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.

2/27/2024

* Write the backend endpoint to get all (or maximum 50) highest accuracy documents. **Querying from the database is more efficient than sorting/filtering in the backend/frontend**
* At the moment, the cap of 50 is enforced on the app. User can’t specify the specific number of entries to show
* Met with an error from Python – RuntimeException: can't create new thread at interpreter shutdown 🡪 This is because Python 3.12 introduced a change that prevents creating new threads after the main thread has exited.
* Done writing the backend endpoint!

3/3/2024

* Rendering “get\_jobs” in the Frontend as a table 🡪 Use **MUI X DataGrid**
* Meeting an error of useState not updating. Specifically, it doesn’t seem like the object is updated fast enough to be rendered. This is likely due to the asynchronous nature of React, where the code keeps running without waiting for a line to finish. 🡪 Turns out to be a problem with installing npm packages in the wrong place. I simply had to delete the incorrect installation and install into the right place.
* Finished a minimal table that’s functional

3/4/2024

* Implemented feature: When a job is done training, the table will be automatically updated
* Currently working on styling and making the frontend looks nice 🡪 Learning to style with CSS and reading the MDN documentation, which is really helpful.
* There was a lot of asking chatGPT/Perplexity to point me to the right resources, so I’m not sure if I’m learning anything significant.

3/5/2024

* Refactored Frontend code to be more component-oriented.
* Started looking into rendering an actual progress bar
  + Learning conditional rendering/dynamic rendering because we wouldn’t want the progress bar to be re-rendered every time a benchmark is achieved.
  + Learned that with useState, even if the component doesn’t change at all, will **trigger a re-render**.
  + Added another piece of information to emit in **mnist\_model.py**.
* Noticed a problem: The DataGrid is highlighting top 3 rows on each page, while we only want to highlight top 3 rows of the first page 🡪 Will fix later

3/6/2024

* Noticed a problem: Users can click outside of the form to trigger the input fields 🡪 Don’t know how to solve this yet
* Started working on the message board 🡪 There will be 2 components: MessageBoard.js and Message.js
* Currently debugging why messages get overwritten. Specifically, when I call setMessages() twice in a function, the first message doesn’t have enough time to be set. Instead, it gets overwritten by the second message 🡪 The solution is to use **the setState’s function’s callback**. **Using the setState function’s callback ensures that we work with the most up-to-date state**
* Some messages we might want to show the user:
  + Submission confirmation
  + Job already exist 🡪 return calculated accuracy/Job in queue waiting for process
  + When a job finishes, should tell the user run time + accuracy. It will look like this: “job epochs, learning\_rate, batch\_size done with accuracy + run\_time 🡪 This will be done via websocket, as the backend will have to signal that a job is done for the frontend to find that job in the database 🡪 **Need another endpoint reserved for looking up finished job and report the accuracy**.
* Talked to anh Sơn. Anh Sơn khuyên đừng dùng Material UI tại khi cần sẽ rất khó customize. Khuyên là nên code chay hoặc dùng bootstrap, shadcn UI
* Plan for tomorrow: Design the flex box layout first, then build the components

3/7/2024

* Designed flexbox layout
* Playing with css code to layout the flexboxes 🡪 Learning flexbox is actually fun!
* Finished sketching out the borders
* Started implementing the components

3/11/2024

* Learned that it is **not recommended to use class and className at the same time**, as the behavior would be unpredictable
* Finished Form component and Progress Bar components
* Currently working on the DataTable, switched to only displaying the top 10. The code for coloring is currently not optimal (styling is being applied to separate cells, while it could/should be applied to a whole row)
* class là invalid syntax, className mới đúng 🡪 Chỉ dùng className moving forward.
* Limit the size of “messages” state to 10 elements.
* Currently working on the MessageBoard
* Added a timestamp to each message.
* Building a mini-component: A timer to show elapsed runtime
* Building a mini-component: A spinning gear (icon)

3/12/2024

* Applying styling to “empty” renders.
* Tweaking message contents for more meaningful information.
* **Building an additional functionality**: When a job is done, socketIO should emit a signal to let the Frontend knows this job is done. Frontend should then find that job to report accuracy and update the table.
* Noticed a discrepancy between train time and time recorded in the table 🡪 Time recorded in the table is train time + evaluate time 🡪 Fixed so that it’s only train time
* Encountered an error: "Failed to execute 'fetch' on 'Window': Request with GET/HEAD method cannot have body," 🡪 Learned that “fetch” API doesn’t allow a request body to be included with GET or HEAD methods. A workaround would be to use **URLSearchParams()** and convert a json into query parameters, then add it directly to the url, something like this: `http://localhost:9000/find-job?${queryParams}`. The backend also needs to change, we need to use **request.args.get(key)** to access the parameters.