01/24/2024:

* Went through a rough architecture/system design for the software
* Discussed what a package should carry, what will be on the front end, where to put the ML model, and how to get logging information from the backend and display it on the front end in real time

01/29/2024:

* Learned and used WebSocket to get logging information from the ML script and display it on the front end in real time
* Refactored ML code so that every function is in a designated class
* Refactored manager.py and mnist\_model.py to enable manager to update progress, allowing mnist\_model to focus on running ML tasks
* Setup a websocket in server’s app.py, pass it to manager, and let manager.py emit updates to the client app.js
* Currently debugging why socket.io is emitting the same information multiple times (id 0 emits 1 time, id 1 emits 2 times, id 2 emits 3 times, …) 🡪 Used socketio.once() to solve the problem
* Learned how to close/disconnect a WebSocket to avoid memory leaks/security issues.
* Used ChatGPT to refactor code and make it cleaner.
* Q: Don’t know how to check if the websocket is properly closed.

02/07/2024

* Carrying out issue #1: minimal front to back
* Just a simple front-to-back integration that lets users pick the params and train the model immediately. No Producer yet.
* Currently debugging this error: Access to fetch at 'http://localhost:9000/start\_experiment' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled. 🡪 Here’s what I found: The error stems from a security mechanism that browsers implement called the same-origin policy. The same-origin policy fights one of the most common cyber attacks out there: cross-site request forgery. In this maneuver, a malicious website attempts to take advantage of the browser’s cookie storage system. 🡪 Identified that it is mostly a **backend** problem, simply created a key ‘Access-Control-Allow-Origin’ – value ‘http://localhost:3000’ for the backend response object (which is a dict/json)
* Also modified the code a bit to remove proxy in package.json, set the server to use port 9000 instead of 5000.

02/08/2024

* Starts learning about **“Design job scheduling system”**. Some sources that I read:
  + [Job Scheduling Design: Behind the Scenes of a Distributed Job Scheduler (redwood.com)](https://www.redwood.com/article/job-scheduling-design/)
    - Desired job scheduling workflow -> API
    - Job metadata management -> Job ID, timestamp, execution time, dependencies
    - How to implement a task scheduler
    - Defining job execution
  + [Ace the System Design Interview: Job Scheduling System | by Zixuan Zhang | Towards Data Science](https://towardsdatascience.com/ace-the-system-design-interview-job-scheduling-system-b25693817950)
    - Create a new job with its schedule
  + [How we designed Dropbox ATF: an async task framework - Dropbox](https://dropbox.tech/infrastructure/asynchronous-task-scheduling-at-dropbox)
* Realized it’s such a big and complex topic, not something doable in 1 day 😊
* Discussed with pda, and agreed that we should start by designing a **linear system** first
* High level design:
  + Database design:
    - Read operations:
      * Given a userID, retrieve all jobs that belong to it (by client)
      * Find all jobs that are scheduled to run right now (by internal servers)
    - Write operations:
      * A user can create a new job schedule (by client)
      * The workers will add execution histories to the database (by internal servers)
      * The system updates the next execution timestamp of a job after running it (by internal servers)
    - Schema
      * Job table: UserID, JobID, timeCreated (UNIX timestamp)
      * History table: JobID, startTime (UNIX timestamp), runtime (seconds), epochs, lr, size, accuracy
* Finished a rough system design

02/14/2024

* Started to explore how to implement MongoDB Atlas into the application
* Upgraded Python interpreter version from 3.10.9 to 3.12.2
* Encountered a few problems:
  + When I switch the Python interpreter, VSCode looks at the 3.12 library, which is empty.
  + When I try to use pip and install the packages, it keeps installing to Python 3.10 and says that the package is already there.
  + When I uninstall Python 3.10.9, the computer seems to think that there is no longer Python while version 3.12 is literally installed. I couldn’t also use pip to install packages into 3.12
* Edit the system environment variables, in “Advanced” click on “Environment Variables”, then edit the PATH variable so that it points to Python version 3.12 and not 3.10 (which no longer exists) and points to the version 3.12 scripts 🡪 I learned that I have to manually hand-hold my computer and tell it where to look for Python and its packages (pip is pre-installed with Python if downloaded through Python’s website)
* Successfully connected Python with MongoDB database
* Since I already have a pretty clear idea what the database would hold, as well as what the entities look like, it makes sense to use an **Object Document Mapper (ODM)** library to enforce a structure on what data gets to go inside the database, since MongoDB is schema-less. 🡪 Thinking of using **MongoEngine**

2/15/2024

* Met with pda to fix a few bugs
  + A lot of unnecessary files -> Safely removed them
  + Backend not extracting information from Frontend -> Backend was accessing information in request.form, while Frontend was sending package as JSON -> Easy fix by accessing request.json
  + Frontend failed to fetch route start\_startexperiment due to CORS policy -> It was a hard fix because I thought I have already solved it before. I learned that **the CORS policies are separate for Backend routes and websocket**. I have only configured CORS policy for the websocket, and not the Backend routes.
  + Type error in JSON -> Easy fix since the JSON data are all strings while I want integers/floats. I only have to convert them in the Backend.

2/18/2024

* Defined the Job schema
* Connected to MongoDB using **MongoEngine**
* Start writing the route “create-job”
* Learned about **class methods**
* Progress so far: Can add jobs to DB, some basic validation mechanisms, Frontend and Backend communicating properly (currently only using console for printing/logging).

2/19/2024

* Met with pda to merge RabbitMQ code with DB code
* Ran some tests to debug and check for errors
* Discussed the plan moving forward
* Minor fixes to code to make it work

2/21/2024

* Refactoring code before implementing further functionalities
* Write code to update corresponding Document (job config) when the training is done.
* Learned that if files are created/run in the same process, you don’t need to make a connection to the database again (app.py made the connection to MongoDB, manager.py will already have access to the DB)
* Made sure the Frontend logic is correct: When submitting an existing job, it returns accuracy if the job is done.