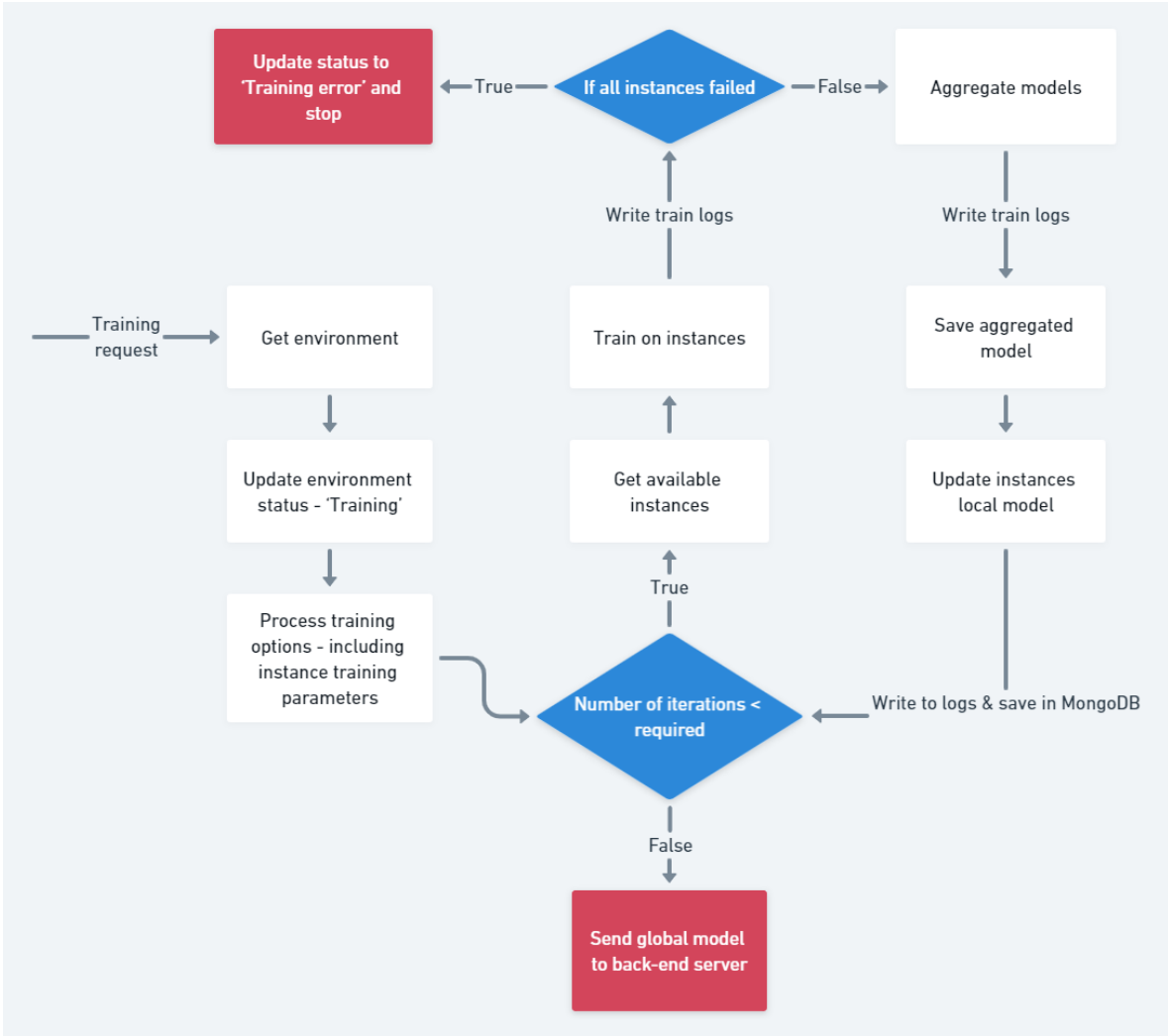


Figure 4.7: Training process handling for Main Controller. We have omitted the database object from the diagram for simplicity.

Our back-end server is built using Express [43] and NodeJS [6], which we detailed in 2.7. The back-end is responsible for two types of interactions:

- Requests targeting environments.
- Requests targeting MongoDB.

Requests targeting environments

We have made use of BullJS [2] as a task handler due to its performance and ease of use. Using BullJS required setting up Redis - a key-value pair database [26] - to store the task queue and its data.

When a request pertaining to environments interaction arrives at the back-end - see Figure 4.8 -, a task is created with the request encapsulated in it, it then gets added to the Redis queue, and a 202 Accepted response code is returned to inform the client that the request has been accepted and will be executed sometime in the future. Together with the 202 response code, an endpoint link is sent corresponding to the specific task id that has been created, such that clients can use the endpoint to check the status of the task.

After a task has been added to the queue, BullJS takes care of scheduling, executing and saving

its state. We implemented the handlers, called workers, that given specific task queues, perform requests and handle errors in the appropriate way.

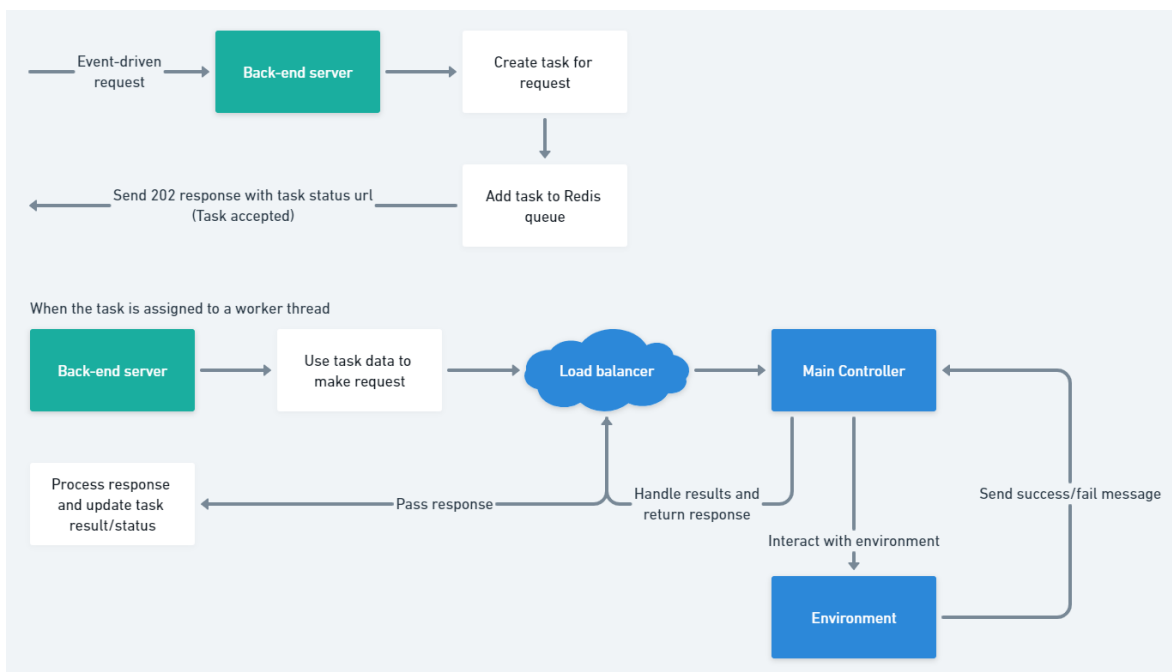


Figure 4.8: Event-driven request handling on the back-end server. We have omitted interactions with the database as the structure would get too cluttered.

This approach enables clients to make requests to the back-end without getting blocked until the execution of the request finishes, thus reducing the amount of load on the back-end, reducing the number of connections to the server, and allowing interaction continuously - see Figure 4.9.

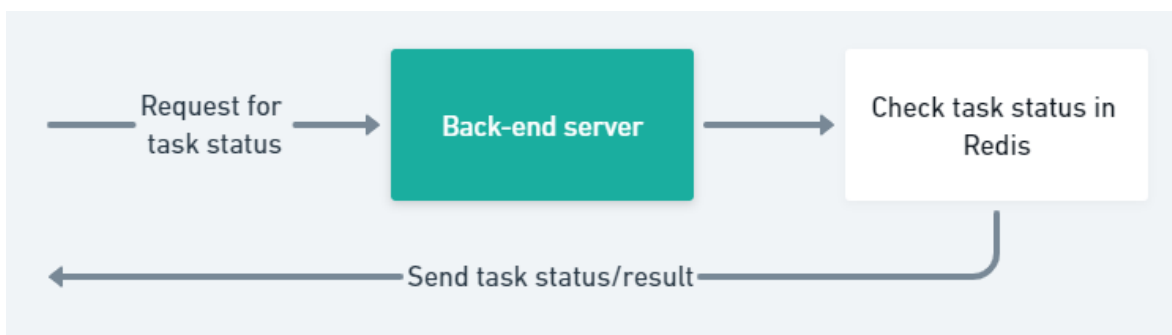


Figure 4.9: Serving task status is a simple GET request endpoint on the back-end server. The task information is gathered from the local redis database.

We are conscious that this implementation comes with huge development complexities, and make the interaction with clients tedious, as they have to check the status of a task until it has finished, however we believe that the benefits justify the choice.

Requests targeting MongoDB

Clients may be interested in getting a list of their environments, dataset information for each instance, get training logs for an environment, and more. Therefore, the back-end is the component that fulfils requests intended for MongoDB.

These requests are handled in the usual HTTP approach, where an endpoint is targeted, the server interacts with the MongoDB database to fetch the result, and return it as a response. These requests are blocking because they keep the HTTP connection open as long as the request is handled, however interacting with the database is an operation that is brief and not performance heavy, and intended to be handled with HTTP requests.

4.2.4 Front-end

As stated in 3.1.2, we used NextJS to build our front-end. Although we have not deployed the front-end to a host, and thus have not benefited from all the features of NextJS, it still provided us with quick development times - e.g., automatically refreshing pages on development changes -, provided caching for our requests, improved performance and also allowed us to modularise our codebase.

We have adhered to a workflow where pieces of User Interface (UI) were made as reusable as possible, therefore resulting in components such as modals - i.e., pop-ups -, alert boxes, forms, and tables that could be reused on any page - see F for examples. We individualised each component by creating self-managed states for each. Where components had to make updates on a parent component - i.e., the component that renders it - we allowed for state change function to be passed as **props** - i.e., as arguments to be used by the rendered child component.

The front-end, although minimalist in looks, operates complex handlers for requests behind the scenes. We used SWR [39] to create hooks (functions) that fetch the back-end for data. These hooks are special by their implementation. SWR hooks can be set to fetch an endpoint at a given period of time, on interactions with the web-page - e.g., leaving the page - and it automatically performs caching and cache invalidation. Thus, we can dynamically update tables and loading spinners without reloading the page.

We present our form for creating environments in Figure 4.10. Users can simply enter the number of instances they need, the type of machines running on Google Cloud and add probabilities of failure. However, these steps are not enough to start a learning process. As illustrated in Figure 4.11, a user must first enter the distribution of the training data on each device - a value smaller than the length of the dataset used to randomly distribute as many data points -, add data for the task - training, validation or testing -, create a model, and only then run a training round.

Finally, the front-end is responsible to fetch the status of tasks, as explained in 4.2.3. The front-end presents loading spinners for tasks that are event-driven - e.g., environment creation - and dynamically updates the UI to reflect progress. We have made use of JavaScript's **setTimeout** method to fetch the status of a task every second, and handle errors and finalisation gracefully. An example of status update can be seen in G.

Final Year Project

Environments

Datasets

Logout

Create environment

Machines configuration

Number of instances

GENERAL-PURPOSE COMPUTE-OPTIMIZED

Machines series

Please select your machine series

Machines type

Please select your machine type

Environment options

Instance number

Probability of failure

ADD OPTIONS

Figure 4.10: Form for creating an environment. The form abstracts away the dynamism of creating the VMs on GC, starting the Docker containers, setting up the configurations of the instance and getting the IP addresses of the machines.



Figure 4.11: User workflow for training a model using our platform.

Chapter 5

Testing and appraisal

During development, we have constantly emphasised testing our code. We have prepared in our repository end-to-end testing requests which can be imported using Postman [37], and executed to test specific endpoints in our solution. However, we also adhered to industry standards and have written automated tests for the controllers and instances.

Due to the difficulty of testing some functions requiring complex mocking of files and requests, we have aimed at getting a code coverage of above 70% for our two main components - controllers and instances. We used **pytest** [23], a testing framework for python, to automatically run our tests and make code coverage detection possible.

In Figure 5.1, a list of files with their corresponding number of statements, missed statements, percentage of coverage and untested lines is presented. Statements are parts of the code that execute a command - e.g., function call. For the main controller, it was essential to test the functions interacting with the database - e.g., functions in 'environment_helpers' - and those performing training - e.g., functions in 'controller_helpers'. We have also extensively tested the available routes and error handlers. Although we did not provide automated tests for functionalities such as creating an environment, we do provide Postman end-to-end requests to test for the creation of environments. Moreover, we have also performed manual tests for creating environments with different configurations.

Furthermore, we have also thoroughly tested the instance code as shown in Figure 5.2. Our emphasis was on the factory classes that handle model creation - i.e., files named with 'nn_' - and the helper methods for handling interactions with the local dataset - 'data_helpers' - and for model train/update - 'model_helpers'. We have emphasised tests related to creating models based on configurations that cover all the available classes generated by the factories.

Finally, to test the FL capabilities of our solution, we have implemented a U-Net [41] NN. We found this model to be a good test as it entails 23 convolution and activation layers, it uses pooling and up-sampling, and it makes use of our implementation of concatenation between layers. Furthermore, the model has to learn thousands of parameters, which need to be aggregated and managed by the controller as well. We used two training situations: a single instance and 3 instances. We were able to train the models in both cases without crashes. However, we do not provide further analysis of the models in this work as the scope is outside of our current solution.

5.1 Appraisal

We introduce this section to further discuss impediments in our current approach, and note possible solutions.

Firstly, our current approach to training does not work in accordance with limitations imposed by GC. Due to our blocking approach to training - i.e., the main controllers keep the HTTP connection

open until training finishes - our connection gets cut off on long lasting training processes. Due to the time constraints and added complexity, we are unable to provide a working solution to this problem. However, we identified a possible solution by transforming the training process into an event-driven approach. This approach would be similar to our back-end server implementation, where tasks for training would be created on instances and their status checked by the controller at a given period of time, thus mitigating the issue of long connectivity.

Secondly, testing models after training requires having access to the intrinsic factory classes used to build the network. This is due to the implementation of **PyTorch** and we have no control over this behaviour. A possible solution may be to investigate an approach using other libraries, or allowing users to download our Python code for the network creation.

Thirdly, we found that writing **JSONs** to build **NNs** is not as intuitive as writing code. Therefore, it may be viable to allow users to add Python code scripts to be executed instead. This solution would also solve the issue of loading models after training as mentioned above.

Finally, we have identified a bottleneck in our configuration request during environment creation. Due to instances running Docker containers on **GC**, they suffer an unpredictable delay. Therefore, when controllers send the configurations, it may happen that the docker container has not yet been started, thus the request will fail, and the operation triggers a clean-up process that erases the environment from MongoDB and **GC**. We have tried fixing this issue by adding a **sleep** command of 90 seconds to accommodate for the delay before sending the configurations. However, from our tests, this does not always work as the number of created instances affects the creation duration. A possible solution to this issue would be for the instances to make a request to the controller when they are ready. Then, the controllers could store that information in the database. During this time, the controller could wait for this state update before sending the configurations, or until a timeout occurs. If the state becomes ready, the configurations would be safely sent over to the running instances - see Figure 5.3.

Name	Stmts	Miss	Cover	Missing
app.py	22	0	100%	
conf/test.py	1	0	100%	
environment_classes/_init_.py	0	0	100%	
environment_classes/environment.py	9	0	100%	
environment_classes/target_environment.py	4	0	100%	
error_handlers/_init_.py	0	0	100%	
error_handlers/abort_handler.py	7	0	100%	
helpers/_init_.py	0	0	100%	
helpers/controller_helpers.py	173	74	57%	22-23, 45, 40, 53, 58-83, 88-91, 139-161, 165-173, 178, 183-181, 231-234, 253-256
helpers/environment_helpers.py	123	47	62%	16-17, 35-49, 55-60, 64-67, 127, 136, 163-170, 174-175, 179-180, 185, 189-201, 212-221, 232-247, 252-255, 259, 263, 267
helpers/nm_helpers.py	2	0	100%	
helpers/request_helpers.py	91	56	38%	10-11, 20-21, 28, 36-51, 57, 74-88, 92-98, 102-110, 114-129, 133-140, 144-152
helpers/tests/_init_.py	0	0	100%	
helpers/tests/mocks/_init_.py	0	0	100%	
helpers/tests/mocks/mock_json_response.py	11	6	45%	4-6, 9, 12, 15
helpers/tests/mocks/mock_stream_response.py	12	2	83%	13, 16
helpers/tests/mocks/mock_text_response.py	8	4	50%	4-5, 8, 11
helpers/tests/test_controller_helpers.py	99	6	94%	161-162, 165-166, 170-171
helpers/tests/test_controller_helpers_request_mocks.py	120	2	98%	210, 227
helpers/tests/test_environment_helpers.py	93	2	98%	18-19
helpers/tests/test_request_helpers.py	55	6	89%	41-42, 53, 64, 74, 83
nn_model_factory/_init_.py	0	0	100%	
nn_model_factory/model/_init_.py	0	0	100%	
nn_model_factory/concatenate.py	12	5	58%	0-10, 13, 16
nn_model_factory/nn_activation_factory.py	73	10	86%	29, 52-54, 82-84, 97-99
nn_model_factory/nn_factory/_init_.py	0	0	100%	
nn_model_factory/nn_factory/nn_convolution_layer_factory.py	43	0	100%	
nn_model_factory/nn_factory/nn_layer_factory.py	5	1	80%	7
nn_model_factory/nn_factory/nn_linear_layer_factory.py	25	0	100%	
nn_model_factory/nn_factory/nn_pooling_layer_factory.py	95	0	100%	
nn_model_factory/nn_factory/nn_vision_layer_factory.py	15	1	93%	18
nn_model_factory/nn_layer_factory.py	24	0	100%	
nn_model_factory/nn_model.py	32	13	59%	33-41, 44-54
nn_model_factory/tests/_init_.py	0	0	100%	
nn_model_factory/tests/test_activation_factory.py	60	0	100%	
nn_model_factory/tests/test_convolution_layer_factory.py	54	1	98%	209
nn_model_factory/tests/test_linear_layer_factory.py	32	0	100%	
nn_model_factory/tests/test_model.py	32	0	100%	
nn_model_factory/tests/test_pooling_layer_factory.py	106	1	99%	297
nn_model_factory/tests/test_vision_layer_factory.py	16	0	100%	
routes/_init_.py	0	0	100%	
routes/environment.py	86	53	38%	19-20, 26-46, 54-59, 66-84, 90-102, 108-120, 128-143
routes/error_handlers/_init_.py	0	0	100%	
routes/error_handlers/server_errors_handler.py	55	35	36%	12-13, 18-24, 32-40, 46-54, 58-59, 65-81
routes/health.py	10	1	90%	13
routes/model.py	40	19	52%	15-16, 22-31, 44-51, 57-65
TOTAL	1645	345	79%	

Figure 5.1: Code coverage for Main Controller server. The columns represent, starting from the left, the file name, the number of statements - e.g., assignments, function calls -, the number of missed statements, the percentage of statement coverage, and the line numbers for missing statements - i.e., untested statements. The tests show a 79% coverage, which is enough to consider our code safe. We did omit tests related to functionalities involving file management - which can be found in environment, controller and request helpers - but also those relating to interacting with Terraform and [GC](#) - e.g., 'routes/environment'.

Name	Stmts	Miss	Cover	Missing
conftest.py	0	0	100%	
helpers/_init__.py	0	0	100%	
helpers/app_helpers.py	60	14	77%	11-12, 17-18, 23-27, 61, 72, 76-78
helpers/data_helpers.py	54	18	67%	25-27, 46-61, 73-74, 82
helpers/model_helpers.py	8	0	100%	
helpers/nn_helpers.py	4	0	100%	
helpers/tests/_init__.py	0	0	100%	
helpers/tests/test_app_helpers.py	63	0	100%	
helpers/tests/test_data_helpers.py	53	0	100%	
helpers/tests/test_model_helpers.py	16	0	100%	
nn_loss/_init__.py	0	0	100%	
nn_loss/nn_loss_factory.py	49	1	98%	23
nn_loss/tests/_init__.py	0	0	100%	
nn_loss/tests/test_loss_factory.py	56	0	100%	
nn_model_factory/_init__.py	0	0	100%	
nn_model_factory/model/_init__.py	0	0	100%	
nn_model_factory/model/concatenate.py	12	5	58%	8-10, 13, 16
nn_model_factory/nn_activation_factory.py	73	10	86%	29, 52-54, 82-84, 97-99
nn_model_factory/nn_factory/_init__.py	0	0	100%	
nn_model_factory/nn_factory/nn_convolution_layer_factory.py	43	0	100%	
nn_model_factory/nn_factory/nn_layer_factory.py	5	1	80%	7
nn_model_factory/nn_factory/nn_linear_layer_factory.py	25	0	100%	
nn_model_factory/nn_factory/nn_pooling_layer_factory.py	95	0	100%	
nn_model_factory/nn_factory/nn_vision_layer_factory.py	15	1	93%	18
nn_model_factory/nn_layer_factory.py	24	0	100%	
nn_model_factory/nn_model.py	32	13	59%	33-41, 44-54
nn_model_factory/tests/_init__.py	0	0	100%	
nn_model_factory/tests/test_activation_factory.py	60	0	100%	
nn_model_factory/tests/test_convolution_layer_factory.py	54	1	98%	209
nn_model_factory/tests/test_linear_layer_factory.py	32	0	100%	
nn_model_factory/tests/test_model.py	32	0	100%	
nn_model_factory/tests/test_pooling_layer_factory.py	106	1	99%	297
nn_model_factory/tests/test_vision_layer_factory.py	16	0	100%	
nn_optimizer/_init__.py	0	0	100%	
nn_optimizer/nn_optimizer_factory.py	58	13	78%	19, 26-27, 35-36, 44-45, 53-54, 62-63, 76-77
nn_optimizer/tests/test_optimizer_factory.py	35	0	100%	
TOTAL	1080	78	93%	

Figure 5.2: Code coverage for instance server. The columns represent, starting from the left, the file name, the number of statements - e.g., assignments, function calls -, the number of missed statements, the percentage of statement coverage, and the line numbers for missing statements - i.e., untested statements. Our tests cover 93% of the code, which allows us to say the code is thoroughly tested. Most of the code that is left untested is either related to error handling - which was difficult to mock - or required complex request mocks that we were not able to implement - e.g., file transfer.

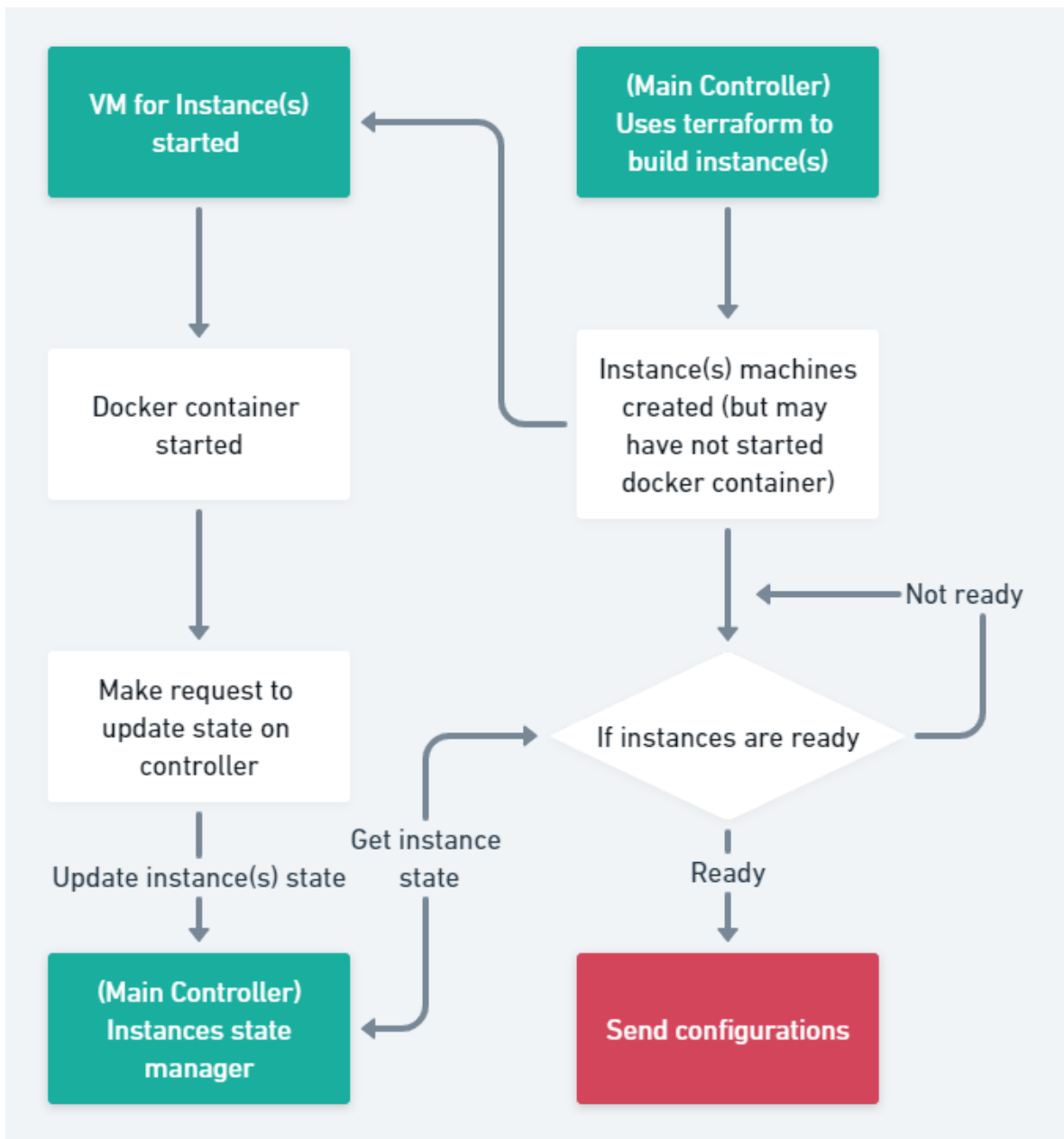


Figure 5.3: Simplified state transition in the case instances inform controllers about their state. The instate state manager in the controller could store the states in MongoDB or a local database.

Chapter 6

Conclusion

Our solution showed that it is possible to build a platform to create testing environments for working with [FL](#), while also allowing for flexibility, ease-of-use and real-life simulation. We were able to automate the creation of infrastructure, and also allowed for having the option to configure the machines used, how data is distributed, what model to be used and how to perform the training with chances of failure. However, we have identified limits within our approach and choice of libraries, architecture and cloud provider.

Identifying our limits showed us why creating automated infrastructure for [FL](#) is not an easy task and why no solution has been created before. Although we simplified the process by using [JSON](#) for sending model and training information, we realised it may not be as beneficial, and would require access to the code handling the network creation to be able to load the model. Moreover, our solution was able to handle images only, whereas text data may be of more interest for researchers. We found that generalising the type of data used is complex, and would require careful analysis of possible options.

In the end, we think our solution is a good beginning for automating the process of testing [FL](#) algorithms, and may provide a starting point for future research interest. Our approach showed how cloud-based providers could be used for simulating devices, how the process may be carried out in an event-driven architecture, how [UI](#) could abstract the intricacies of environment interactions and also how building models could be dynamic and flexible.

6.1 Future additions

Although we mentioned in our requirements the need for various aggregation methods, our current approach only averages model parameters. However, as briefly mentioned in [4](#), our design would easily allow for the addition of other aggregation methods.

Furthermore, our current data distribution for training is random. Our design may also allow for various options. for example using the specific data at the given index - e.g., use image 3, 4, etc.

Finally, our approach may also enable working with text data and other types of networks. This may come with the requirement of giving users the possibility of adding their own code snippets, thus increasing complexity and security requirements, however we find it an important addition to our solution.

Chapter 7

Bibliography

- [1] ASAD, M., MOUSTAFA, A., AND ITO, T. Federated learning versus classical machine learning: A convergence comparison.
- [2] ASTUDILLO, M. Bulljs. <https://github.com/OptimalBits/bull>.
- [3] BECK, K., BEEDLE, M., VAN BENNEKUM, A., COCKBURN, A., CUNNINGHAM, W., FOWLER, M., GRENNING, J., HIGHSMITH, J., HUNT, A., JEFFRIES, R., KERN, J., MARICK, B., MARTIN, R. C., MELLOR, S., SCHWABER, K., SUTHERLAND, J., AND THOMAS, D. Manifesto for agile software development, 2001.
- [4] BONAWITZ, K. A., EICHNER, H., GRIESKAMP, W., HUBA, D., INGERMAN, A., IVANOV, V., KIDDON, C. M., KONEČNÝ, J., MAZZOCCHI, S., MCMAHAN, B., OVERVELDT, T. V., PETROU, D., RAMAGE, D., AND ROSELANDER, J. Towards federated learning at scale: System design. In *SysML 2019* (2019). To appear.
- [5] DENG, L., HINTON, G., AND KINGSBURY, B. New types of deep neural network learning for speech recognition and related applications: an overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 8599–8603.
- [6] FOUNDATION, O. Nodejs. <https://nodejs.org/en/about/>.
- [7] FOUNDATION, P. S. Python concurrent library. <https://docs.python.org/3/library/concurrent.futures.html>.
- [8] GALAKATOS, A., CROTTY, A., AND KRASKA, T. Distributed machine learning. 1196–1201.
- [9] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [10] GOOGLE. Compute engine: Virtual machines (vms) — google cloud. <https://cloud.google.com/compute>.
- [11] GOOGLE. Google cloud available machine types. <https://cloud.google.com/compute/docs/machine-types>.
- [12] GOOGLE. Why google cloud. <https://cloud.google.com>.
- [13] HAMMES, D., MEDERO, H., AND MITCHELL, H. Comparison of nosql and sql databases in the cloud.
- [14] HASHICORP. Terraform documentation. <https://www.terraform.io/docs>.
- [15] HILL, K. How target figured out a teen girl was pregnant before her father did. <https://www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did/?sh=7b497a566668>, February 2012. [Online; posted Feb. 16, 2012].

- [16] HINTON, G. E., OSINDERO, S., AND TEH, Y.-W. A fast learning algorithm for deep belief nets. *Neural Comput.* 18, 7 (jul 2006), 1527–1554.
- [17] HINTON, G. E., AND SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504–507.
- [18] INC., M. Nosql vs sql databases. <https://www.mongodb.com/nosql-explained/nosql-vs-sql>.
- [19] KAIROUZ, P., MCMAHAN, H. B., AVENT, B., BELLET, A., BENNIS, M., BHAGOJI, A. N., BONAWITZ, K., CHARLES, Z., CORMODE, G., CUMMINGS, R., D’OLIVEIRA, R. G. L., EICHNER, H., ROUAYHEB, S. E., EVANS, D., GARDNER, J., GARRETT, Z., GASCÓN, A., GHAZI, B., GIBBONS, P. B., GRUTESER, M., HARCHAOUI, Z., HE, C., HE, L., HUO, Z., HUTCHINSON, B., HSU, J., JAGGI, M., JAVIDI, T., JOSHI, G., KHODAK, M., KONEČNÝ, J., KOROLOVA, A., KOUSHANFAR, F., KOYEJO, S., LEPOINT, T., LIU, Y., MITTAL, P., MOHRI, M., NOCK, R., ÖZGÜR, A., PAGH, R., RAYKOVA, M., QI, H., RAMAGE, D., RASKAR, R., SONG, D., SONG, W., STICH, S. U., SUN, Z., SURESH, A. T., TRAMÈR, F., VEPAKOMMA, P., WANG, J., XIONG, L., XU, Z., YANG, Q., YU, F. X., YU, H., AND ZHAO, S. Advances and open problems in federated learning, 2021.
- [20] KETKAR, N. *Introduction to PyTorch*. Apress, Berkeley, CA, 2017, pp. 195–208.
- [21] KONEČNÝ, J., MCMAHAN, H. B., YU, F. X., RICHTÁRIK, P., SURESH, A. T., AND BACON, D. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).
- [22] KRASKA, T., TALWALKAR, A., DUCHI, J. C., GRIFFITH, R., FRANKLIN, M. J., AND JORDAN, M. I. Mlbase: A distributed machine-learning system. In *Cidr* (2013), vol. 1, pp. 2–1.
- [23] KREKEL, H., AND PYTEST-DEV TEAM. Pytest - python testing framework. <https://docs.pytest.org/en/7.1.x/>.
- [24] LIU, J., HUANG, J., ZHOU, Y., LI, X., JI, S., XIONG, H., AND DOU, D. From distributed machine learning to federated learning: A survey, 2021.
- [25] LIU, L., ZHANG, J., SONG, S., AND LETAIEF, K. B. Client-edge-cloud hierarchical federated learning. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)* (2020), pp. 1–6.
- [26] LTD., R. Redis. <https://redis.io>.
- [27] MARKOFF, J. Scientists see promise in deep-learning programs. <https://www.nytimes.com/2012/11/24/science/scientists-see-advances-in-deep-learning-a-part-of-artificial-intelligence.html>, November 2012. [Online; posted Nov. 23, 2012].
- [28] MCCULLOCH, W. S., AND PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (Dec 1943), 115–133.
- [29] MCMAHAN, B., MOORE, E., RAMAGE, D., HAMPSON, S., AND Y ARCAS, B. A. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics* (2017), PMLR, pp. 1273–1282.
- [30] MCMAHAN, B., AND RAMAGE, D. Federated learning: Collaborative machine learning without centralized training data. <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>, April 2017. [Online; posted Thursday, April 6, 2017].
- [31] MCMAHAN, B., AND RAMAGE, D. *Your phone personalizes the model locally, based on your usage (A). Many users’ updates are aggregated (B) to form a consensus change (C) to the shared model, after which the procedure is repeated.* Brendan McMahan and Daniel Ramage, Apr 2017.
- [32] META PLATFORMS, I. React library. <https://reactjs.org>.

- [33] MOHRI, M., SIVEK, G., AND SURESH, A. T. Agnostic federated learning. In *International Conference on Machine Learning* (2019), PMLR, pp. 4615–4625.
- [34] NETWORKS, F. Nginx. <https://www.nginx.com>.
- [35] PALLETS. Flask framework, 2010.
- [36] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.
- [37] POSTMAN, I. Postman. <https://www.postman.com/product/what-is-postman/>.
- [38] RAUCH, G. Next js - web application framework. <https://nextjs.org>.
- [39] RAUCH, G. Next js - web application framework. <https://swr.vercel.app>.
- [40] RIES, E. Minimum viable product: a guide, August 2009.
- [41] RONNEBERGER, O., FISCHER, P., AND BROX, T. U-net: Convolutional networks for biomedical image segmentation, 2015.
- [42] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. Learning internal representations by error propagation.
- [43] STRONGLOOP, IBM, AND OTHER EXPRESSJS.COM CONTRIBUTORS. Express. <https://expressjs.com>.
- [44] TECHNOPENEDIA. Minimum viable product (mvp), 2020.
- [45] VERBRAEKEN, J., WOLTING, M., KATZY, J., KLOPPENBURG, J., VERBELEN, T., AND RELLERMEYER, J. S. A survey on distributed machine learning. *ACM Comput. Surv.* 53, 2 (mar 2020).
- [46] WANG, S., TUOR, T., SALONIDIS, T., LEUNG, K. K., MAKAYA, C., HE, T., AND CHAN, K. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1205–1221.
- [47] YANG, Q., LIU, Y., CHEN, T., AND TONG, Y. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.* 10, 2 (jan 2019).
- [48] ZHANG, X., HU, M., XIA, J., WEI, T., CHEN, M., AND HU, S. Efficient federated learning for cloud-based aiOT applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 11 (2021), 2211–2223.

Appendix A

GitHub repository

<https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2021/dxb020>

Appendix B

NNLayerFactory

```
...
class NNLayerFactory:
    def get_layer(self, layer_type, subtype, parameters):
        options = {
            "Convolution": self._build_convolution,
            "Pooling": self._build_pooling,
            "Linear": self._build_linear,
            "Vision": self._build_vision
        }
        if layer_type in options:
            return options[layer_type](subtype, parameters)
        raise Exception("Layer_type_not_in_options")

    def _build_convolution(self, subtype, parameters):
        convolutional_layer_factory = NNConvolutionLayerFactory()
        return convolutional_layer_factory.get_layer(subtype, parameters)
...
```

Appendix C

NNConvolutionLayerFactory

```
...
class NNConvolutionLayerFactory(NNAbstractLayerFactory):
    def get_layer(self, layer_type, parameters):
        options = {
            "Conv1d": self._build_conv1d,
            "Conv2d": self._build_conv2d,
            "Conv3d": self._build_conv3d,
            "ConvTranspose1d": self._build_conv_transpose1d,
            "ConvTranspose2d": self._build_conv_transpose2d,
            "ConvTranspose3d": self._build_conv_transpose3d,
            "Fold": self._build_fold,
            "Unfold": self._build_unfold,
        }
        if layer_type in options:
            return options[layer_type](parameters)
        raise Exception("Convolutional_layer_type_not_in_options")

    def _build_conv1d(self, parameters):
        parameters_set = {
            "in_channels",
            "out_channels",
            "kernel_size",
            "stride",
            "padding",
            "padding_mode",
            "dilation",
            "groups",
            "bias",
        }
        parameters = get_params_from_list(parameters, parameters_set) # method that searches for parameters in the list
        return nn.Conv1d(**parameters)
...
```

Appendix D

Example neural network JSON

```
{
  "layer": {
    "layer_type": "Convolution",
    "subtype": "Conv2d",
    "parameters": {
      "in_channels": 1,
      "out_channels": 32,
      "kernel_size": 3,
      "padding": 1
    }
  },
  {
    "activation": {
      "activation_type": "ReLU",
      "parameters": {
        "inplace": true
      }
    }
  }
}
```

Appendix E

Example training parameters JSON


```
{
  "loss": {
    "loss_type": "CrossEntropyLoss",
    "parameters": {}
  },
  "optimizer": {
    "optimizer_type": "RMSprop",
    "parameters": {
      "lr": 0.001,
      "weight_decay": 0.00000001,
      "momentum": 0.9
    }
  },
  "hyperparameters": {
    "epochs": 60,
    "batch_size": 4,
    "reshape": "4, 1, 96, 96",
    "normalize": true
  }
}
```


Appendix F

Modals

Add environment data distribution

TRAINING DATASET VALIDATION DATASET TESTING DATASET

Upload data image(s) 

Upload labels image(s) 

SAVE

CLOSE

Figure F.1: Form for adding data (images) to environment

Add model layers

Layer types and the available options

Convolution


▼

Pooling

▼

Linear

▼

Specify your options using the following structure :

```
1  [
2    {
3      layer: {
4        layer_type: 'Convolution',
5        subtype: 'Conv2d',
6        parameters: {
7          in_channels: 1,
8          out_channels: 4,
9          kernel_size: 3,
10         stride: 1,
11         padding: 1
12       }
13     },
14   ],
15   {
16     layer: {
17       layer_type: 'Convolution',
18       subtype: 'Conv2d',
19       parameters: {
20         in_channels: 1,
21         out_channels: 4,
22         kernel_size: 3,
23         stride: 1,
24         padding: 1
25       }
26     }
27   }
```

SAVE

CLOSE

Figure F.2: Modal form for creating the NNs model

Train model on environment

Training iterations

Maximum trials for device availability


Minimum number of required devices

Losses, optimisers and hyperparameter options

Loss ▼

Optimiser ▼

Hyperparameters ▼

Specify your options using the following structure :

```
1  {
2    loss: {
3      loss_type: 'CrossEntropyLoss',
4      parameters: {}
5    },
6    optimizer: {
7      optimizer_type: 'RMSprop',
8      parameters: {
9        lr: 0.001,
10       weight_decay: 1e-8,
11       momentum: 0.9
12     }
13   },
14   hyperparameters: {
15     epochs: 60,
16     batch_size: 4,
17     reshape: '4, 1, 96, 96',
18     standardize: true
19   }
20 }
```



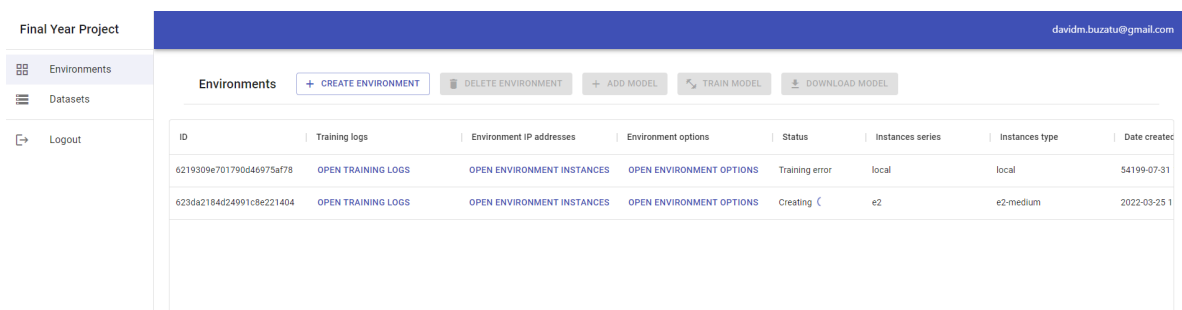
START TRAINING

CLOSE

Figure F.3: Form for starting the FL training process

Appendix G

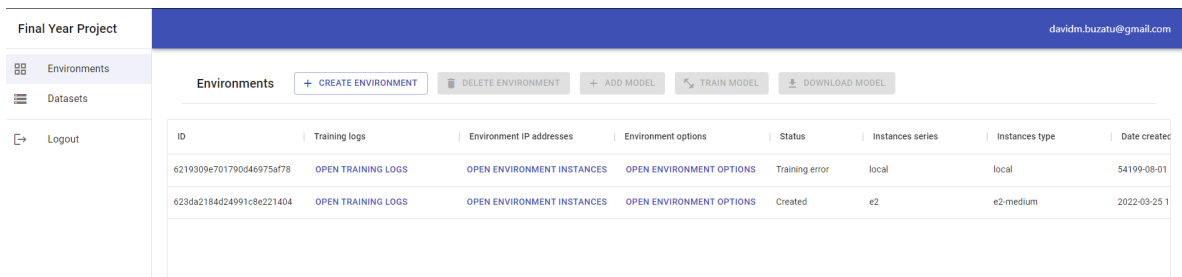
Environment states update



The screenshot shows a web interface for 'Final Year Project' with a user 'davidm.buzatu@gmail.com'. The left sidebar has 'Environments', 'Datasets', and 'Logout'. The main content area is titled 'Environments' and contains buttons: '+ CREATE ENVIRONMENT', 'DELETE ENVIRONMENT', '+ ADD MODEL', 'TRAIN MODEL', and 'DOWNLOAD MODEL'. Below these is a table with 8 columns: ID, Training logs, Environment IP addresses, Environment options, Status, Instances series, Instances type, and Date created. Two rows are visible. The first row has ID '6219309e701790d46975af78', status 'Training error', and date '54199-07-31'. The second row has ID '623da2184d24991c8e221404', status 'Creating', and date '2022-03-25 1'.

ID	Training logs	Environment IP addresses	Environment options	Status	Instances series	Instances type	Date created
6219309e701790d46975af78	OPEN TRAINING LOGS	OPEN ENVIRONMENT INSTANCES	OPEN ENVIRONMENT OPTIONS	Training error	local	local	54199-07-31
623da2184d24991c8e221404	OPEN TRAINING LOGS	OPEN ENVIRONMENT INSTANCES	OPEN ENVIRONMENT OPTIONS	Creating	e2	e2-medium	2022-03-25 1

Figure G.1: Environment creation state



The screenshot shows the same web interface as Figure G.1, but the status of the second environment has changed to 'Created'.

ID	Training logs	Environment IP addresses	Environment options	Status	Instances series	Instances type	Date created
6219309e701790d46975af78	OPEN TRAINING LOGS	OPEN ENVIRONMENT INSTANCES	OPEN ENVIRONMENT OPTIONS	Training error	local	local	54199-08-01
623da2184d24991c8e221404	OPEN TRAINING LOGS	OPEN ENVIRONMENT INSTANCES	OPEN ENVIRONMENT OPTIONS	Created	e2	e2-medium	2022-03-25 1

Figure G.2: Environment created state

Appendix H

List of Acronyms

AI Artificial Intelligence	3
FL Federated Learning	3
GUI Graphical User Interface	3
CNNs Convolutional Neural Networks	6
NNs Neural Networks	3
DNNs Deep Neural Networks	3
ML Machine Learning	6
JSON JavaScript Object Notation	6
non-i.i.d Non independent and identically distributed	4
IaC Infrastructure as Code	9
CLI Command Line Interface	9
HTTP Hypertext Transfer Protocol	10
APIs Application Programming Interfaces	10
API Application Programming Interface	10
NoSQL Not only Structured Query Language	11
SQL Structured Query Language	11

MVP Minimum viable product	15
VMs Virtual Machines	8
IP Internet Protocol	19
GCP Google Cloud Platform	8
CE Compute Engine	8
VM Virtual Machine	14
GPU Graphics Processing Unit	8
CRUD Create/Read/Update/Delete	20
GC Google Cloud	25
CPU Central processing unit	25
UI User Interface	28
NN Neural Network	3