Computer Science and Information Systems 21(4):1269–1292 https://doi.org/10.2298/CSIS231224036P

PTB-FLA Development Paradigm Adaptation for ChatGPT

Miroslav Popovic[1], Marko Popovic[2], Ivan Kastelan[1], Miodrag Djukic[1], and Ilija Basicevic[1]

[1] University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovica 6,
21000 Novi Sad, Serbia
{miroslav.popovic, ivan.kastelan, miodrag.djukic, ilija.basicevic}@rt-rk.uns.ac.rs
[2] RT-RK Institute for Computer Based Systems, Narodnog fronta 23a,
21000 Novi Sad, Serbia
marko.popovic@ rt-rk.com

Abstract. The Python Testbed for Federated Learning Algorithms (PTB-FLA) is a simple Python FL framework that is targeting edge systems and is by its design easy to use by human ML&AI developers. The original PTB-FLA development paradigm intended for humans consists of the four phases (producing the sequential code, the federated code, the federated code with callbacks, and the PTB-FLA code, respectively), and hence dubbed the four-phases (development) paradigm, was validated in the case study on the logistic regression. In this paper, we adapted the original paradigm into the two new paradigms for ChatGPT, named the adapted four-phases paradigm and the adapted two-phases paradigm, respectively. In tune with its name, the latter consists of two phases (producing the sequential and the PTB-FLA code, respectively). We successfully validated both new paradigms using the same case study on logistic regression that was used for the original paradigm. The results are positive and encouraging as the resulting program codes are of better quality than the codes solely made by humans using the original paradigm.

Keywords: Decentralized Intelligence, Federated Learning, Python, ChatGPT, Software Development.

1. Introduction

Contemporary Internet in a simplified view can be seen as the so-called cloud-fog-edge continuum (or hierarchy) with various devices (laptops, smart phones, smart sensors, actuators, IoTs, robots, etc.) connected at its edge. This complex system (or system of systems) is highly heterogenous (as there are various kinds of cloud, fog, and edge systems/networks) and dynamic (as devices, or complete systems, may dynamically connect/disconnect i.e., join, and leave, or reconfigure while being connected). Developing applications for this environment without tools is both tedious and costly.

Both academia and industry are aware of this issue, and one of many ongoing projects trying to address it is the ongoing EU Horizon 2020 project TaRDIS [1], which aims to create a toolbox for easy programming of innovative applications, such as: (1) multi-level smart electrical vehicles charging, (2) privacy-preserving learning through decentralised training in smart homes, (3) distributed navigation concepts for LEO satellite constellations, and (4) highly resilient factory shop floor digitalization to be implemented in a real factory comprising several production lines, a warehouse, an intralogistics fleet of robots, etc. These systems must use some ML/AI algorithms and be trustworthy and responsible.

The Python Testbed for Federated Learning Algorithms (PTB-FLA) [2] is being developed within TaRDIS as a framework for developing federated learning algorithms (FLAs) i.e., as a runtime environment for FLAs under development. PTB-FLA was written in pure Python to keep application footprint small (to fit to IoTs) and to keep its installation simple. The PTB-FLA was designed with a simple API based on Single Program Multiple Data (SPMD) pattern, to be easy to use by ML&AI developers who do not need to be professional programmers, and as such, also amenable to ChatGPT.

Initially, PTB-FLA was validated and illustrated by three simple examples that were developed by humans using the ad hoc approach to directly write the decentralized PTB-FLA code [2]. For more complex applications, a more natural and feasible approach is to start from a given referent sequential code (either legacy or open source or newly crafted) and transform it into the semantically equivalent PTB-FLA code. Recently, PTB-FLA four-phases development paradigm appeared as a systematic guide for humans to transform a sequential code into the PTB-FLA code, and it was validated in a case study where the sequential logistic regression Python code [3] was transformed into the PTB-FLA codes (for both centralized and decentralized FLAs) [4].

More recently, PTB-FLA has been shown to be amenable to AI by successful redevelopment of the three simple examples from [2] using the ad hoc approach, where ChatGPT was given the context comprising the input sequential code and the target function main and was asked to create the target callback functions [5].

In the next three subsections we present the problem formulation and its solution, the paper achievements, and the paper organization.

1.1. Problem Formulation and Its Solution

The problem that we tried to solve in this paper, was how to adapt the PTB-FLA development paradigm used by humans for ChatGPT in order to (i) minimize the human labor by delegating a part of the job to ChatGPT, and at the same time to (ii) minimize the ChatGPT input by creating minimal contexts (i.e., prompts), because commercial AI services (like GPT-4) are charged according to the input they processed.

We solved that problem experimentally by using the simple iterative development process, which is described in Section 2 in detail. Here is a short summary: we used the text (guidelines) of the original development paradigm to create the initial ChatGPT context, and then in each iteration we used feedback from ChatGPT (in the form of questionnaire answers ChatGPT provided) to manually modify the context i.e., to adapt the paradigm, aiming towards the minimal context without redundant parts. In each iteration, the current paradigm (given in the current context) was also validated, because it had to produce the output PTB-FLA code that is semantically equivalent to the input sequential code. The application code that was used for this purpose is the logistic regression case study from [4].

1.2. Paper Achievements

As a continuation of [4] and [5], we firstly adapted the four-phases development paradigm originally devised for humans to guide ChatGPT to successfully develop the same algorithms as in [4] (to save space, here we present only the centralized algorithm development). Secondly, we adapted the four-phases paradigm into a two-phases development paradigm. As the first phase is always done by humans, the two-phases paradigm is rather close to the ideal solution where developers in the second phase immediately get the complete target PTB-FLA code. (All the paradigms are presented in Section 3.).

We would like to emphasize that both adapted paradigms (the four-phases and the two-phase) are original contributions that are presented for the first time in this paper, and both should be treated unbiasedly as both have their own strengths and weaknesses. The four-phases paradigm requires more human labor but provides better traceability (somewhat like gray box testing), whereas the two-phase paradigm requires minimal (almost no) labor but is less traceable (being based on the black box approach) and therefore riskier. The last statement should be taken as a simplified one and more as the authors’ expectation, because of the limitations of the experimental validation made in this paper (as discussed in the paper).

Both development paradigms were experimentally validated, see Sections 4 and 5. In the experimental validation, which was made in early summer 2023, ChatGPT was using the well-known GPT-3.5 model. We report on the adapted development paradigms performance in terms of human labour and size of ChatGPT context needed to develop the logistic regression PTB-FLA code in Section 6.

To the best of authors knowledge this is the first paper that formulated the problem and its solution of this kind and therefore we could not compare our paper with similar papers of other authors, so originality came at the price of lack of closely related work. However, we did our best to cover a more broadly related work in Section 7.

1.3. Paper Organization

The paper is organized as follows. Section 2 presents the development process, Section 3 presents the adapted development paradigms, Section 4 presents the four-phases paradigm validation, Section 5 presents the two-phases paradigm validation, Section 6 presents the adapted paradigms performance, Section 7 presents related work, and Section 8 concludes the paper.

2. Development Process

In this paper, to adapt and validate development paradigms (see Section 1.1), we used the simple iterative development process, which comprises the following steps: (1) specify the context (i.e., the prompt), (2) ask ChatGPT to complete the assignment, (3) test the generated code, (4) if the code successfully passes testing, stop, else if the code is close to what is expected and contains small number of simple bugs, manually modify the code (rational is further discussed below) and return to step 3, else return to step 1. We tried to create minimal contexts with the goal to minimize the overall effort/cost.

This paragraph is a short digression explaining the rational mentioned in the step 4 above. Initially, our intention was to make ideal contexts that would lead to ideal results. So, whenever the output code failed to pass the testing, we were returning to the step 1 and trying to refine the context. By doing this we discovered that ChatGPT exhibits one unexpected behaviour: given some context it may produce the code that is rather close to what is expected, but after giving it an extended context it may produce the code that is more far from what is expected than the previous one (see the paragraph below). Therefore, we switched to the following policy: if the output code is tweakable (i.e., has small number of bugs), then it is acceptable, and we consider the corresponding prompt sufficient.

We do not have an explanation for the unexpected behaviour mentioned above. This was happening both in the same session and in different sessions. A major difficulty in using ChatGPT is that by its design it tries not to repeat itself, so classical approach to testing with repetitive tries does not work for it.

For the sake of this research, to get an estimation of context quality (minimality), after an example was verified, we ask ChatGPT for feedback by asking it to complete the questionnaire comprising the following three questions: (i) On the scale from 1 to 10, where 10 is the hardest, how would you rate this assignment? (ii) What was the most important part of the context that you used to complete the assignment? and (iii) What was the redundant part of the context?

At the end of this section, we would like to clarify that we did not make any attempt to prevent ChatGPT hallucinations/confabulations, but if this happens and results in the generated code with bugs, this will be discovered during code testing (the phase 3 of the process outlined above), and this code will be rejected i.e., thrown away. However, preventing ChatGPT hallucinations/confabulations is an interesting research topic and a possible item of our future work.

3. Development Paradigms

In this section we present the PTB-FLA development paradigms. The next three subsections present the original four-phases paradigm that was initially devised for humans [4], the four-phases paradigm' phases, and the adapted two-phases paradigm for AI, respectively (the adapted four-phases paradigm adaptation is explained at the end of Section 3.3).

3.1. Original Four-Phases Paradigm for Humans

The original four-phases development paradigm [4] is primarily intended to serve as a FLA developer guide through the process of developing a target FLA using PTB-FLA, which we call the FLA development process. The input to this process is the Python sequential program code of target AI/ML algorithm, whereas the output from this process is the PTB-FLA code with the same semantics, which means that for given input data it produces the same output data with some tolerance *e*. The tolerance *e* is typically some small error value (ideally zero).

Of course, the output PTB-FLA code must be compliant with the PTB-FLA programming model which is a restricted programming model that imposes the following two restrictions: (1) using the Single Program Multiple Data (SPMD) pattern, and (2) specifying code for server and client roles in form of callback functions. Obviously, there are many ways to define such a development process. The original intention was to prescribe it as a paradigm which is much more disciplined than ad hoc development, but also not too rigid to keep it attractive and creative.

The main idea was to follow the principle of correct-by-construction, which when applied in this context meant to define the development process that for a given referent code yields the output PTB-FLA code with the equivalent semantics. Following the approach used by program compilers, the original four-phases development paradigm was devised as the series of program code transformation phases where each phase consumes its input code and produces the semantically equivalent code that is closer to the target PTB-FLA code.

3.2. Four-Phases Paradigm' Phases

This section contains the text adapted from [4]. The four development phases (i.e., phase 1, phase 2, phase 3, and phase 4) are by the convention named by their output code i.e., (1) the referent sequential code, (2) the federated sequential code, (3) the federated sequential code with callbacks, and (4) the PTB-FLA code, respectively.

The input to phase 1 is the raw sequential code and the output is the referent sequential code. The input raw sequential code may come from various sources and may have various forms. Nowadays, many AI/ML algorithm solutions in Python are available online in Colab notebooks, where snippets of textual mathematical explanations, code snippets, and graph plots dynamically created by code play (i.e., execution) are interleaved. Typically, these solutions are primarily intended for learning/understanding the solutions through interactive experimentation, where readers are even encouraged to tweak the code and play with it.

In phase 1, a PTB-FLA developer essentially must select only the necessary code snippets (leaving out the alternative or redundant snippets), to tweak them if needed, and to integrate them into a standalone Python module(s) that they could preferably run on their PC (localhost), typically in a terminal of some IDE. The important requirement for the referent sequential code is that for a given input dataset it must deterministically produce some output data e.g., learned (trained) model coefficients and/or some quality indicators like accuracy, because this output data is used as the referent data by the next development phases, which they must produce to be semantically equivalent. To that end, a PTB-FLA developer should use asserts that automatically compare whether the output data is (approximately) equal to the referent data, and if not, report the corresponding assertion error.

In phase 2, a PTB-FLA developer makes the federated sequential code by the following three steps: (1) partition the input dataset into partitions (that could be distributed across clients), (2) split the monolithic computing of the complete input dataset into a series of computing on individual partitions (that could be collocated with corresponding partitions) such that they produce the set of partition models, and (3) add the computing for aggregating the set of partition models into the final model and for computing quality indicators (that could be located on a server), as well as for comparing output and referent data. For example, a single function call (calling the function *f*) to process the complete dataset could be split into a series of function calls (calling the same function *f* with different arguments) to process individual dataset partitions. Note that this is still one sequential program that runs on a single machine (developer's PC).

In phase 3, a PTB-FLA developer makes the federated sequential code with callbacks by the following four steps: (1) copy (and tweak) the computing on an individual partition (say a partition number *i*) into the client callback function, (2) replace the series of computing on individual partitions with the series of client callback function calls (with the arguments corresponding to the partition *j* in the call number *j*), (3) copy (and tweak) the computing for aggregating the set of partition models to the final model into the server callback function, and (4) replace the former computing with the server callback function call (the code for computing quality indicators should remain in its place). In the running example, the series of function calls (calling the same function *f* with different arguments) to process individual dataset partitions should be replaced with the corresponding series of client callback function calls, which lead to indirect calls to the function *f* (each call to the client callback function maps to the corresponding call of the function *f*).

In phase 4, a PTB developer makes the PTB-FLA code by the following two steps: (1) add the code for creating the instance *ptb* of the class *PtbFla* and for preparing local and private data for all the instances, and (2) replace the code for calling callback functions (both the series of client callback function calls and the server callback function call) with the call to the function fl_centralized (in case of a centralized FLA) or fl_decentralized (in case of a decentralized FLA) on the object *ptb*.

3.3. Adapted Two-Phases Paradigm for ChatGPT

We adapted the original four-phases paradigm by keeping phase 1 and merging and optimizing the steps of the original phases 2 to 4 into the optimized phase 2 (with the intention to create a minimal context i.e., prompt). In phase 2, ChatGPT translates the referent sequential code into the semantically equivalent PTB-FLA code by following the instructions given in the next paragraph.

Rewrite the referent sequential code into the function main by following these four steps: (1) partition the input dataset into individual client partitions (that could be distributed across clients), (2) add the code to initialize the PtbFla object and prepare local and private data for the current instance, (3) replace the monolithic computing on the complete input dataset with the call to the function fl_(de)centralized on the object ptb, and (4) keep the code that evaluates the final model (i.e., the updated local data) on the test data. Put the computing on an individual partition that produces a partition model into the client callback function. Put the computing for aggregating the set of partition models into the final model into the server callback function.

The instructions above are meant to be general i.e., applicable to any code. However, note that the term "input dataset" mentioned in steps 1 and 3 above is somewhat ambiguous, and a more precise term is “training dataset”. We discovered this during paradigms’ validation by ChatGPT for the case study on logistic regression (to be presented shortly), wherein we switched to the later term and added the prologue step: (0) split input dataset into the training data and the test data. This note is valid for the four-phases paradigm’ phases (in section 3.2), too.

4. Four-Phases Paradigm Validation

This section presents the validation of the four-phases paradigm adapted for ChatGPT. The next subsections present the validation of phase 2, phase 3, and phase 4, respectively (phase 1 is conducted manually, so it is skipped). For each phase two tables are given – the first table contains the context, whereas the second table comprises the code generated by ChatGPT that was manually tweaked to make it runnable and comparable to the handwritten code (this code passed runtime verification and its output results are the same as for the handwritten code), the ChatGPT explanation of the generated code, and the ChatGPT answers to the questionnaire.

In the tables, comments in italic and enclosed by angle brackets are added as clarifications or replacements and are note part of the original text. Similarly, parts of the contexts in italic were declared as redundant by ChatGPT.

4.1. Phase 2

The context for phase 2, see Table 1, comprises the revised general phase 2 guidelines, the description of the sequential code (the part in italic turned out to be redundant), the assignment for ChatGPT, and the referent sequential code from phase 1.

Table 1. The context for phase 2 of the four-phases paradigm

"""
Revised general phase 2 guidelines:
< *Here was a copy of the text from Section 3.2 modified according to the note in Section 3.3.* >

Description of the referent sequential code:
The code uses the Social Network Ads (SNA) dataset, which comprises 400 samples (or rows) corresponding to user profiles and each record comprises the following features (or columns): (1) User ID, (2) Gender, (3) Age, (4) Estimated (yearly income), and (5) Purchased. Note that in this example only features Age and Purchased are used.
The main steps in the code are: (1) split SNA dataset into training and test datasets (320 and 80 samples, respectively), (2) train the linear logistic regression model comprising two coefficients, namely b0 and b1, (3) using the trained model, make predictions whether users in the test dataset would make purchase or not, and calculate the prediction accuracy as the ratio of test users for whom the predictions were correct.

Assignment:
Using the phase 2 guidelines, rewrite the referent sequential code, which is given in the function seq_base_case, into the federated sequential code in the function seq_horizontal_federated. Important notes related to the assigment follow.
Note 1: data should be first train_test_split into X_train, X_test, y_train, y_test, and then X_train and y_train should be np.array_split into noClients (=2) partitions!
"""

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from math import exp
# Import supplementary functions
from sup_funs import logistic_regression, evaluate

def seq_base_case():
    # Load the data
    data = pd.read_csv("Social_Network_Ads.csv")

    # Divide the data to training set and test set. For reproducible set random_state=42.
    X_train, X_test, y_train, y_test = train_test_split(data['Age'], \
    data['Purchased'], test_size=0.20, random_state=42)

    # Training the model
    b0, b1 = logistic_regression(X_train, y_train)
    print(f"Training on all the training records (original sequential code): b0={b0}, b1={b1}")

    # Test the model
    y_pred, accuracy = evaluate(X_test, y_test, b0, b1)
    print(f"Accuracy = {accuracy}")

def seq_horizontal_federated():
    """ Please complete this function """
```

The tweaked code and the feedback for phase 2 are given in Table 2. The code generated by ChatGPT did not contain any logical bugs and just a couple of tweaks (marked by the keyword TWEAK) were needed to make it runnable and comparable to the handwritten code. It is worth mentioning that the code generated by ChatGPT is of better quality than the manually written code in [4] (it is generalized for any number of clients and better uses numpy features).

Both the explanation of the generated code and the questionnaire answers are sound and human-like. ChatGPT rated the hardness of the assignment as 6 and found the given context almost minimal (only the part describing the SNA dataset is redundant).

Table 2. The tweaked code and the feedback for phase 2 of the four-phases paradigm

< To save space only the function seq_horizontal_federated is shown. >

```
def seq_horizontal_federated():
    # TWEAK 1: set noClients, e.g. to 2 (to be able to compare with the handwritten code).
    noClients = 2

    # Load the data
    data = pd.read_csv("Social_Network_Ads.csv")

    # Divide the data into training set and test set
    X_train, X_test, y_train, y_test = train_test_split(data['Age'], \
    data['Purchased'], test_size=0.20, random_state=42)

    # Split the training data into partitions
    partitions_X_train = np.array_split(X_train, noClients)
    partitions_y_train = np.array_split(y_train, noClients)

    # Perform federated learning on each partition
    models = []
    for i in range(noClients):
        # TWEAK 2: setting the initial b0 and b1, and logistic_regression call with 4 args
        #     were added to align with handwritten code and make it input to the next phase.
        # Train the logistic regression model on each partition
        b0, b1 = 0.0, 0.0  # Initial values of coefficients
        b0, b1 = logistic_regression(partitions_X_train[i], partitions_y_train[i], b0, b1)
        models.append((b0, b1))

    # Aggregate the models
    aggregated_b0 = np.mean([model[0] for model in models])
    aggregated_b1 = np.mean([model[1] for model in models])

    # Test the aggregated model on the test data
    y_pred, accuracy = evaluate(X_test, y_test, aggregated_b0, aggregated_b1)
    print(f"Accuracy = {accuracy}")
```

In this revised code, the main changes include:

1. Splitting the training data X_train and y_train into noClients partitions using np.array_split.
2. Performing federated learning on each partition by training a logistic regression model using the logistic_regression function.
3. Storing the trained models in a list called models.
4. Aggregating the models by calculating the mean of the coefficients (b0 and b1) from all the models.
5. Testing the aggregated model on the test data X_test and y_test using the evaluate function.

Questionnaire answers:

1. On a scale from 1 to 10, I would rate this assignment as a 6. It involves understanding the concept of federated learning and applying it to a logistic regression model. The task requires splitting the data into partitions, performing federated learning on each partition, and aggregating the models. While it may not be the most challenging assignment, it does require a solid understanding of the concepts and some programming skills.
2. The most important part of the context for completing the assignment was understanding the concept of federated learning and its application in a sequential code setting. This included understanding the steps involved in federated learning, such as data partitioning, training models on individual partitions, and aggregating the models. Additionally, the instructions provided the necessary details about the dataset and the specific steps required for the assignment.
3. The redundant part of the context was the initial description of the Social Network Ads (SNA) dataset and its features. While it provides some background information, it is not directly relevant to the task of rewriting the code into a federated sequential code. The important part of the context is already captured in the assignment description, which specifies the steps required for the code transformation.

4.2. Phase 3

The context for phase 3, see Table 3, comprises the revised general phase 3 guidelines, the description of the formats (i.e., signatures) of the callback functions, the important note on the formats of messages, the assignment for ChatGPT, the tweaked code from phase 2, and the placeholders for code completions.

Table 3. The context for phase 3 of the four-phases paradigm

"""
General phase 3 guidelines:
< Here was a copy of the text from Section 3.2 modified according to the note in Section 3.3. >

The client and server callback functions have the following format.
sfun(ldata, pdata, msg), returns ldataPrime.
cfun(pdata, msgs), returns ldataSecond.

```
Where:
sfun is the server callback function,
ldata the local data (typically a model),
pdata the private data (typically training data),
msg is the message carrying the server local data,
ldataPrime is the client updated local data,
cfun the client callback function,
msgs is the list of all the clients updated local data, and
ldataSecond is the server updated local data.

IMPORTANT NOTE:
In this example, local data for a node is a list with two coefficents [b0, b1],
whereas private data is a list [X_train, y_train].
Therefore, msg is a list (with two element),
whereas msgs is a list of lists (where each of these lists has two elements).

Assignment:
Using the phase 3 guidelines, rewrite the federated sequential code, which is given in the function
seq_horizontal_federated, into the federated sequential code with callbacks in the function
seq_horizontal_federated_with_callbacks and the supplementary client and server callback functions.
Important notes related to the assigment follow.
Note 1: Please do not introduce any new functions, just complete the three incomplete functions below!
Note 2: Assume that the initial local data (ldata) for the server is [0., 0.].
"""
```

< Here we provided the tweaked code from phase 2 see Table 2. >

```
def seq_horizontal_federated_with_callbacks():
    """ Please complete this function """

"""
Client callback function arguments format and usage.
Format: localData = [locb0, locb1], privateData = [X_train, y_train], msg = [srvb0, srvb1].
Usage: localData is not used, privateData contains client training data, and msg contains the server model
i.e., the initial coefficients.
"""
def fl_cent_client_processing(localData, privateData, msg):
    """ Please complete this function """

"""
Server callback function arguments amd usage.
Format: msgs = [[b00, b01], [b10, b11], ...]. In bij: i - partion (or client) index, j - coefficent index.
Usage: privateData is not used and msgs contains clients models.
"""
def fl_cent_server_processing(privateData, msgs):
    """ Please complete this function """
```

The tweaked code and the feedback for phase 3 are given in Table 4. The code generated by ChatGPT in the first attempt contained just a single logical bug and after we explained it to ChatGPT (see our dialog with ChatGPT in the explanation filed in Table 4), it generated correct code in the second attempt. It is worth mentioning that the code generated by ChatGPT is of better quality than the manually written code in [4] (the code looks nice and in the callback functions really condensed).

Both the explanation of the generated code and the questionnaire answers are sound (except, perhaps, the third answer) and human-like. ChatGPT rated the hardness of the assignment as 6. The third answer is somewhat strange because SNA dataset is not mentioned in the context for phase 3 (we intentionally skipped it, because we wanted to create a minimal context), so the third answer came unexpectedly to us. After giving it a second thought, it seems that either ChatGPT remembered the context for the previous phase (because it was conducted in the same session) or its comment is on possible generalization – consider these final words in the third answer: “as the transformation process would apply to other models and datasets as well”, see them in Table 4. However, both alternatives do not compromise the context minimality.

Table 4. The tweaked code and the feedback for phase 3 of the four-phases paradigm

< *To save space only the function seq_horizontal_federated_with_callbacks and the callback functions are shown.* >

```
def seq_horizontal_federated_with_callbacks():
    # Load the data
    data = pd.read_csv("Social_Network_Ads.csv")

    # Divide the data into training set and test set
    X_train, X_test, y_train, y_test = train_test_split(data['Age'], \
    data['Purchased'], test_size=0.20, random_state=42)

    # Split the training data into partitions
    noClients = 2  # Number of clients
    partitions_X_train = np.array_split(X_train, noClients)
    partitions_y_train = np.array_split(y_train, noClients)

    # Initialize the server model coefficients
    server_model = [0.0, 0.0]

    # Perform federated learning on each partition using callbacks
    client_models = []  # List to store client models
    for i in range(noClients):
        # Train the logistic regression model on each partition with the client callback
        client_model = fl_cent_client_processing([], [partitions_X_train[i], \
        partitions_y_train[i]], server_model)
        client_models.append(client_model)

    # Update the server model using the client models with the server callback
    server_model = fl_cent_server_processing([], client_models)

    # Get the final aggregated model coefficients
    aggregated_b0, aggregated_b1 = server_model
    print(f"The final model coefficients: aggregated_b0={aggregated_b0}, \
    aggregated_b1={aggregated_b1}")

    # Test the aggregated model on the test data
    y_pred, accuracy = evaluate(X_test, y_test, aggregated_b0, aggregated_b1)
    print(f"Accuracy = {accuracy}")

"""
Client callback function arguments format and usage.
Format: localData = [locb0, locb1], privateData = [X_train, y_train], msg = [srvb0, srvb1].
Usage: localData is not used, privateData contains client training data, and msg contains the server model,
i.e., the initial coefficients.
"""
def fl_cent_client_processing(localData, privateData, msg):
    b0, b1 = logistic_regression(privateData[0], privateData[1], msg[0], msg[1])
    return [b0, b1]
```

```
"""
Server callback function arguments and usage.
Format: msgs = [[b00, b01], [b10, b11], ...]. In bij: i - partition (or client) index, j - coefficient index.
Usage: privateData is not used and msgs contains clients' models.
"""
def fl_cent_server_processing(privateData, msgs):
    aggregated_b0 = np.mean([model[0] for model in msgs])
    aggregated_b1 = np.mean([model[1] for model in msgs])
    return [aggregated_b0, aggregated_b1]
```

< *ChatGPT:* > The seq_horizontal_federated_with_callbacks function performs federated learning using callbacks.
< *Authors to ChatGPT:* > You made just one mistake in the function seq_horizontal_federated_with_callbacks, the server should calculate the aggregated model after all the clients updated their models, so the sever callback function should not be called within the for loop but after that loop. Please could you correct that mistake and give the corrected function?
< *ChatGPT:* > Apologies for the mistake. Here's the corrected version of the function. In this updated version, the server callback function fl_cent_server_processing is called after all the client models have been collected in the client_models list.

Questionnaire answers:

1. On a scale from 1 to 10, I would rate this assignment as a 6. It involves understanding the concept of federated sequential code with callbacks and implementing the necessary changes in the provided code. While it may require some familiarity with the concepts of federated learning and callback functions, the guidelines provided in the assignment help in completing the task.
2. The most important part of the context for completing the assignment is the understanding of federated sequential code and the purpose of using callbacks in the context of federated learning. The guidelines provided in the assignment describe the steps involved in transforming the original code into code with callbacks, which helps in understanding the expected changes.
3. The redundant part of the context is the mention of the specific logistic regression model and the dataset ("Social_Network_Ads.csv"). While they provide a concrete example for the assignment, the main focus is on transforming the code into a federated sequential code with callbacks. The specific model and dataset used can be considered irrelevant to the main objective of the assignment, as the transformation process would apply to other models and datasets as well.

4.3. Phase 4

The context for phase 4, see Table 5, comprises the revised general phase 4 guidelines, the description of the formats of the callback functions, the important note on the formats of messages, the assignment for ChatGPT with additional notes (added in the second version of the context, which was also the final version), the tweaked code from phase 3, and the placeholders for the code completion.

Table 5. The context for phase 4 of the four-phases paradigm

"""
General phase 4 guidelines:
< *Here was a copy of the text from Section 3.2 modified according to the note in Section 3.3.* >

The PTB-FLA API includes the following functions that are defind in the class PtbFla.
PtbFla(noNodes, nodeId, flSrvId=0), no return value.
fl_centralized(sfun, cfun, ldata, pdata, noIters=1), returns ldataUpdated.
Where:
sfun is the server callback function,
cfun the client callback function,
ldata the local data (typically a model),
pdata the private data (typically training data),

noIters the number of iterations (default value is 1), and
ldataUpdated is the updated ldata.

The client and server callback functions have the following format.
< *The description is the same as in Table 3.* >

IMPORTANT NOTE:
< *The note is the same as in Table 3.* >

Assignment:
Using the phase 4 guidelines, rewrite the federated sequential code with callbacks, which is given in the function seq_horizontal_federated_with_callbacks, into the PTB-FLA code in the function main, which is launched as noNodes processes. Important notes related to the assignment follow.
Note 1: Please do not introduce any new functions, just complete the function main!
Note 2: PTB-FLA code should use the same client and server callback functions as the function seq_horizontal_federated_with_callbacks!
Note 3: Since there is one server and noClinets clients, noNodes = noClients + 1 i.e., noClients = noNodes - 1.
"""
< *Here we provided the tweaked code from phase 3 see Table 4.* >

```
"""
The function main is executed by noNodes processes (or nodes) and each process gets its own command line arguments.
Assume that for all nodes noNodes is 3 and flSrvId is 2,
and individual nodes are assigned nodeId values 0, 1, and 2, respectively.
"""
def main():
    # Process command line arguments
    noNodes = int( sys.argv[1] )
    nodeId = int( sys.argv[2] )
    flSrvId = int( sys.argv[3] )
    """ Please complete this function """
```

The tweaked code and the feedback for phase 4 are given in Table 6. This session was a bit disappointing and time consuming and was exactly the reason why we changed the process to manually tweak the code in case when it contains a small number of obvious bugs rather than continuing a seemingly endless and diverging sessions.

In this case, it took us six attempts to guide ChatGPT with additional explanations and notes from our side so that it produces the correct code. However, even the code we got in the sixth attempt that at glance looked like right, had a single bug, which we discovered when executing the code, and which we decided to manually tweak to get it right (see TWEAK 2 in the code in Table 6). Overall, we may state that this effort paid off as the final tweaked code is of better quality than the manually written code in [4] (because it is generalized for any number of clients and has better usage of APIs).

Both the explanation of the generated code and the questionnaire answers are sound and human-like. ChatGPT rated the hardness of the assignment as 7. The third answer needs to be explained. Although the phase 3 code already comprised the client and server callback functions, ChatGPT introduced new callback functions (server_callback and client_callback) and used them instead of already provided functions (and this was not the only problem in the generated code). Perhaps it introduced these new callback functions because it remembered that it was asked is previous assignments to create callback functions. No matter what the reason was, these new functions that ChatGPT introduced were redundant, and in its third answer it admitted it, however this does not compromise the context minimality.

Table 6. The tweaked code and the feedback for phase 4 of the four-phases paradigm

< *To save space only the function main is shown, and only the most important TWEAK.* >

```
"""
The function main is executed by noNodes processes (or nodes) and each process gets its own command line
arguments.
Assume that for all nodes noNodes is 3 and flSrvId is 2,
and individual nodes are assigned nodeId values 0, 1, and 2, respectively.
"""
def main():
    # Process command line arguments
    noNodes = int(sys.argv[1])
    nodeId = int(sys.argv[2])
    flSrvId = int(sys.argv[3])

    # Load the data
    data = pd.read_csv("Social_Network_Ads.csv")

    # Divide the data into training set and test set
     X_train, X_test, y_train, y_test = train_test_split(data['Age'], \
     data['Purchased'], test_size=0.20, random_state=42)

    # Split the training data into partitions
    noClients = noNodes - 1  # Number of clients
    partitions_X_train = np.array_split(X_train, noClients)
    partitions_y_train = np.array_split(y_train, noClients)

    # Create an instance of PtbFla
    ptb = PtbFla(noNodes, nodeId, flSrvId)

    # Prepare local data and private data for the current node
    localData = [0.0, 0.0]  # Initial local data (model coefficients)

     # TWEAK 2: the server doesn't have its private data, so the next 2 lines were -
     # added and the 3rd is indented
    privateData = None
    if nodeId != flSrvId:
        privateData = [partitions_X_train[nodeId], partitions_y_train[nodeId]]  # Private data

    # Perform federated learning using PTB-FLA
     updated_ldata = ptb.fl_centralized(fl_cent_server_processing, \
     fl_cent_client_processing, localData, privateData)

    # Get the final aggregated model coefficients
    aggregated_b0, aggregated_b1 = updated_ldata
     print(f"The final model coefficients: \
     aggregated_b0={aggregated_b0}, aggregated_b1={aggregated_b1}")

    # Test the aggregated model on the test data
    y_pred, accuracy = evaluate(X_test, y_test, aggregated_b0, aggregated_b1)
    print(f"Accuracy = {accuracy}")

    # TWEAK 3: this code was added to make it comparable with handwritten code
    # Must be
    if nodeId == flSrvId:
        refbs = seq_horizontal_federated_with_callbacks()
        assert refbs[2][0] == aggregated_b0 and refbs[2][1] == aggregated_b1, \
        "b0 and b1 must be equal to ref_b0 and ref_b1, respectively!"

    # Shutdown
    del ptb
    pkey = input('press any key to continue...')
```

< *Below is the ChatGPT explanation after the first attempt.* >

I have updated the main() function to include the PTB-FLA code. The client and server callback functions remain the same. The PtbFla instance ptb is created, and the local and private data are prepared. The server and client callback functions are defined as inner functions within main(). Finally, the callback function calls are replaced with the fl_centralized function call on the ptb object.

< *To save space our dialog with ChatGPT in the next five attempts is not shown here.* >

Questionnaire answers:

1. I would rate this assignment a 7 on the difficulty scale. The assignment involved understanding the concept of federated learning, callback functions, and their integration into the PTB-FLA code. It required careful reading and interpretation of the provided context to ensure the correct implementation.
2. The most important part of the context was the description of the PTB-FLA API functions, namely PtbFla and fl_centralized. Understanding their input arguments, return values, and how they should be used was crucial in completing the assignment.
3. The redundant part of the context was the introduction of new callback functions (server_callback and client_callback). These functions were not required and caused confusion. It was sufficient to use the provided callback functions fl_cent_server_processing and fl_cent_client_processing directly in the fl_centralized function call.

5. Two-Phases Paradigm Validation

This section presents the validation of the two-phases paradigm adapted for ChatGPT. As phase 1 is conducted manually, we immediately proceed to phase 2.

The context for phase 2, see Table 7, comprises the revised general guidelines for the two-phases paradigm, the description of the formats of the callback functions, the important note on the formats of messages, the assignment for ChatGPT with additional notes (added in the second version of the context, which was also the final version), the referent code from phase 1, and the placeholders for the code completion.

Table 7. The context for phase 2 of the two-phases paradigm

"""

Revised general guidelines for the two-phases paradigm:
< *Here was a copy of the text from Section 3.3.* >

The PTB-FLA API includes the following functions that are defined in the class PtbFla.
< *The description is the same as in Table 5.* >

The client and server callback functions have the following format.
< *The description is the same as in Table 3.* >
IMPORTANT NOTE:
< *The note is the same as in Table 3.* >

Assignment:
Rewrite the referent sequential code, which is given in the function seq_base_case, into the PTB-FLA code in the function main and supplementary client and server callback functions in the functions fl_cent_client_processing and fl_cent_server_processing, respectively. The PTB-FLA code is launched as

```
noNodes processes. Important notes related to the assignment follow.
Note 1: There is one server and noClinets clients, noNodes = noClients + 1 i.e., noClients = noNodes - 1.
Note 2: The server does not have its own private data.
"""
< Here we provided the referent sequential code from phase 1, see Table 1. >

< The prolog comment for the function below is the same as in Table 3. >
def fl_cent_client_processing(localData, privateData, msg):
    """ Please complete this function """

< The prolog comment for the function below is the same as in Table 3. >
def fl_cent_server_processing(privateData, msgs):
    """ Please complete this function """

< The prolog comment for the function below is the same as in Table 5. >
def main():
    # Process command line arguments
    noNodes = int( sys.argv[1] )
    nodeId = int( sys.argv[2] )
    flSrvId = int( sys.argv[3] )
    """ Please complete this function """
```

The tweaked code and the feedback for phase 2 are given in Table 8. This session started great as ChatGPT immediately produced perfect callback functions and the function main with just a single bug – it “forgot” that FL server doesn’t have its own private data. This code was easily tweakable, and we consider this as a great achievement – it justifies both the paradigm/context effectiveness and comes close to an ideal solution where a developer gives the context and the input referent sequential code and gets back (almost) the perfect target PTB-FLA code.

Now because the code was easily tweakable, we wanted to see whether ChatGPT could tweak it on its own, so we added two additional notes related to the assignment (in the second version of the context in Table 7). Unfortunately, ChatGPT unexpectedly produced the code that was more far away from the target code. So, we entered a rather tedious session and finally in the sixth attempt ChatGPT produced the correct function main. Overall, we may again state that this effort paid off as the final tweaked code is of better quality than the manually written code in [4].

The dialog with ChatGPT was cumbersome but the questionnaire answers are sound (except the third one) and human-like. ChatGPT rated the hardness of the assignment as 7. The third answer came unexpectedly, because in this session ChatGPT did not introduce new callback functions at all, obviously it mixed up this session with the previous one however this does not compromise the context minimality.

Table 8. The tweaked code and the feedback for phase 2 of the two-phases paradigm

```
< To save space only the function main and the callback functions are shown. >

def fl_cent_client_processing(localData, privateData, msg):
    b0, b1 = logistic_regression(privateData[0], privateData[1], msg[0], msg[1])
    return [b0, b1]

def fl_cent_server_processing(privateData, msgs):
    aggregated_b0 = np.mean([model[0] for model in msgs])
    aggregated_b1 = np.mean([model[1] for model in msgs])
    return [aggregated_b0, aggregated_b1]
```

```
def main():
    # Process command line arguments
    noNodes = int(sys.argv[1])
    nodeId = int(sys.argv[2])
    flSrvId = int(sys.argv[3])
    print(noNodes, nodeId, flSrvId)

    # Load the data
    data = pd.read_csv("Social_Network_Ads.csv")

    # Divide the data to training set and test set. For reproducible set random_state=42.
     X_train, X_test, y_train, y_test = train_test_split(data['Age'], \
     data['Purchased'], test_size=0.20, random_state=42)

    # Split the training data into partitions
    noClients = noNodes - 1  # Number of clients
    partitions_X_train = np.array_split(X_train, noClients)
    partitions_y_train = np.array_split(y_train, noClients)

    # Initialize the PTB-FLA object
    ptb = PtbFla(noNodes, nodeId, flSrvId)

    # Prepare local data and private data for the current instance
    localData = [0.0, 0.0]  # Initial values of coefficients
     privateData = None if nodeId == flSrvId \
     else [partitions_X_train[nodeId], partitions_y_train[nodeId]]

    # Perform federated learning using PTB-FLA
     updated_ldata = ptb.fl_centralized(fl_cent_server_processing, \
     fl_cent_client_processing, localData, privateData)

    # Get the updated model coefficients
    aggregated_b0, aggregated_b1 = updated_ldata

    # Print the final model coefficients
     print(f"The final model coefficients: \
     aggregated_b0={aggregated_b0}, aggregated_b1={aggregated_b1}")

    # Test the aggregated model on the test data
    y_pred, accuracy = evaluate(X_test, y_test, aggregated_b0, aggregated_b1)
    print(f"Accuracy = {accuracy}")

    # Must be (TWEAK 2)
    if nodeId == flSrvId:
        refbs = seq_horizontal_federated_with_callbacks()
        assert refbs[2][0] == aggregated_b0 and refbs[2][1] == aggregated_b1, \
        "b0 and b1 must be equal to ref_b0 and ref_b1, respectively!"

    # Shutdown
    del ptb
    pkey = input('press any key to continue...')
```

< ChatGPT: > Please complete the fl_cent_client_processing and fl_cent_server_processing functions as per the assignment instructions.

< Authors to ChatGPT: > Please could you also complete the callback functions.

< ChatGPT (after flashing out the functions): > These callback functions implement the logistic regression training on each client partition and the aggregation of the models on the server side, respectively.

< To save space our dialog with ChatGPT in the next five attempts is not shown here. >

Questionnaire answers:

1. I would rate this assignment a 7 on the difficulty scale. It involves understanding and applying the concepts of federated learning, PTB-FLA, and callback functions to transform a sequential code into a distributed PTB-FLA code. Additionally, there were some details and modifications required to address specific issues and bugs.
2. The most important part of the context was understanding the PTB-FLA framework, including the functions PtbFla, fl_centralized, and the roles of the server and client callback functions. Additionally, understanding the structure and format of the local data and private data used in the example was crucial for completing the assignment.
3. The redundant part of the context was the introduction of new callback functions (server_callback and client_callback) in the earlier versions of the code. These functions were not required, and the provided callback functions (fl_cent_server_processing and fl_cent_client_processing) should have been used instead.

6. Adapted Development Paradigms Performance

In this section, we report on the adapted development paradigms performance in terms of human labour (in working hours) and size of ChatGPT context (in number of characters with spaces) needed to develop the logistic regression PTB-FLA code. We would like to emphasize that data on human labour should be treated as rough estimates because it is based on our freeform notes in private diaries. Data on ChatGPT context size are exact and we got them by the text editor.

Table 9 is organized as follows, the left part relates to the human labor and the right part relates to the ChatGP contexts size, the top part contains raw (input) data, and the bottom part contains calculated (output) data; some of the fields and not applicable (but this should be obvious, so we skip explaining these exceptions). There are two types of calculated data: (i) the working speed up that is defined as the ratio of working hours, and (ii) the ChatGTP context size reduction that is defined as the ratio of ChatGPT context sizes.

Table 9. Adapted paradigms performance data

	Human labour [h]			Context size [ch with spaces]	
Phase	4-Ph human	4-Ph GPT	2-Ph GPT	4-Ph GPT	2-Ph GPT
Phase 2	8	4	4	2462	2685
Phase 3	12	4		2593	
Phase 4	4	4		2334	
Total	24	12	4	7389	2685
	Speed up:	2	6	Reduction:	2.75

The calculated data in the bottom raw reveal encouraging results, the adapted 4-phases development paradigm for ChatGPT achieved the speed up of 2 times over the original 4-phases development paradigm for humans, whereas the adapted 2-phase development paradigm for ChatGPT achieved the speed up of 6 times over the original 4-phase development paradigm for humans. However, one should keep in mind that these impressive results are based on rough estimates of working hours.

Regarding the ChatGPT context sizes, the calculated data in the bottom row of Table 9 reveal that the adapted 2-phases development paradigm for ChatGPT achieves the context size reduction of 2.75 times over the adapted 4-phases development paradigm for ChatGPT i.e., it makes using ChatGPT 2.75 times cheaper.

7. Related Work

To make it easier to follow, we start this section with a taxonomy of the features and a classification of related works by the features, see Table 10. The taxonomy comprises the following features: (1) low code and no code paradigms, (2) code generation, (3) paradigm adaptations for ChatGPT, and (4) other (references not directly comparable with this paper). For references comparable with this paper (not in “Other”), we discuss their weakness and how this paper solves this weakness.

Table 10. Related works versus the taxonomy features

Paper	Low code No code	Code Generation	Paradigm Adaptatio n	Other (Comment)
This	Yes	Yes	Yes	
1				Yes (Umbrella EU project)
2				Yes (Target FL framework)
3				Yes (Logistic regression use case inputs)
4	Yes			
5	Yes	Yes		
6		Yes		
7				Yes (Text-bot usage modes by P.J. Denning)
8		Yes		
9		Yes		
10				Yes (Status of the ChatGPT popularity)
11				Yes (Overview of low code program. models)
12	Yes	Yes		
13	Yes			
14				Yes (Discussion on ChatGPT privacy issues)
15				Yes (Well-known ChatGPT hallucinations)
16				Yes (Strategies for ChatGPT hallucinations)
17				Yes (Strategies for ChatGPT hallucinations)
18				Yes (Strategies for ChatGPT hallucinations)
19				Yes (Assessment of program. assignments)
20				Yes (Abstract state machines, Fed. Networks)
21				Yes (Discussion thread summarising systems)
22				Yes (Decision-making for mental health)

Not long ago, various systems and frameworks have appeared that can automate code generation e.g., Amazon’s CodeWhisperer, GitHub’s Copilot, IBM’s Project Wisdom, etc. Although it is unlikely AI will eliminate jobs for developers anytime soon, it is poised to revolutionize the way software is created. For instance, the OpenAI Codex generates software in many popular programming languages, including Python, Perl, Ruby, and PHP [6]. Discussion: from the three relevant features, [6] only features code generation, whereas this paper additionally features low code and paradigm adaptation.

P.J. Denning has seen three modes of interaction with text-bots: jump-start, provocateur, and appropriator. In this paper we use the *jump-start mode*, and this term means the bot helps the human complete a job faster e.g., a programmer asks a text-bot to generate initial code, then reviews and edits to make it error free [7]. This reference is classified as “Other” in Table 10.

Sarkar et al. explored how programming with large language models is similar and different from the prior conceptualisations of programmer assistance and found that while LLM-assisted programming shares some properties of compilation, pair programming, and programming via search and reuse, there are some fundamental differences [8]. Discussion: from the three relevant features, [8] only features code generation, whereas this paper additionally features low code and paradigm adaptation.

Bird et al. conducted three studies to understand how developers use GitHub Copilot: (i) an analysis of forum discussions from early Copilot users, (ii) a case study of Python developers using Copilot for the first time, and (iii) a large-scale survey of Copilot users to understand its impact on productivity [9]. Discussion: from the three relevant features, [9] only features code generation, whereas this paper additionally features low code and paradigm adaptation.

However, it seems that ChatGPT is still generally the most popular LLM with its 180 million users as of December 2023 [10] and therefore we decided to use it for validation of our adapted development paradigms in this paper. This reference is classified as "Other" in Table 10.

Martin Hirzel in his recent paper on low-code programming models [11] discussed the following three representative techniques for low-code programming: visual programming languages (VPLs), programming by demonstration (PBD), and programming by natural language (PBNL), and all three languages use some domain-specific language (DSL) that is typically embedded in a general-purpose programming language (like Python). In the context of [11], the approach used in this paper is close to low-code approach based on PBNL. This reference is classified as "Other" in Table 10.

Busch et al. in [12] extended their language driven engineering environment (LDE) for low-code/no-code development via dedicated DSLs to also support specification in natural language with the following benefits: (i) the tasks to be solved via LLM-based code generation can be tailored in size and conceptual complexity and (ii) the overall heterogeneously constructed system can be directly validated at system level using automata learning and model analysis. The main difference between this paper and [12] is that in this paper the final PTB-FLA code is correct by construction and is semantically equivalent to the referent sequential code (both produce the same results). Discussion: from the three relevant features, [12] features low-code/no-code and code generation, whereas this paper additionally features paradigm adaptation.

An interesting approach to low-code implementation and no-code reuse is given in [13], where authors use Pyrus, a web-based, low-code/no-code graphical modelling environment for ML and data analytics applications, to solve the challenge of re-using AI/ML models that do not address data with imbalanced class by implementing ML algorithms in Python that do the re-balancing and then using these algorithms and the original models in the data processing pipelines in Pyrus. At present, this paper uses a low-code approach, and extending it by no-code reuse is an interesting direction for the future work. Discussion: of the three relevant features, [13] features only low-code/no-code, whereas this paper additionally features code generation and paradigm adaptation.

Regarding the discussion on data privacy issues with ChatGPT (issues with open code, who is owning the data when query is sent and what can OpenAI do with the data, etc.) [14], we would like to clarify that in our development process (see Section 4) ChatGPT never receives the private data (i.e., the data that is used to train the model), it only receives the program code. Therefore, in this respect, our development process is completely aligned with the idea of federated learning. Other issues covered in [14] are out of the scope of this paper. This reference is classified as “Other” in Table 10.

In this paper, we accepted the well-known ChatGPT hallucinations [15] as a kind of tool imperfectness, because they were rarely happening in our experiments, and therefore we did not try to mitigate or counter them. Meanwhile, some strategies were proposed to tackle this issue [16-18], and we plan to conduct research in this direction in our future work. These references are classified as “Other” in Table 10.

At the end of this section, we would like to shortly mention the four interesting papers that are orthogonal to this paper and relevant to our longer-term future work.

First, Paiva et al. in [19] provided a thorough comparison of the most widespread semantic graph representations for the automated assessment of programming assignments. This and similar techniques and tools may be used to assess the program codes received from the ChatGPT. This reference is classified as “Other” in Table 10.

Second, Ghilezan et al. in [20] designed the architecture of the overlay network federation using abstract state machines, and they showed that the federation can be used as a formal model of forward contact tracing in the context of Covid-19 pandemic. Abstract state machines or similar formal methods perhaps may be used to formally verify the program codes received from ChatGPT. This reference is classified as “Other” in Table 10.

Third, Khan et al. in [21] we introduce the long short-term memory (LSTM) auto-encoder as a sentence embedding technique to improve the performance of the discussion thread summarizing system (DTS), which creates a candid view of the entire discussion of a query. DTS and similar techniques and tools may be used to independently analyze series of longer sessions with ChatGPT when developing more complex PTB-FLA algorithms. This reference is classified as “Other” in Table 10.

Fourth, Yu and Lin in [22] provided a scientific decision-making basis for mental health education in colleges and universities by extracting the classification rules from the optimal decision tree model. Similar techniques and tools may be used to analyze ChatGPT responses. This reference is classified as “Other” in Table 10.

8. Conclusion and Future Work

In this paper, we present the four-phase and the two-phase PTB-FLA development paradigms adapted to be used by ChatGPT. The former paradigm comprises the four phases: (1) the sequential code, (2) the federated sequential code, (3) the federated sequential code with callbacks, and (4) the PTB-FLA code, whereas the latter comprises the two phases: (1) the sequential code and (2) the PTB-FLA code. The PTB-FLA development paradigms were adapted and validated using the logistic regression case study, where centralized FL algorithms were developed.

The main original contributions of this paper are: (1) the PTB-FLA development paradigms adapted for ChatGPT, (2) our contexts (i.e., prompts) for the individual phases, the tweaked program codes, and the questionnaire, (3) ChatGPT generated program codes, the explanations, and the questionnaire answers, and (4) the report on adapted development paradigms performance in terms of human labour and ChatGPT context size.

The main advantages of all the PTB-FLA development paradigms over ad hoc approach are: (1) developed FLAs are correct by construction (they produce the same results as their referent sequential counterparts) and (2) the development of FLAs is easier than by ad hoc approach.

The main advantage of the two PTB-FLA development paradigms adapted for ChatGPT over the original development paradigm for humans is that they are faster and cheaper, because they require less human labour, and it is reasonable to assume that ChatGPT costs will be negligible compared to human labor (at this time it is for free).

When comparing the two PTB-FLA development paradigms adapted for ChatGPT, the two-phase paradigm: (1) aids easier development that is close to an ideal solution where a developer gives the context and the input referent sequential code and gets back (almost) the perfect target PTB-FLA code, and (2) it is cheaper to use as it requires smaller ChatGPT context.

The main shortcoming of all the PTB-FLA development paradigms (both for humans and ChatGPT) is that they still have not been tested during the development of some real applications.

The main limitation of this paper is that only ChatGPT was used in the experimental validation.

The main directions of the future work are: (1) use the PTB-FLA development paradigms to develop other more complex FLAs and real applications, (2) continue improving the PTB-FLA development paradigms based on the feedback from developing more complex FLAs and real applications, (3) research adapting and specifying the PTB-FLA development paradigms for other emerging AI tools e.g., other general purpose LLMs like Bard, as well as other models that are specialized for coding, such as Code LLama, StarCoder, WizardCoder, and Safurai 001, etc., and (4) research techniques to mitigate ChatGPT hallucinations.

Acknowledgement. Funded by the European Union (TaRDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.
Authors are grateful to anonymous reviewers for their valuable comments and suggestions.

References

1. TaRDIS: Trustworthy and Resilient Decentralised Intelligence For Edge Systems. [Online]. Available: https://www.project-tardis.eu/ (current March 2024)
2. Popovic, M., Popovic, M., Kastelan, I., Djukic, M., Ghilezan, S.: A Simple Python Testbed for Federated Learning Algorithms. In Proceedings of the 2023 Zooming Innovation in Consumer Technologies Conference. IEEE Xplore, New Jersey, USA, 148-153. (2023)
3. Logistic Regression. [Online]. Available: https://colab.research.google.com/drive/1qmdfU8tzZ08D3O84qaD11Ffl9YuNUvlD (current December 2023)
4. Popovic, M., Popovic, M., Kastelan, I., Djukic, M., Basicevic, I.: A Federated Learning Algorithms Development Paradigm. In: Kofron, J., Margaria, T., Seceleanu, C. (eds.): Engineering of Computer-Based Systems. Lecture Notes in Computer Science, Vol. 14390. Springer, Cham, 26–41. (2024)

5. Popovic, M., Popovic, M., Kastelan, I., Djukic, M., Basicevic, I.: Developing Elementary Federated Learning Algorithms Leveraging the ChatGPT (2023). In Proceedings of the 31st Telecommunications Forum (TELFOR). IEEE Xplore, New Jersey, USA, 1-4. (2023)
6. Greengard, S.: AI Rewrites Coding. Communications of the ACM, Vol. 66, No. 4, 12–14. (2023)
7. Denning, P.J.: The Profession of IT: Can Generative AI Bots Be Trusted? Communications of the ACM, Vol. 66, No. 6, 24–27. (2023)
8. Sarkar, A., Gordon, A.D., Negreanu, C., Poelitz, C., Ragavan, S.S., Zorn, B.: What is it like to program with artificial intelligence? (2023). [Online]. Available: https://arxiv.org/abs/2208.06213 (current March 2024)
9. Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., Gazit, I.: Taking Flight with CopilotTaking Flight with Copilot. Communications of the ACM, Vol. 66, No. 6, 56–62. (2023)
10. Duarte, F.: Number of ChatGPT Users (2023). [Online]. Available: https://explodingtopics.com/blog/chatgpt-users (current March 2024)
11. Hirzel, M.: Low-Code Programming Models. Communications of the ACM, Vol. 66, No. 10, 76–85. (2023)
12. Busch, D., Bainczyk, A., Steffen. B.: Towards LLM-Based System Migration in Language-Driven Engineering. In: Kofron, J., Margaria, T., Seceleanu, C. (eds.): Engineering of Computer-Based Systems. Lecture Notes in Computer Science, Vol. 14390. Springer, Cham, 191–200. (2024)
13. Singh, A., Minguett, O.: IDPP: Imbalanced Datasets Pipelines in Pyrus. In: Kofron, J., Margaria, T., Seceleanu, C. (eds.): Engineering of Computer-Based Systems. Lecture Notes in Computer Science, Vol. 14390. Springer, Cham, 60–69. (2024)
14. Liesenfeld, A., Lopez, A., Dingemanse, M.: Opening up ChatGPT: Tracking openness, transparency, and accountability in instruction-tuned text generators. In Proceedings of the 5th international conference on conversational user interfaces. ACM, New York, NY, Article No. 47, 1-6 (2023)
15. Rawte, V., Sheth, A., Das, A.: A Survey of Hallucination in Large Foundation Models. (2023). [Online]. Available: https://arxiv.org/abs/2309.05922 (current March 2024)
16. Tonmoy, S. M., Zaman, S. M., Jain, V., Rani, A., Rawte, V., Chadha, A., Das, A.: A Comprehensive Survey of Hallucination Mitigation Techniques in Large Language Models. (2024). [Online]. Available: https://arxiv.org/abs/2401.01313 (current March 2024)
17. Rawte, V., Priya, P., Tonmoy, S. M., Zaman, S. M., Sheth, A., Das, A.: Exploring the Relationship between LLM Hallucinations and Prompt Linguistic Nuances: Readability, Formality, and Concreteness. (2023). [Online]. Available: https://arxiv.org/abs/2309.11064 (current March 2024)
18. Martino, A., Iannelli, M., Truong, C.: Knowledge Injection to Counter Large Language Model (LLM) Hallucination. In: Pesquita, C., et al. (eds.): The Semantic Web: ESWC 2023 Satellite Events. Lecture Notes in Computer Science, Vol. 13998. Springer, Cham, 182-185. (2023).
19. Ghilezan, S., Kašterović, S., Liquori, L., Marinković, B., Ognjanović, Z., Stefanović, T.: Federating Digital Contact Tracing using Structured Overlay Networks. Computer Science and Information Systems, Vol. 19, No. 3, 1261-1282. (2022)
20. Paiva, J. C., Leal, J. P., Figueira, Á.: Comparing Semantic Graph Representations of Source Code: The Case of Automatic Feedback on Programming Assignments. Computer Science and Information Systems, Vol. 21, No. 1, 117–142. (2024)
21. Khan, A. W., Al-Obeidat, F., Khalid, A., Amin, A., Moreira, F.: Sentence embedding approach using LSTM auto-encoder for discussion threads summarization. Computer Science and Information Systems, Vol. 20, No. 4. (2023)
22. Yu, J., Lin, J.: Data mining technology in the analysis of college students' psychological problems. Computer Science and Information Systems, Vol. 19, No. 3, 1583-1596. (2022)

Miroslav Popovic received his Dipl. Eng., M.Sc., and Ph.D. degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 1984, 1988 and 1990, respectively. He has been Full Professor at the University of Novi Sad since 2002. Currently he is giving courses on parallel and systems programming and on computer networks. His research interests are engineering of computer-based systems and intelligent distributed systems. He has authored or co-authored more than 165 scientific papers, and the book Communication protocol engineering, 2nd Edition (CRC Press).

Marko Popovic received his B.Sc., M.Sc., and PhD degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 2015, 2017, and 2020, respectively. Currently he is Scientific Researcher affiliated with the RT-RK Institute of Computer Based Systems, Novi Sad, Serbia. His research interests are in the areas of engineering of computer-based systems and intelligent distributed systems. He has authored or co-authored more than 23 scientific papers.

Ivan Kastelan received Dipl. Eng. M.Sc, and PhD degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 2008, 2009, and 2014, respectively. Currently he is Associate Professor at the University of Novi Sad, teaching courses on algorithms and computer engineering. His research interests include digital and computer systems design, design of algorithms, and engineering education. He is the author of one textbook, more than 90 scientific papers, and 13 patents.

Miodrag Djukic received Dipl. Eng. and M.Sc degree in 2008., from the Faculty of Technical Sciences, University of Novi Sad, Serbia, and PhD degree in 2015 from the same faculty. Currently he is an Assistant Professor at the University of Novi Sad, teaching courses on compilers and programming. His research interests include compilers, software tools, formal methods, and engineering education. He has authored or co-authored more than 50 scientific papers.

Ilija Basicevic received Dipl. Eng. M.Sc, and PhD degrees from the Faculty of Technical Sciences, University of Novi Sad, Serbia, in 1998, 2001, and 2009, respectively. Currently he is Full Professor at the University of Novi Sad, teaching courses on computer networks. His research interests are in the areas of Internet protocols and network security. He has authored or co-authored more than 100 scientific papers and one textbook.

Received: December 24, 2023; Accepted: May 28, 2024.