BRAIN.AI
TEAM 5B
Section 1B

Hao Zhuang,305119276, Timzhuang
Jiayu Hu, 005118400, hujiayu9696
Sichen Song, 405183530, shsssc
Taixing Shi, 505191214, stx1998x
Yash Choudhary, 704630134, yashch123

Github: https://github.com/CS130-W20/team-B5

Design

Our work on this part of the project can be divided into two phases, design and implementation. As for design, we have designed the schema (database structure) and API interface for our application. The database we choose is Mysql where we have tables for user, session, model, data, and tasks. Foreign keys are used inside each table to make the association in our class diagram. For API interface design, we used the Open API specification format, which is a standardized yaml format that describes the API specification. This specification could help in generating code, documentation, and test cases in the implementation phase.

In the implementation phase, we made progress in all of the three sub-components of our application: the web UI, the backend API server, and the workers. For the backend server, we have finished most of the functionalities. The services for task management and user/session management have been finished. As for model and data, their creation, viewing, modification are finished. We are still working on their deletion, in which we must make use of Google Cloud Storage API that we are getting familiar with. For the worker development, we have finished the framework of the worker as well as data pre-processing and applying model to data. We also have a working prototype of the model training part. The only remaining part is to encapsulate model training and integrate it with the worker. For the front end, we have finished designing all the views necessary for displaying models, data, and tasks. We have also completed the logic for these views to render the information provided by the backend server. Moreover, we have finished developing the system for handling users' signing in and signing out. The only thing remaining for the front end is to integrate the front end with the backend server using the provided API.

Individual Contribution

- Frontend: Jiayu Hu https://github.com/CS130-W20/team-B5/pull/17
- Worker: Sichen Song https://github.com/CS130-W20/team-B5/pull/14
- Backend Server:
    - Hao Zhuang https://github.com/CS130-W20/team-B5/pull/14
    - Sichen Song https://github.com/CS130-W20/team-B5/pull/18
- Backend testing
    - Taixing Shi https://github.com/CS130-W20/team-B5/pull/20
    - Yash Choudhary https://github.com/CS130-W20/team-B5/pull/21

Architecture Patterns

We have utilized the microservice architecture pattern in our project. The interface between backend and front end is Rest API. Our backend is written as node.js express middleware functions that could be deployed to any cloud function services such as Amazon Lambda or Google cloud function. We choose this pattern since it could help us to decouple the development of frontend and backend to achieve better parallelism in development. Also, with the help of standardized API specification, we could mock API requests and responses easily for testing. Lastly, the pattern allows cloud function deploy services to take care of scaling.

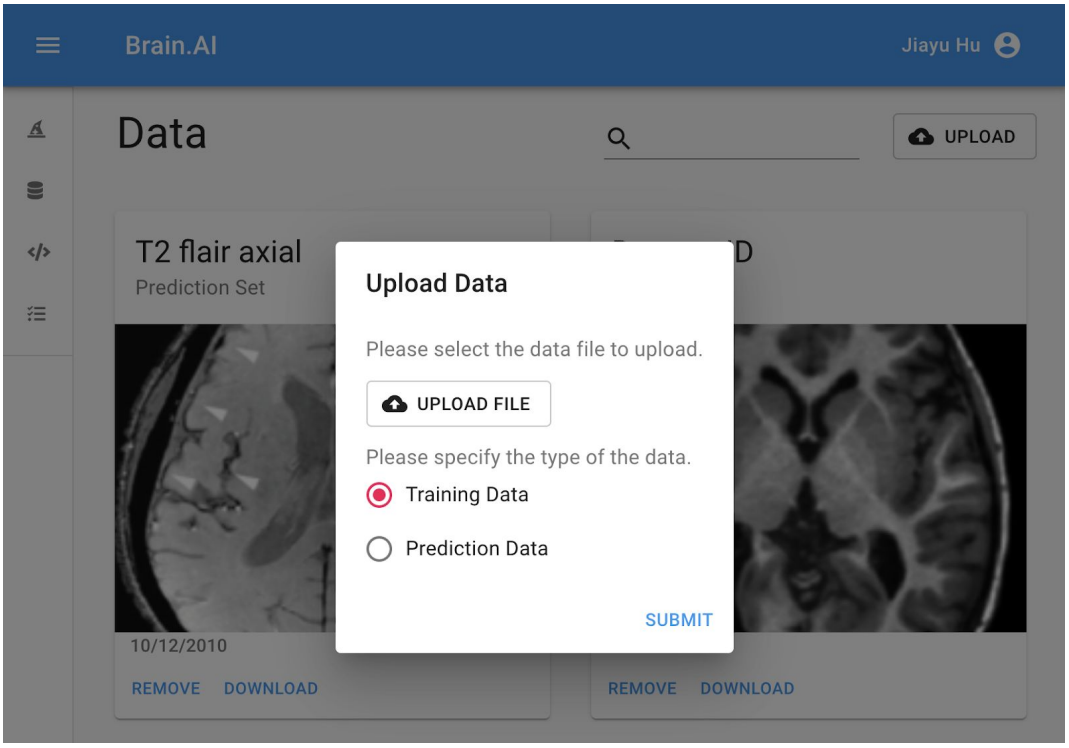
Also, we have used the event-driven architecture pattern for training and running models. In our design, backend servers act as event producers while the workers act as event consumers. The database acts as an event queue. With this pattern, the task creator (backend server) is decoupled from task runners (workers). As mentioned previously, the backend server could scale easily as microservices. With the help of this pattern, the workers could be separately scaled from the backend servers. This feature is critical since workers and backend servers perform different tasks and have different performance requirements. Lastly, this pattern also allows us to overcome the problem of lack of transaction support in microservice pattern. The database component has very good transaction support and fits the requirement of acting as an event queue. In this way we could avoid the problem of race conditions when multiple producers/consumers are in use.

Existing frameworks

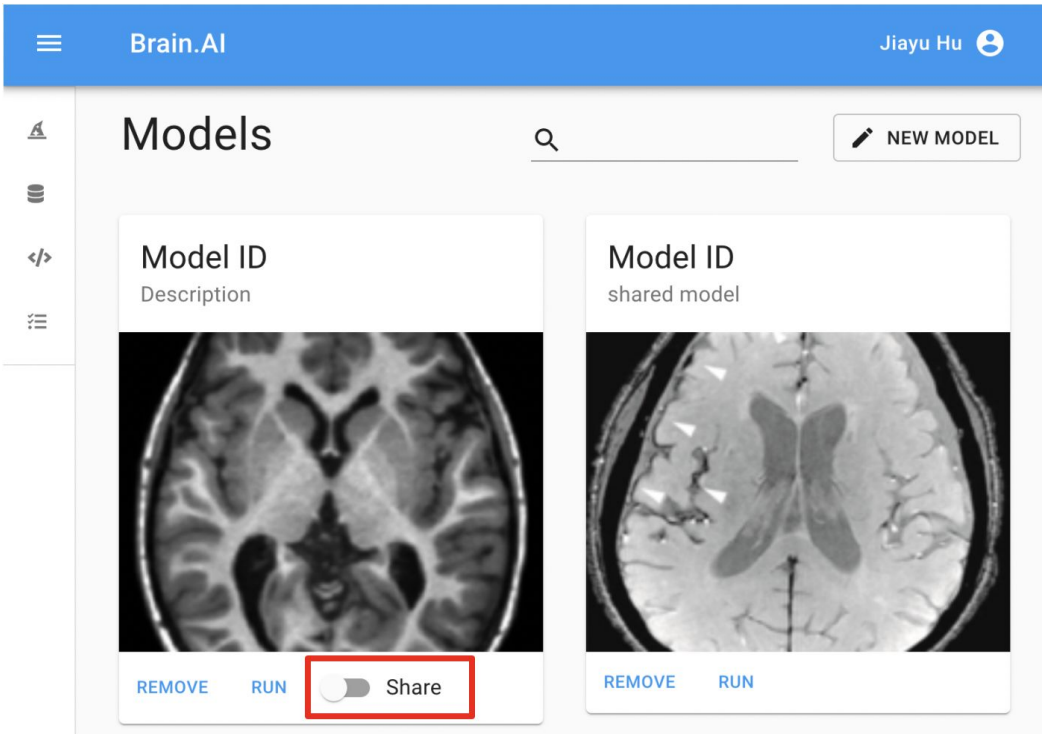
We used existing frameworks to boost our development. As for frontend, we used Material-UI component library together with React. These two libraries allow us to reuse existing components and focus more on the frontend business logic instead of UI layout and component design. The functional style of them also allows us to avoid bugs when developing complex dataflow. As for backend, we used oas-tools which is a node-js express middleware that parse requests according to our API specification. Taking advantage of it could allow our backend coding to focus more on business logic. We make use of Google cloud compute to run our workers, which provides GPU for our model training/running. As for testing, we used Tavern, which is a command line tool for automated testing of APIs using YAML-based syntax. It is simple and highly customizable, and we used it for all restful API testing. Lastly, we also used Google Cloud storage to store data and modules. Using the storage system of Google allows us to save the bandwidth of our servers. The high speed of the CDN system of Google also gives our users better upload/download experience. The Google Cloud Storage can also be mounted as a folder in Linux operating systems. This allows the workers to operate on cloud storage with pure file system interface, which dramatically reduced our amount of work since the workers are written with MATLAB production server and have very limited support for API requests.

User Interaction Scenario

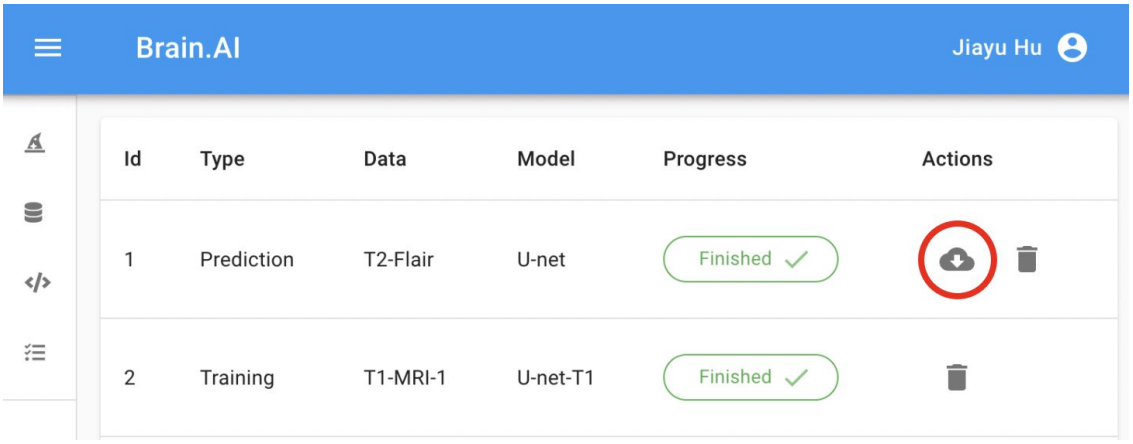
Users can use our front-end web page to upload data, create models, and download the prediction result for a prediction data set. As an example, the following dialog in data view is for our users to upload their data.


Brain.AI
Jiayu Hu
Data
UPLOAD
T2 flair axial
Prediction Set
Upload Data
Please select the data file to upload.
UPLOAD FILE
Please specify the type of the data.
Training Data
Prediction Data
SUBMIT
10/12/2010
REMOVE
DOWNLOAD
REMOVE
DOWNLOAD

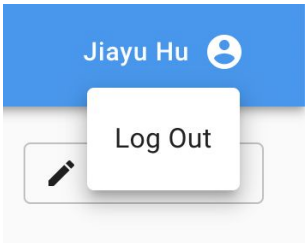
In the model view, users are able to share their models to other users using the switch marked in the red box.



Finally, our users can download the prediction result in the task view shown as follows. After the download icon marked as red circle is clicked, the prediction result will be downloaded.



Users can click the upper right corner to log out or sign in shown as follows.



Our team didn't make any changes to our architecture design from part B.

Description of API

We make use of the Open API specification to specify the API in our backend server. Documents are automatically generated from it.

The API specification is available at https://github.com/CS130-W20/team-B5/blob/master/docs/backend%20API/brainai-openapi.yaml and the online documentation is avaliable at https://cs130-w20.github.io/team-B5/.

Here's part of our API description.

createUser : create new user

path: /user, method: post

Query parameters

Name	Description
email*	String *Email of user* Required
password*	String *password of user* Required

Responses

Status: 200 - successfully created user

Status: 400 - bad input parameter

Status: 409 - conflict username

login : user login

path: /session, method: post

Query parameters

Name	Description
email*	String *email of user* Required
password*	String *password of user* Required

Responses

Status: 200 - successfully login

- Schema

{

user_token:

string

the token that authorize the user

example: 22d40d3978dd7945bbfcc0bc32a078e2129a253a97cbb04cbb71bc8c506aa9c5

}

Status: 400 - bad input parameter

Status: 401 - bad credential

Testing

Due to the complexity & separation of logical components in our application, we decided to test the worker, backend server and API package separately. Additionally, after further development, this approach will also make it easier for us to implement integration testing.

Testing the worker

For the worker the main focus was on its functionalities of verifying uploaded data, model prediction, and task status reporting. These functionalities were thoroughly tested.
The use cases being tested here range from when a user wants to find out the progress on a particular task to a user applying a prediction model to a dataset.
The test cases are available at
https://github.com/CS130-W20/team-B5/tree/master/worker/tests.

The MATLAB unit testing framework is used. For data verification, sample data files are provided for the worker to analyze their directory structure as well as file format. It's tested so that the worker could successfully identify whether the dataset is for training or for applying the model and can successfully generate a preview image for that dataset. For testing the report task status function, valid and invalid tasks as well as the status of task are passed to the function. The result of the function call should be reflected in the database. Thus, the updated database entries are checked by the test cases to see the task status is updated as expected. Lastly, the model prediction function is tested with a sample model and a sample dataset. The test case checks that one result image is generated for each of the input images.

Testing the API

For the backend, we have finished the basic black box restful API testing. The test cases are available at https://github.com/CS130-W20/team-B5/tree/master/backend/testing. User APIs are thoroughly tested by simulating a process where users first create User and login with their email and password, then log out with their session token. All http response codes and JSON responses of User APIs are properly tested. This testing covers the use case of a user signing up & creating an account on Brain.ai and when a user logs in & out of the application using proper credentials.

Testing the backend controllers

For our application the controllers in use are divided into User, Data, Model & Task APIs. These handle a lot of the use cases that are under development.
Testing Data APIs ensure that the code is reliable enough to verify the use cases of a user uploading, deleting & modifying a dataset.
Testing Model APIs ensure that the code is reliable enough to verify the use cases of a user uploading, deleting & modifying an existing or new model.

The use cases of Task APIs are mostly verified by strongly testing the workers (above).

To thoroughly test the controllers, unit functional testing was performed for each module separately using the Jest framework. The complete test cases are available at https://github.com/CS130-W20/team-B5/tree/master/backend/tests.

Sample testing scenarios for application:

1. User creating an account on application
    a. Test case for successfully creating a new user with given credentials
        i. *Expected response: status_code 200*
    b. Test case for bad (invalid) email/password inputs not working
        i. *Expected response: status_code 400*
    c. Test case for not creating new account if user already exists
        i. *Expected response: status_code 409*
    d. Test case for empty password field raising exception
        i. *Expected response: status_code 400*

- Test failure indicators: *status_code is returned wrong, user is added to user table in db, multiple same email users are created, no exception is raised for invalid inputs*

2. User logging into application
    a. Test case for successfully logging in user with right credentials
        i. *Expected response: status_code 200*
    b. Test case for not logging in user if wrong credentials are given
        i. *Expected response: status_code 409*
    c. Test case for not logging in user if bad (invalid) input credentials are given
        i. *Expected response: status_code 400*

- Test failure indicators: *status_code is returned wrong, wrong credentials login is possible, no exception is raised for invalid inputs*

3. User uploading dataset to application
    a. Test case for user successfully uploading a dataset
        i. *Expected response: status_code 200*
    b. Test case for user failing to upload dataset because of invalid session token
        i. *Expected response: status_code 401*
    c. Test case for raising exception if bad (invalid) name is given to uploading dataset
        i. *Expected response: status_code 409*
    d. Test case to ensure application responds with correct URL for uploaded dataset
        i. *Expected response: response.JSON has expected URL*

- Test failure indicators: *status_code is returned wrong, wrong URL is returned for dataset, invalid session tokens are passed ,no exception is raised for invalid inputs*

4. User modifying an existing dataset
    a. Test case for user to successfully rename an uploaded dataset
        i. *Expected response: status_code 200*

    b. Test case for failing to rename a dataset because of incorrect data id
        i. *Expected response: status_code 400*
    c. Test case for failing to rename a dataset because of incorrect dataset name
        i. *Expected response: status_code 400*
    d. Test case of raising exception if bad (invalid) input parameters are provided
        i. *Expected response: status_code 401*
- Test failure indicators: *status_code is returned wrong, user can rename dataset even after passing incorrect data id or name, no exception is raised for invalid inputs*

Scrum Challenges & Conclusion

For the development of features our team used the Scrum methodology to divide features into stories & logically work through them to complete each Epic. Since we have multiple components in our architecture like workers, backend and frontend defining solid user story for each use case was very helpful in organizing our workflow. Also, due to Product Backlog refinement during multiple scrum meetings we were able to coordinate and focus on the most important features. Some challenges we faced were because our components have high dependability, our stories get too big with multiple tasks to achieve (too many stories being opened). To mitigate this, we will try to aim at small batch sizes for stories for upcoming sprints and maybe try a daily standup approach to keep things in check.