Capstone Project: MUNNNToolbox

Project Design Report

Murad Shahsuvarov

Jehad Abualhassan

Bahador Najivandzadeh

Prabin Kumar Shrestha

Daniel Manasseh

Hussein El Samad

Fall 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's data-driven world, the application of neural networks has become indispensable across various domains, from image recognition to natural language processing. However, harnessing the full potential of neural networks often requires a deep understanding of their architectures, which can be daunting for many users. This is where our project steps in - to bridge the gap between neural network designs and user-friendly accessibility.

Our project aims to create a Neural Networks Toolbox, a user-friendly platform that empowers individuals with little to no deep learning expertise to design, customize, and train neural network architectures effortlessly. To accomplish this ambitious goal, our dedicated team is divided into two distinct but interconnected parts: the front-end team and the back-end team.

The front-end team is tasked with crafting an intuitive and user-friendly interface that empowers users to effortlessly design neural network structures. This interface will facilitate the selection of various neural network components and provide real-time visualization. Within this interface, users will have the flexibility to input and manage multiple datasets, as well as enabling users to design and customize multiple neural network architectures.

On the other side, the back-end team is responsible for the backbone of our toolbox. They are developing a set of powerful APIs and services that abstract away the complexities of neural network training. They will also implement an efficient data management system.

1.1 Activity Diagram Description:

The user interface toolbox for our neural network will be divided into three tabs, each serving distinct but interconnected functions. The user will be able to transition from one tab to another however he wills. The three tabs will be as follows:

1) Dataset Section tab:

This tab involves dataset selection, the display of existing datasets, and the option to upload a new dataset. If the user chooses to upload a new dataset, the system will validate the input file in CSV format. If validation fails, it will display an error and return the user to the upload part.

Start Point: The user selects the "Dataset" tab.

Action: The system displays a list of existing datasets.

Action: The system shows a button labeled "Upload New Dataset."

Decision Point: The user can choose to either select an existing dataset or proceed to upload a new one.

If the user selects an existing dataset:

End Point: The system loads the selected dataset for further operations.

If the user selects "Upload New Dataset":

Action: The user is presented with an interface to upload a dataset in CSV format.

Action: The user selects a CSV file for upload.

Action: The system validates the uploaded file to ensure it is in the correct CSV format.

Decision Point: If the file is valid:

Action: The system proceeds to process the dataset.

End Point: The user can continue with other actions on the uploaded dataset.

If the file is not valid:

Action: The system displays an error message indicating that the file format is incorrect.

Transition: The user is directed back to the file upload interface to try again.

End Point: This part ends when the user has either selected an existing dataset or successfully uploaded and validated a dataset for further use.

2) Architecture tab:

In the second tab, known as the "Architecture" section, the user can select existing neural

network architectures or design custom ones. Once the user have designed an architecture he'll be able to save it. The system will then validate it and if there appears to be an error, it will display that.

Start Point: The user selects the "Architecture" tab.

Action: The system displays a list of existing neural network architectures.

Action: The system shows a button labeled "Design New Architecture."

Decision Point: The user can choose to either select an existing architecture or proceed to design a new one.

If the user selects an existing architecture:

End Point: The system loads the selected architecture for further operations.

If the user selects "Design New Architecture":

Action: The user is presented with a visual board for designing a custom neural network architecture.

Action: The user designs the architecture by adding layers, nodes, connections, etc.

Action: The system allows the user to save the designed architecture.

Action: The system validates the designed architecture to ensure it meets specified criteria.

Decision Point: If the architecture is valid:

Action: The system saves the custom architecture.

End Point: The user can continue with other actions on the designed architecture.

If the architecture is not valid:

Action: The system displays an error message indicating that the architecture has errors.

Transition: The user is directed back to the architecture design interface to correct the errors.

End Point: This part ends when the user has either selected an existing architecture or successfully designed and validated a custom architecture for further use.

3) Training tab:

In this section, the user can select datasets, architectures, specify inputs and outputs, and initiate training. Subsequently, the user can use the trained model to make predictions and download the dataset containing predictions.

Start Point: The user selects the "Training" tab.

Action: The system displays dropdowns to select a dataset and an architecture from the database.

Action: The system shows form fields for specifying inputs and outputs.

Action: The user selects a dataset, an architecture, and specifies inputs and outputs.

Action: The user initiates the training process.

Decision Point: The system validates if the selected dataset is compatible with the chosen architecture and if model inputs are valid.

If validation fails:

Action: The system displays an error message.

Transition: The user is taken back to the input specification step to correct errors.

If validation is successful:

Action: The system begins the training process.

Action: The user waits while the system trains the model.

Action: Upon completion of training, the system sends a notification with a link to use the trained model.

Action: The user clicks the link received after training completion.

Action: The system displays dropdowns to select a trained model and a dataset.

Action: The system shows form fields for specifying inputs and outputs for prediction.

Action: The user selects a model, a dataset, and specifies inputs and outputs for prediction.

Decision Point: The system validates the selected model, dataset, and input/output specifications.

If validation fails:

Action: The system displays an error message.

Transition: The user is taken back to the input specification step to correct errors.

If validation is successful:

Action: The system performs predictions on the dataset based on the selected model and Input/output specifications.

Action: The system returns the dataset with predictions.

Action: The user can download the dataset, which includes prediction results.

End Point:

This part ends when the user successfully completes the entire process, from initiating training to obtaining and downloading the dataset with prediction results

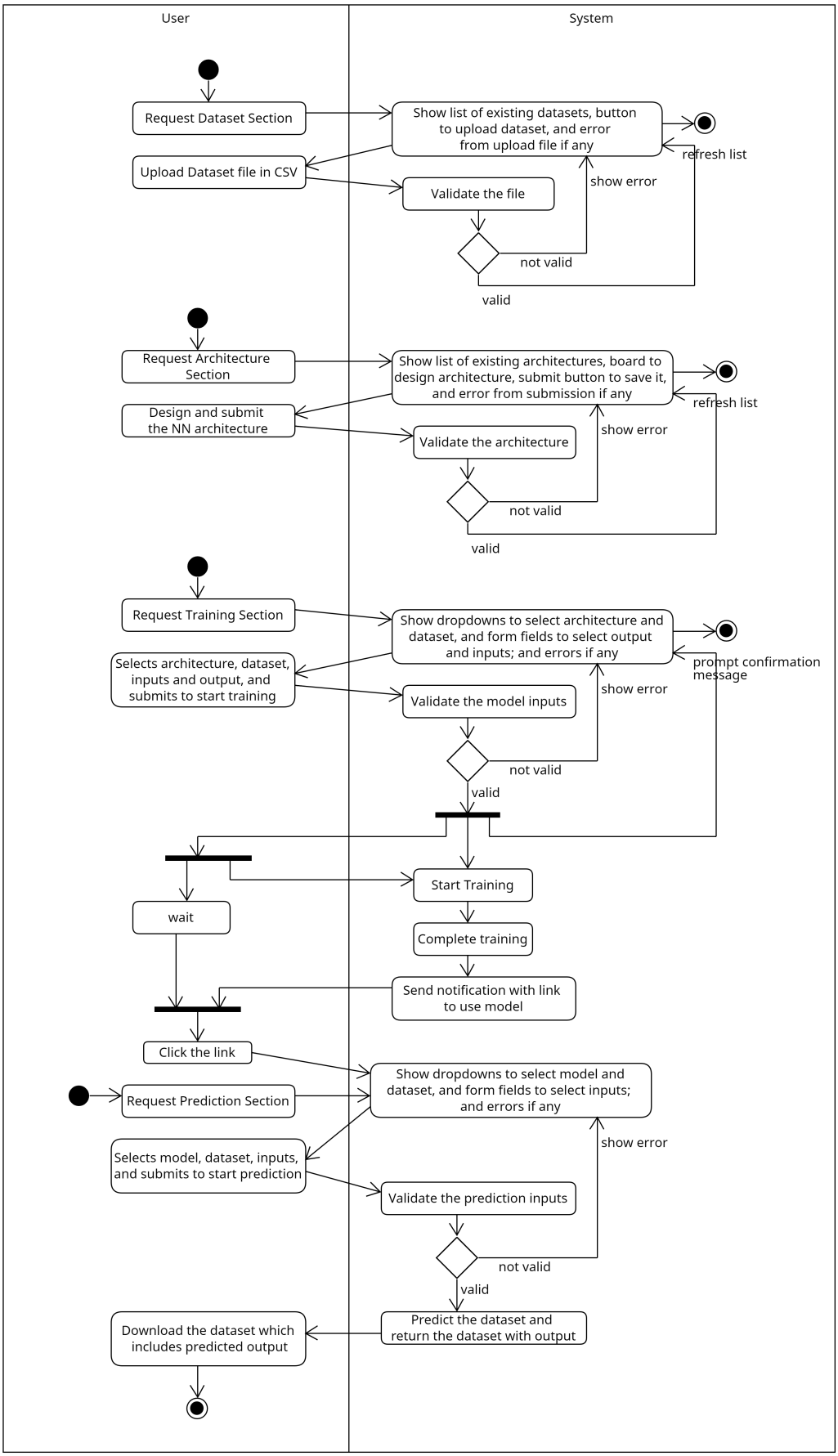

Figure 1.1: MUNNNToolbox Workflow Activity diagram

Chapter 2

Backend Design

2.1 Introduction

The MUNNNToolbox project employs a meticulously designed backend system architected using a microservices approach. This architecture ensures not only modularity but also scalability, flexibility, and an efficient division of responsibilities.

2.2 System Architecture

2.2.1 Client Requests

Requests are initiated by clients from various platforms. Upon reaching our system, these requests are promptly distributed by load balancers to ensure a balanced workload among the multiple request handlers.

2.2.2 Flask-based Request Handlers

All request handlers are built using Flask, a lightweight web framework that is well-suited for creating robust and scalable web servers. These handlers serve as the first line of processing for incoming client requests. They are responsible for interacting with the Apache Kafka messaging system: pushing incoming tasks to the Request Topic and awaiting their completion in the Response Topic.

2.2.3 Data Management with Apache Kafka

Apache Kafka, a high-throughput distributed messaging system, stands as the backbone of our data flow mechanism. It enables asynchronous communication between our services. Tasks are picked from the Request Topic, processed, and once completed, the results are dispatched to the Response Topic.

2.2.4 Task Scheduling with Celery

Upon receiving a model training task, it is the duty of our Celery-based schedulers to ensure its timely execution. Celery provides us with a reliable and efficient task scheduling mechanism, ensuring that tasks are evenly distributed and processed without delay.

2.2.5 Model Training in Keras/TensorFlow

The core of our service lies in model training, for which we leverage the power of Keras and TensorFlow. These frameworks provide the tools needed to process complex machine learning tasks efficiently, ensuring optimal results in a scalable manner.

2.2.6 Communication and Deployment

Our microservices, housed within Kubernetes pods, guarantee a resilient and auto-scalable environment, ensuring that each component of our backend system can independently scale based on the load. The inter-service communication is facilitated seamlessly through Apache Kafka, ensuring efficient data transfer without direct dependencies between the services.

2.2.7 Notifications with Email Service

Once a model is trained, our dedicated Email Service notifies the user about the completion of the task, providing them with pertinent details and results.

2.2.8 Persistent Storage

All vital data, inclusive of the trained models and their associated metadata, find their home in a centralized lightweight Mongo DB database, ensuring data integrity and easy retrieval.

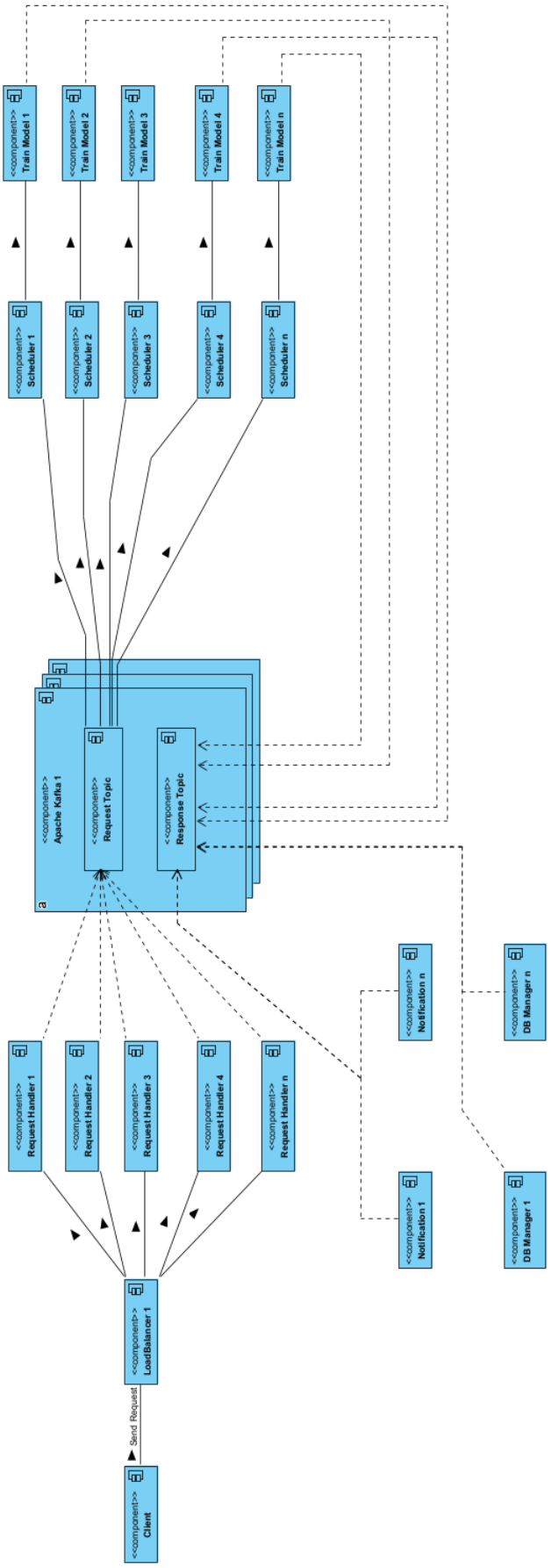

Figure 2.1: Back-End Solution Architecture

2.3 API Endpoints

2.3.1 User Management

- **Endpoint:** `/api/user/google-auth`
  - **Type:** POST
  - **Body:** JSON { "token": "Google authentication token" }
- **Endpoint:** `/api/user/logout`
  - **Type:** POST
- **Endpoint:** `/api/user/profile`
  - **Type:** GET
- **Endpoint:** `/api/user/update`
  - **Type:** PUT
  - **Body:** JSON { "username": "string" (required), "email": "string" (optional), "password": "string" (optional) }

2.3.2 Model Management

- **Endpoint:** `/api/model/train`
  - **Type:** POST
  - **Body:** JSON { "architecture_id": "Existing architecture id", "dataset_id": "id of a dataset", "Input": "Comma separated params", "Output": "Comma separated params", "userId": "user ID or username", "trainPercent": "Subset of dataset in percentage", "validationPercent": "Subset of dataset in percentage", "testPercent": "Subset of dataset in percentage"}
  - **Description:** Submit Training
- **Endpoint:** `/api/model/list`
  - **Type:** GET
  - **Parameters:** userId (to retrieve models associated with a specific user)

  - **Description:** Retrieve models associated with a specific user (It has to contains status of every model as well)

- **Endpoint:** `/api/model/delete/{model_id}`
  - **Type:** DELETE
  - **Description:** Delete a model based on its id
- **Endpoint:** `/api/model/status/{model_id}`
  - **Type:** GET
  - **Description:** Check model status
- **Endpoint:** `/api/model/result/{model_id}`
  - **Type:** GET
  - **Description:** Retrieve model

2.3.3 Neural Network Architecture Management

- **Endpoint:** `/api/nn-architecture/upload`
  - **Type:** POST
  - **Body:** JSON { "userId": "user ID or username", "architecture_name": "Name for architecture", "architecture": "NN architecture details as JSON format" }
  - **Description:** Upload a new neural network architecture.
- **Endpoint:** `/api/nn-architecture/list`
  - **Type:** GET
  - **Parameters:** userId (to retrieve architectures associated with a specific user)
  - **Description:** List all neural network architectures associated with a specific user.
- **Endpoint:** `/api/nn-architecture/delete/{architecture_id}`
  - **Type:** DELETE
  - **Description:** Delete a specific neural network architecture based on its ID.
- **Endpoint:** `/api/nn-architecture/edit/{architecture_id}`

  - **Type:** PUT
  - **Body:** JSON { "architecture": "Updated NN architecture details" }
  - **Description:** Update the details of an existing neural network architecture.

- **Endpoint:** `/api/nn-architecture/detail/{architecture_id}`
  - **Type:** GET
  - **Description:** Retrieve detailed information about a specific neural network architecture.

2.3.4 Model Validation

- **Endpoint:** `/api/validation/submit/`
  - **Type:** POST
  - **Body:** JSON { "architecture_id": "Existing architecture id", "datasetName": "string", "Input": "Comma separated params", "Output": "Comma separated params", "userId": "user ID or username", "trainPercent": "Subset of dataset in percentage", "validationPercent": "Subset of dataset in percentage", "testPercent": "Subset of dataset in percentage"} item
  - **Description:** Submit Validation

2.3.5 Database Management

- **Endpoint:** `/api/dataset/upload`
  - **Type:** POST
  - **Body:** File
  - **Description:** Upload file
- **Endpoint:** `/api/dataset/list/username`
  - **Type:** POST
  - **Description:** List all datasets available/associated with a user
- **Endpoint:** `/api/dataset/details/datasetId`

  - **Type:** GET
  - **Description:** Retrieve details of a particular dataset.
- **Endpoint:** `/api/dataset/delete/datasetId`
  - **Type:** GET
  - **Description:** Delete a dataset

2.3.6 Notification Management

- **Endpoint:** `/api/notification/list`
  - **Type:** GET
  - **Description:** Retrieve all notifications for a user.
- **Endpoint:** `/api/notification/read/notificationId`
  - **Type:** PUT
  - **Description:** Mark a notification as read.

2.3.7 System Management

- **Endpoint:** `/api/system/status`
  - **Type:** GET
  - **Description:** Check the system's health or status.
- **Endpoint:** `/api/system/metrics`
  - **Type:** GET
  - **Description:** Retrieve metrics or statistics related to system usage, active tasks, etc.

2.4 Data Flow

The backend plays a crucial role in facilitating the communication and data exchange between the frontend interface and the various services and components of our system. This section

delineates the data flow and interactions that take place from the moment a user initiates a request via the frontend until results are returned.

2.4.1 User Interaction and Request Handling

1. The user interacts with the frontend interface, where they can choose to upload datasets, specify neural network architectures, or initiate model training/validation/prediction tasks.
2. Upon user action, the frontend sends a corresponding HTTP request to the backend's Flask-based request handlers, facilitated by the LoadBalancer 1.
3. The request handlers validate the received data, ensuring that necessary parameters like dataset properties and NN architecture specifications are included and correct.

2.4.2 Model Training and Task Queuing

1. Once the request handler confirms the validity of the request, it places a message in the Apache Kafka Request Topic. This message contains all necessary information for the task at hand.
2. Dedicated schedulers, listening to the Request Topic, pick up the new tasks. They then schedule the required operations like model training.
3. The actual model training tasks are managed by Keras/Tensorflow-based microservices. These are allocated by LoadBalancer 3 and are responsible for consuming the tasks from the schedulers and performing the computations.

2.4.3 Storing Results and Notifying Users

1. Once a model training or validation task is complete, the result is stored in a dedicated database for persistence.
2. A message is placed in the Apache Kafka Response Topic, indicating the completion of the task and the location (e.g., a unique ID) of the result in the database.
3. The original request handler, which is listening to the Response Topic, picks up this message.
4. An email notification is sent to the user via the Email Service, informing them that their request is complete and providing them with a link or instructions to view/download their

results.

2.4.4 Feedback to Frontend

1. After processing, the backend sends an HTTP response back to the frontend. This can be an acknowledgment of task initiation, an error message (in case of invalid inputs), or results for immediate tasks.
2. For longer tasks, once the user receives the email notification, they can interact with the frontend interface again to view or retrieve their results, which would be fetched from the database by the backend.

2.4.5 Authentication and User Management

1. Users initiate authentication requests either via regular login credentials or through Google OAuth.
2. The backend verifies the provided credentials or OAuth tokens and sends back an authentication token (e.g., JWT) upon successful validation.
3. This token is used in subsequent interactions between the frontend and backend to ensure security and data integrity.

2.5 Database Design

2.5.1 Users Table

This table manages registered user data.

Field Name	**Type**	**Description**
user_id	INT (Primary Key, Auto Increment)	Unique identifier for each user
username	VARCHAR(255)	Username chosen by the user
password_hash	VARCHAR(255)	Hashed password for security
email	VARCHAR(255)	User's email address
auth_token	VARCHAR(255)	JWT or similar token for user authentication

2.5.2 Datasets Table

This table stores metadata about user-uploaded datasets.

Field Name	Type	Description
dataset_id	INT (Primary Key, Auto Increment)	Unique identifier for datasets
user_id	INT (Foreign Key)	Reference to the user who uploaded the dataset
name	VARCHAR(255)	Name or label of the dataset
path	VARCHAR(255)	File path or location in the storage
upload_date	DATE	Date the dataset was uploaded

2.5.3 Model_Architectures Table

This table captures the neural network architectures specified by users.

Field Name	Type	Description
architecture_name	VARCHAR(50)	Name (Alias) for NN architectures
architecture_id	INT (Primary Key, Auto Increment)	Unique identifier for NN architectures
user_id	INT (Foreign Key)	Reference to the user
configuration	TEXT	JSON storing the NN configuration
creation_date	DATE	Date the architecture was defined

2.5.4 Model_Training_Tasks Table

For tracking model training tasks.

Field Name	Type	Description
model_name	VARCHAR(50)	Model Name (Alias)
task_id	INT (Primary Key, Auto Increment)	Identifier for tasks
user_id	INT (Foreign Key)	Reference to the user initiating the task
dataset_id	INT (Foreign Key)	Reference to the dataset used
architecture_id	INT (Foreign Key)	Reference to the architecture
status	VARCHAR(50)	Status (Pending, Processing, Complete)
result_path	VARCHAR(255)	File path for the trained model or output
start_time	DATETIME	When the task started
end_time	DATETIME	When the task completed

2.6 Model Generation and Training

Model generation and training form the cornerstone of our backend infrastructure. Leveraging the power of Keras/TensorFlow, our backend is designed to efficiently train neural networks based on user-defined architectures and datasets.

2.6.1 Model Generation

Upon receiving a validated neural network architecture and associated dataset from a user, the backend initializes a new model instance using the Keras library. The architecture is translated into a series of layers, with specified neurons, activation functions, and other relevant parameters.

```
model = Sequential()
... # Define layers based on user input
model.add(Dense(units=...))
model.add(Activation(...))
```

2.6.2 Model Training

Once the model is generated, it is trained using the user-provided dataset.

1. The dataset is preprocessed, split into training, testing and validation sets.
2. The model undergoes training using the training set. Metrics such as loss and accuracy are monitored.
3. Validation is performed periodically to check the model's performance on unseen data.
4. Callbacks, like early stopping, can be implemented to enhance the training process.

2.6.3 Model Storage and Retrieval

After training, the model weights are saved to the database for future retrieval. The model's metadata, including its training accuracy, loss, and other relevant metrics, are also stored.

```
model.save('path_to_my_model.h5')
```

Users can later retrieve their trained models, load them into their applications, or further fine-tune them with additional data.

2.6.4 Scheduling and Background Jobs

Model training can be a time-consuming process. To manage this efficiently, we employ Celery to handle background tasks. When a user initiates a model training request, instead of training the model immediately, a task is added to the Celery queue. This ensures that the server remains responsive, and users can continue with other activities or monitor the training's progress.

2.6.5 Resource Management and Scalability

Using Kubernetes, we ensure that our training processes are scalable. When the demand increases, Kubernetes can spawn additional pods to handle the increased load, ensuring efficient resource utilization and fast response times.

2.6.6 Continuous Monitoring and Feedback

Post-training, metrics like loss, accuracy, and validation scores are captured and provided back to the user. This feedback loop ensures that users can iterate and improve their models based on tangible results.

2.7 Task Queue and Background Jobs

In the context of our backend infrastructure, tasks such as model training and data preprocessing, which are computationally intensive and time-consuming, are ideally suited to be managed as background jobs. Executing these tasks synchronously might result in long waiting times for users and can impact the performance and responsiveness of the main application. Thus, we utilize task queues and background job processing mechanisms to efficiently handle these operations.

2.7.1 Role of Celery

Celery plays a pivotal role in our task management strategy. It's a distributed task queue system that allows us to run tasks asynchronously in the background, ensuring our main application remains responsive.

```
from celery import Celery


app = Celery('tasks', broker='pyamqp://guest@localhost//')
```

```
@app.task
def train_model(model_config, dataset):
    ... # Model training logic here
```

2.7.2 Integration with Flask

Integrating Celery with Flask is straightforward. When a user submits a request to train a model, Flask initiates a new Celery task and adds it to the queue. The user receives an immediate response, and the task is executed in the background.

```
from my_celery_tasks import train_model

@app.route('/train', methods=['POST'])
def initiate_training():
    model_config = request.json['model_config']
    dataset = request.json['dataset']
    task = train_model.delay(model_config, dataset)
    return jsonify({"message": "Training initiated", "task_id": task.id}), 202
```

2.7.3 Scalability and Worker Management

Celery workers are responsible for executing the tasks. Using Kubernetes, we can dynamically scale the number of Celery workers based on the demand. When there's an influx of training requests, Kubernetes can spawn more worker pods to handle the increased load.

2.7.4 Monitoring and Error Handling

Celery provides tools to monitor task progress and handle failures. If a task fails due to an error, it can be retried, logged, or even notify administrators. This ensures that our backend is resilient to potential issues and can recover gracefully from failures.

```
@app.route('/task-status/<task_id>')
def task_status(task_id):
    task = train_model.AsyncResult(task_id)
    return jsonify({"status": task.status, "result": task.result})
```

2.8 Error Handling and Logging

The robustness and reliability of any application largely depend on its ability to gracefully handle errors and effectively log crucial events. This section elaborates on the strategies and tools used for error handling and logging within our backend infrastructure.

2.8.1 Error Handling in Flask

Flask provides a convenient way to handle errors using decorators. By defining custom error handlers, we can ensure that users receive a consistent and informative response regardless of the type of error.

```
@app.errorhandler(404)
def not_found(e):
    return jsonify({"error": "Resource not found"}), 404


@app.errorhandler(500)
def internal_error(e):
    return jsonify({"error": "An internal error occurred"}), 500
```

2.8.2 Error Reporting and Alerts

For critical errors, especially those that might affect the functionality of our system, we have implemented alert mechanisms. These can notify the development or operations team immediately if something goes wrong, allowing for swift remediation.

2.8.3 Logging with Flask

Flask's in-built logging mechanism is used to log various events in the system. Logging is crucial not just for debugging but also for auditing and understanding user behavior.

```
from logging import FileHandler, WARNING


file_handler = FileHandler('errorlog.txt')
file_handler.setLevel(WARNING)
app.logger.addHandler(file_handler)
```

2.8.4 Integration with External Logging Services

For more comprehensive log management and to facilitate easier searching, filtering, and alerting, logs are forwarded to an external logging service. This also ensures that logs are retained and backed up, even if the main application encounters issues.

2.8.5 Best Practices

- **Detailed Errors for Development, Generic for Production:** While detailed error messages are beneficial during development, they might expose sensitive information in a production environment. Thus, generic error messages are returned to users, but detailed logs are maintained internally.
- **Regular Audits:** Logs are periodically reviewed to ensure that no sensitive information, like passwords or secrets, is being logged.
- **Immediate Handling:** Instead of letting errors propagate through the system, they are handled as soon as they occur, ensuring system stability.
- **Retrying:** In case of temporary issues, like a brief network outage, operations are retried a few times before being classified as failed.

2.9 Security and Authentication

The security and authenticity of the users and data in our system are paramount. This section provides an overview of the methodologies and technologies adopted to ensure that our backend remains secure, and that only authorized users can access specific resources.

2.9.1 Secure Transmission

To ensure the confidentiality of data in transit, our backend uses HTTPS (secured by TLS) for all data transmissions. This ensures that all communication between the client and server is encrypted and safe from eavesdropping.

2.9.2 Authentication

Authentication verifies the identity of users trying to access the system.

- **Google OAuth:** Users can authenticate using their Google credentials, leveraging the OAuth 2.0 protocol. This method doesn't require our system to handle or store user passwords directly.
- **Regular Authentication:** For users who prefer not to use Google authentication, we offer a traditional username and password mechanism. Passwords are hashed using a secure hashing algorithm before storage.

2.9.3 Authorization

Once authenticated, user permissions determine what resources or actions they can access. Role-based access control (RBAC) is implemented to assign users specific roles, each with its permissions.

2.9.4 Data Encryption

Sensitive data stored in the database, such as user information, is encrypted at rest. This ensures that even in the unlikely event of a data breach, the raw data remains protected and unintelligible.

2.9.5 Input Validation and Sanitization

To protect against SQL injection, XSS, and other injection attacks, all user inputs are rigorously validated and sanitized before being processed.

2.10 Testing and Validation

Ensuring the accuracy, reliability, and stability of our backend system is crucial. As such, we employ rigorous testing methodologies to ensure that each component behaves as expected and integrates seamlessly with the rest of the system. In this section, we will discuss the testing approach and illustrate it with an example from our test suite.

2.10.1 Unit Testing

Unit tests target specific, isolated parts of our software to ensure they perform as expected.

- **Test Framework:** We utilize the `unittest` framework, which is built into Python, to structure and run our unit tests.

- **Mock Data:** Mock data is used to simulate real-world scenarios. This allows us to test our functions and methods under controlled conditions.
- **Assertions:** Assertions are used to compare the output of our function or method to an expected result.

2.10.2 Example: Model Builder Test

To highlight our testing approach, let's consider our 'test_model_builder.py' file:

```
import unittest
from keras.models import Model
from nn_generation import model_builder


class TestModelBuilder(unittest.TestCase):
    ...
    def test_build_model(self):
        model = model_builder.build_model(self.mock_config)
        self.assertIsInstance(model, Model)
        self.assertEqual(len(model.layers), len(self.mock_config['layers']))
    ...
```

In this test suite:

- We're testing the functionality provided by the `model_builder` module.
- The `setUp` method initializes mock data, which is used to simulate a typical model configuration.
- The `test_build_model` method validates that the generated model is an instance of Keras' `Model` class and that the number of layers in the model matches the configuration.

2.10.3 Integration and System Testing

Beyond unit tests, integration tests verify that different components of our system work together as expected, while system tests ensure the system works as a whole.

2.11 CI/CD

Continuous Integration (CI) and Continuous Deployment (CD) play a pivotal role in modern software development, ensuring that software is both reliable and rapidly deployable. By employing GitLab as our CI/CD platform, we ensure that our backend system is always in a deployable state, with automated testing and streamlined release processes.

2.11.1 Continuous Integration with GitLab

- **Automated Build and Testing:** Upon every code push to our Git repository, GitLab automatically triggers a build and runs our test suites, ensuring immediate feedback if any issues arise.
- **Merge Requests:** Before any code is merged into the main branch, it goes through a merge request. GitLab CI provides automated feedback on these merge requests, ensuring that new features or changes do not introduce bugs.
- **Docker Integration:** GitLab CI seamlessly integrates with Docker, allowing us to build, test, and push Docker images directly within our CI pipeline.

2.11.2 Continuous Deployment with GitLab

- **Automated Deployments:** On passing all tests and after code review, GitLab can automatically deploy our backend services to the appropriate environment, ensuring rapid release cycles.
- **Environments:** GitLab allows us to define multiple environments such as staging, production, and development. This ensures that our code is tested in a replica of the production environment before actual deployment.
- **Rollbacks:** If any issues are detected post-deployment, GitLab provides easy rollback options to revert to the previous stable version.

2.11.3 Monitoring and Feedback

Post-deployment, GitLab offers monitoring tools that provide insights into the performance and usage of our applications. This continuous feedback is invaluable for spotting potential issues and areas of improvement.

2.12 Future Enhancements and Conclusion

2.12.1 Future Enhancements

- **Additional Kafka Layers:** As our system scales and handles a more diverse set of tasks, there's a potential to integrate more Apache Kafka topics and layers. This would allow for better segmentation of tasks, more granular monitoring, and efficient parallel processing of data.
- **Flexibility and Customization:** While our current architecture serves our present needs, we aim to add more flexibility to the system. This includes allowing users or administrators to dynamically adjust system parameters, adapt to different data processing needs, and possibly integrate with other external systems or services.
- **Scalability Improvements:** As user demands grow, we will consider refining our load balancing strategies, possibly introducing more advanced algorithms or solutions that can distribute tasks even more efficiently across our services.
- **Advanced Monitoring and Analytics:** To continuously improve our services and maintain high availability, we plan on integrating advanced monitoring tools and analytics to gain insights into system performance, user behavior, and potential bottlenecks.

Chapter 3

Frontend Project Design

3.1 Website Service

3.2 Angular Project Architecture

Angular was chosen as the front-end framework for this project due to its robust architecture, strong community support, and extensive set of features. Angular provides a structured and modular approach to building web applications, making it an ideal choice for developing complex and feature-rich applications. Following is the project structure that consists of Components, API Services, Models, and Utilities.

The UML component diagram in (Fig. 3.1) describes the frontend architecture for this project. The architecture follows the Angular MVC pattern that consists of three distinct parts (Components, API Services, Models) and Utilities to hold common functions.

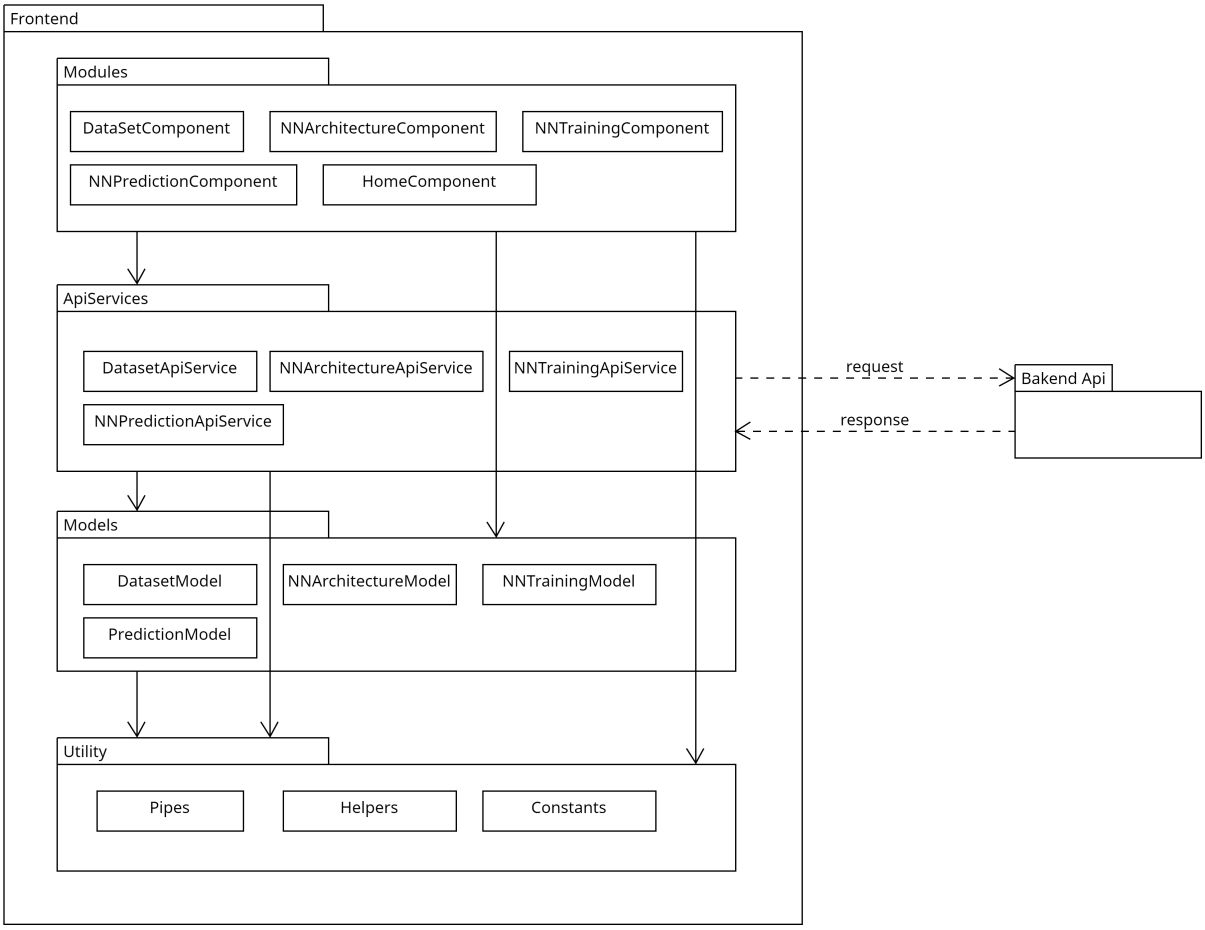

Figure 3.1: MUNNNToolbox front-end architectural UML component diagram

3.3 Angular Components

Angular components are essential parts of the application's user interface, allowing users to interact with and manage various aspects of neural network design, training, and prediction.

1. **Home Component:**
   - *Purpose*: This serves as the entry point of the application, functioning as the landing page. It may also integrate JWT authentication in the future. Additionally, it provides links for users to create new neural networks and access existing ones.
2. **Dataset Component:**
   - *Purpose*: This component allows users to :
     - View a list of previously uploaded datasets.
     - Upload new datasets in the form of CSV files, which are used for training models.

– Submit uploaded files to the backend for validation and storage.

3. **Architecture Component:**

   - *Purpose*: This component offers a visual interface for users to design neural network architectures. It allows users to specify layers, neurons, activation functions, and connections. Users can also view and modify existing diagrams representing network architectures.

4. **Training Component:**

   - *Purpose*: Users can initiate the training process by selecting the architecture they want to train and the dataset, along with the input and output columns to be used during training.

     Fig. 3.2 shows the graphical interface of the component, it shows a list of the user's existing training. Fig. 3.3 describes the form that the user should fill in to submit a new modal training.

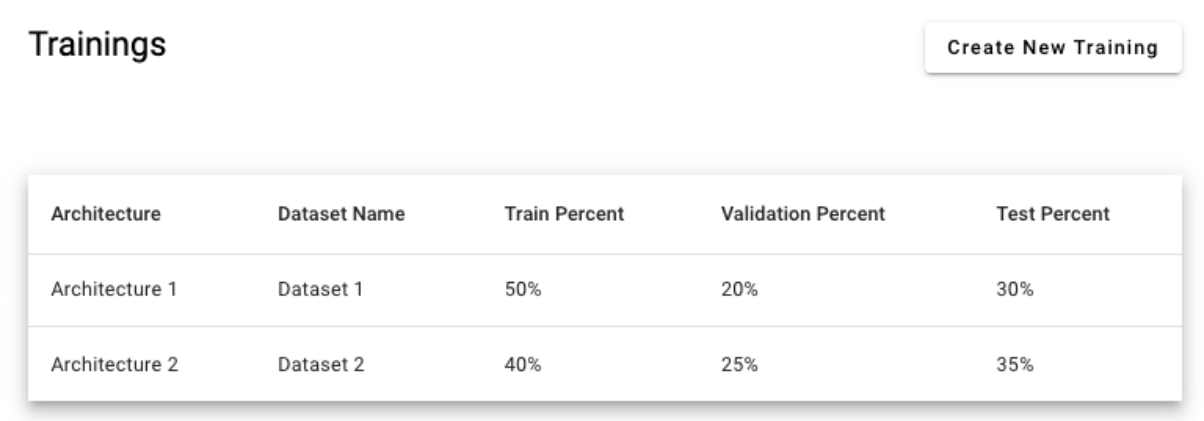

Figure 3.2: MUNNNToolbox front-end Model Training UI List

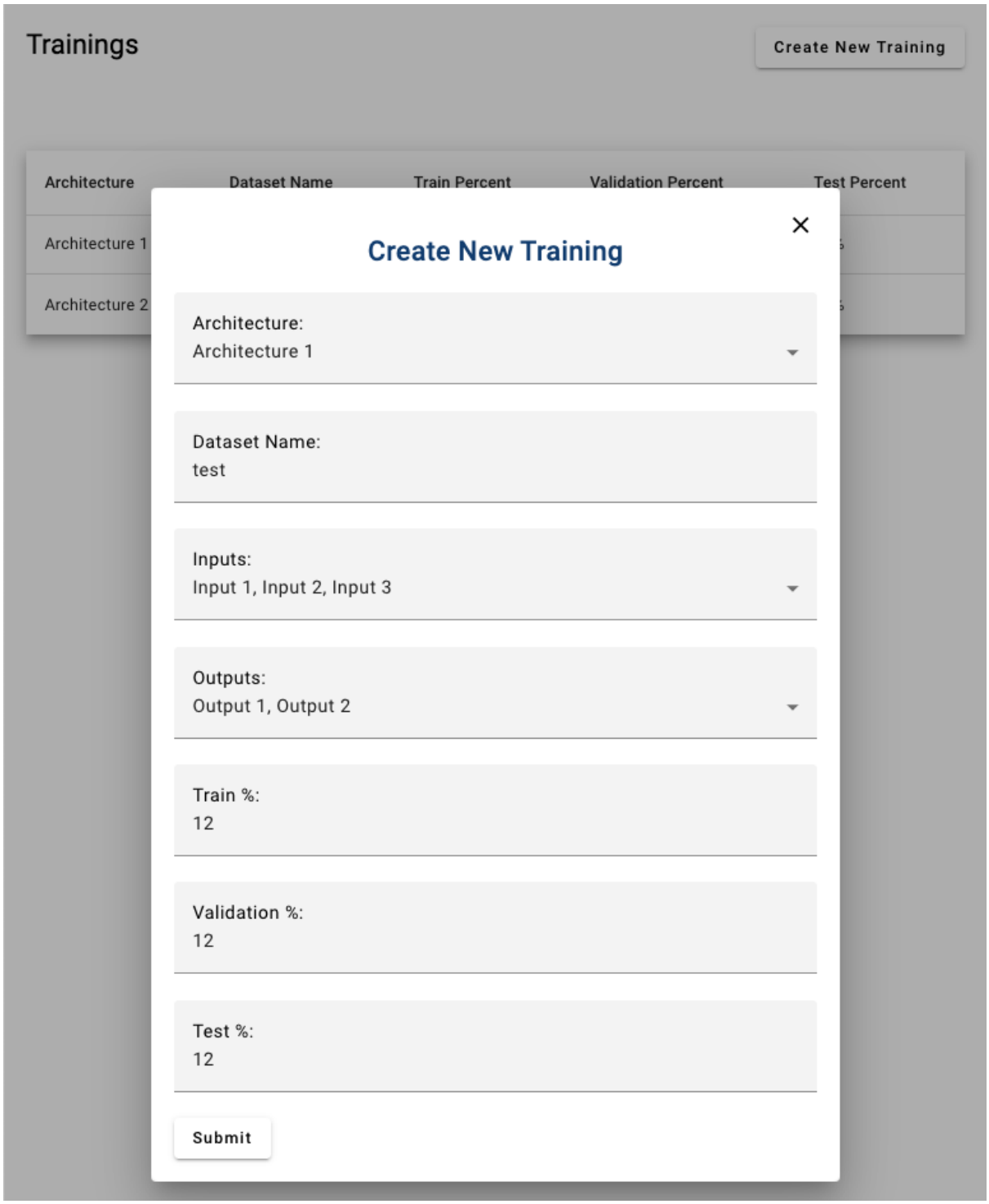

Figure 3.3: MUNNNToolbox front-end Model Training UI Modal

5. **Prediction Component:**

   - *Purpose*: This component enables users to utilize trained neural networks to make predictions based on input data. Users can select the model and input variables to obtain predicted outputs.

3.4 API Services

API services are responsible for facilitating communication between the frontend and backend of the application, enabling data exchange and coordination of various processes.

1. **DataSet Service:**
    - *Responsibilities*: This service handles API communication related to datasets. It manages tasks such as file uploads and retrieval of a list of existing datasets.
2. **Architecture Service:**
    - *Responsibilities*: This service deals with architectural API operations. It is responsible for creating new network architectures, updating existing ones, and retrieving lists of available architectures.
3. **Training Service:**
    - *Responsibilities*: The training service facilitates the initiation of training processes through API calls. It also provides information about training status and a list of all training instances.

3.5 Models

Models represent data structures used to organize and manage information within the application, ensuring consistency and clarity in data handling.

1. **DataSet Model:**
    - *Purpose*: This model represents the structure of datasets uploaded by users.
2. **Architecture Model:**
    - *Purpose*: This model stores user-defined architectural configurations for neural networks.
3. **Training Model:**
    - *Purpose*: This model holds data required to initiate training, including status and ID details.
4. **Prediction Model:**

- *Purpose*: This model stores details necessary for making predictions based on user input data and the return output.

3.6 Utilities

Utilities encompass a collection of tools and resources that enhance the functionality and maintainability of the application.

- **Pipes:** Angular pipes for transforming and formatting data within the application, enhancing data presentation.
- **Helpers:** Utility functions for common tasks and operations, streamlining development and maintenance.
- **Constants:** Constants and configuration values used throughout the application, promoting consistency and ease of updates.

3.7 Tests

In the frontend, the focus should be on writing independent integration and unit tests. Jasmine and Karma have been adopted as the testing tools. Jasmine is a testing framework, and Karma is a test runner, allowing tests to be run in real browsers and capturing test results. Both are well-suited for testing Angular applications.

The plan is to write tests for all components in Angular before implementation. Below are the main Angular application components with a brief description of the tests to be written for each. These test cases are written according to the current UI design and are subject to change.

1. **Home Component:**
   - *Brief Description*: The homepage component serves as the entry point of the application and contains various user interface elements. It integrates user authentication and provides links to create new neural networks and access existing ones.
   - *Tests to Be Written*:
     - Test for verifying the presence and functionality of authentication integration.
     - Tests for the correctness of links to create and access neural networks.

2. **Dataset Component:**

   - *Brief Description*: This component allows users to manage datasets, including viewing existing datasets and uploading new ones in CSV format for model training.
   - *Tests to Be Written*:
     - Tests for uploading CSV files and verifying successful file validation.
     - Tests for listing and displaying existing datasets.

3. **Architecture Component:**

   - *Brief Description*: The architecture component provides a visual interface for designing neural network architectures. Users can specify layers, neurons, activation functions, and connections.
   - *Tests to Be Written*:
     - Tests for rendering the visual interface accurately.
     - Tests for configuring network architecture elements and verifying their functionality.

4. **Training Component:**

   - *Brief Description*: This component allows users to initiate training for neural network architectures. Users select the architecture, dataset, and input/output columns for training.
   - *Tests to Be Written*:
     - Tests for initiating training processes and verifying the correct parameters are passed.
     - Tests for capturing and displaying training status.

5. **Prediction Component:**

   - *Brief Description*: The prediction component enables users to make predictions using trained neural networks. Users select the model and input variables for prediction.
   - *Tests to Be Written*:

- Tests for selecting the model and input variables.
- Tests for making predictions and validating the accuracy of the output.

3.7.1 Datasets Component Test Cases

Listing 3.1 is an example of Jasmine's Test cases for the Dataset component. The Jasmine test for the "Dataset Component" serves as a validation mechanism to ensure the correct behavior of this component. It begins by configuring the testing environment and injecting dependencies such as the DatasetService. The test cases cover various aspects of the component's functionality. The first two test cases focus on uploading and handling files, verifying that the component can successfully manage file uploads and handle invalid file formats. The third test case is dedicated to retrieving a list of existing datasets, simulating the process of fetching data from the DatasetService and confirming that the component correctly displays the retrieved datasets. These tests, alongside others that can be added, contribute to the robustness and reliability of the "Dataset Component" within the Angular application.

Listing 3.1: Sample Jasmine Test for the dataset Component

```
// Sample Jasmine test for the Dataset Component

describe('DatasetComponent', () => {
  let component: DatasetComponent;
  let fixture: ComponentFixture<DatasetComponent>;
  let datasetService: DatasetService; // Import and inject the dataset service
  let mockDatasets: any[]; // Mock data for existing datasets

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [DatasetComponent],
      imports: [FormsModule],
      providers: [DatasetService], // Provide the DatasetService
    });

    fixture = TestBed.createComponent(DatasetComponent);
    component = fixture.componentInstance;
    datasetService = TestBed.inject(DatasetService); // Inject the DatasetService

    // Mock data for existing datasets
    mockDatasets = [{ id: 1, name: 'Dataset 1', description: 'Description 1' }];

    // Mock the DatasetService to return the mock data
```

```
    spyOn(datasetService, 'getDatasets').and.returnValue(mockDatasets);
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should upload a valid CSV file', () => {
    // Simulate user selecting and uploading a valid CSV file
    // Expectations here to verify that the file is successfully uploaded and processed
  });

  it('should handle an invalid file format', () => {
    // Simulate user uploading an invalid file format (e.g., a non-CSV file)
    // Expectations here to verify that the component handles the error correctly
  });

  it('should retrieve a list of existing datasets', () => {
    // Trigger the component to fetch existing datasets
    component.fetchExistingDatasets();
    // Expectations to verify that the component correctly displays the retrieved datasets
    expect(component.existingDatasets).toEqual(mockDatasets);
  });
});
```

3.8 CI/CD

The CI/CD pipeline to be implemented using GitLab. The pipeline will consist of three phases that involve building, testing, and deploying the Angular application. The pipeline will be defined in the project's root folder in a YAML file called '.gitlab-ci.yml'. Here's a brief summary of each phase:

Build Phase: The build phase involves compiling the Angular application, generating artifacts, and preparing it for deployment.

Test Phase: Configure the CI pipeline to execute the written tests as part of the CI process. If tests fail, the CI pipeline should halt and report the failures.

Deployment Phase: Define deployment configurations. Deploy the built Angular application to the selected target if the build and tests are successful.