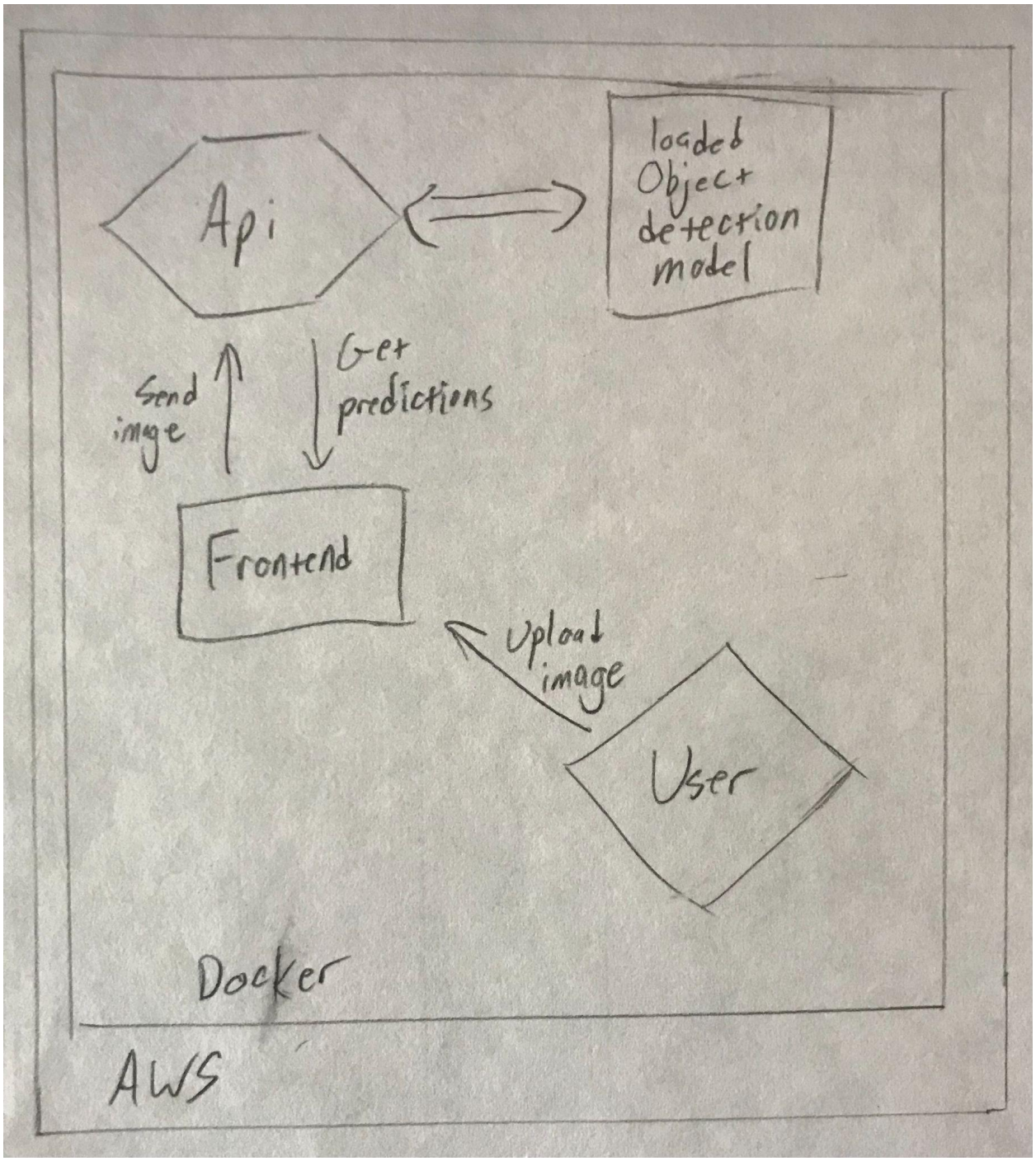
Deployment Solution Architecture

Simple flowchart of interactions between user, frontend, backend, and model.



Components of the system

The major components of my system will be a frontend web application built with React and Bulma, as well as a backend api built with Flask and PyTorch. The user will be able to upload images of a set size, and request predictions from the PyTorch object detection model. PyTorch predictions are returned as a dictionary of bounding box labels, coordinates, and confidence scores. These will then be overlaid on the image the user uploaded.

Data

Expecting an input image size of 416x416x3, this will be stored in a state variable, and will therefore be removed on refreshing the page. At the moment, I am not planning on storing images in a database, but that can be added later.

The model will be stored with the project backend, so that it may be loaded when serving api requests.

Data will be uploaded to the frontend through a user file upload, and will be sent to the api using an HTTP GET request.

Model Lifecycle

A pretrained object detection model will be used for this project initially. The performance of these models vary depending on the size of the model, the intensity of the method used, and the quality of the images submitted to the model for prediction.

Rather than routinely retraining the model if poor performance is observed, it would likely be better to swap out the current model with a

stronger one. This will likely come with memory/runtime costs, and will require extensive trial and error, so will not be attempted until a Minimum Viable Product is complete.

If monitoring the performance of my web app is necessary, it would be good to add features for storing previous user images in a database, and allowing for user feedback on each prediction, as I have no other way of knowing if the model is returning accurate labels.

If that dataset is created and cleaned, PyTorch's mean Average Precision metric is very thorough in a model's performance with different confidence intervals and object sizes. This metric could then be used to compare a new model's viability compared to the current model.

System Error Handling

Errors with the api will likely be invisible to the user, so it may be helpful to dump Flask session logs to an output error or log file. Whereas errors on the frontend side will be displayed in the console, and may sometimes be easier to debug if the problem is easily replicable.

As I'm going to host this on AWS as an as-needed api, I'm hoping that means each api call will be self-contained, and errors with one response won't affect any of the other calls. This may need more research.

Estimated Cost

Using AWS, I'm expecting a cost of fractions of a penny per api call. And I don't expect the cost of the memory usage of my model to be very expensive under one of AWS cheaper tiers.