ML Deployment and Monitoring Service

Nikolai Lyssogor, Ashwin Viswamithiran, Caleb Starkey (Not in CSCI 5253)

Project Goals

We built a service to deploy machine learning models so that they can take requests in production. That is, once a user has finished training a model, they can use our service to spin up a server that can serve prediction requests using their model. We also built a feature that allows users to view some simple statistics relating to how the distributions of training versus production data differ, as this can be an indication that a deployed model may need to be retrained on newer data.

Software and Hardware Components

RPC / API Interfaces: We used REST API to make requests to create, delete, or update deployments, since this happens relatively infrequently. Model deployments themselves will use gRPC since speed matters much more in that context.

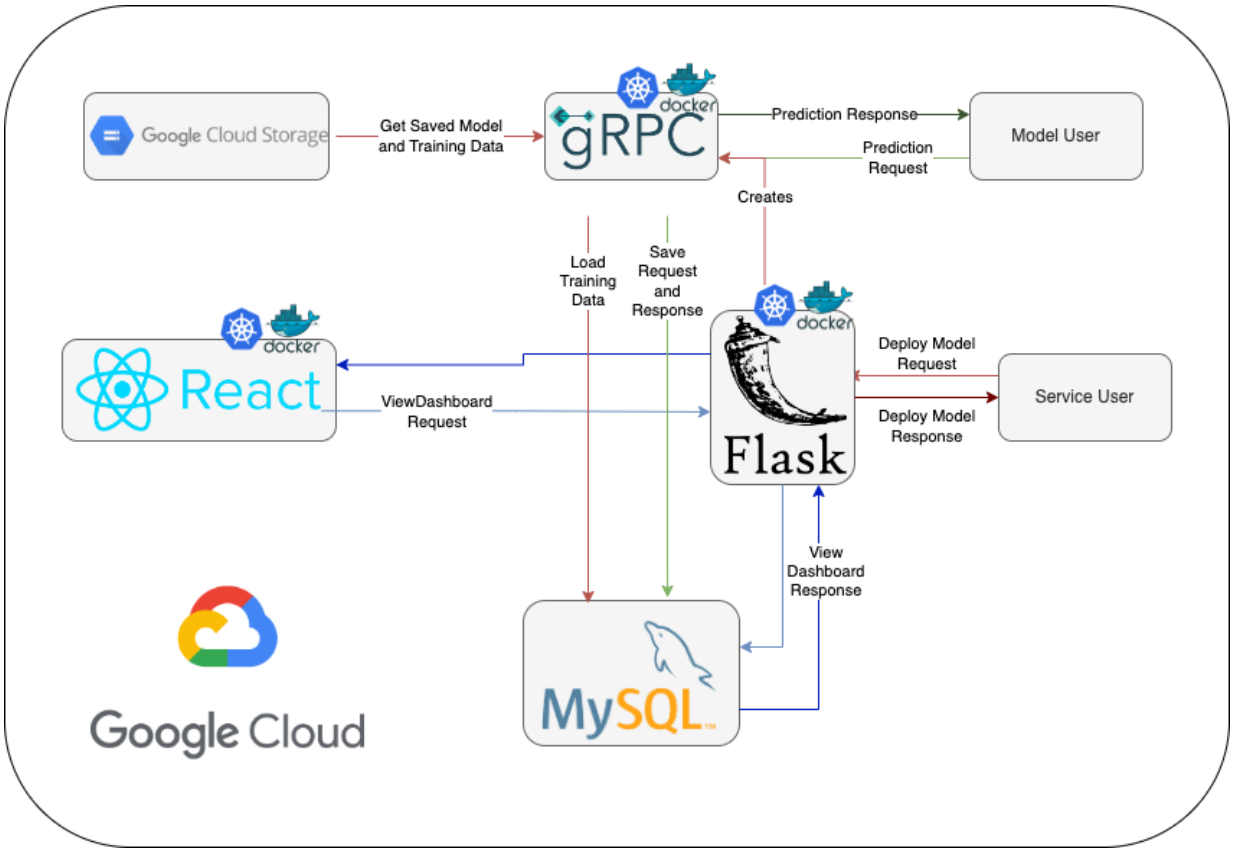
Databases: The information (metadata) about each machine learning model will be stored in a table in MySQL database in a cloudSQL instance. Along with this, each model deployments' training and production data is stored in a different table for each model that is deployed.

Virtual Machines and Containers: Each software component depicted in the architecture diagram below runs in a container, on a virtual machine, managed by Kubernetes. We used different machine types for different components.

Storage Services: To have a model deployed and monitored, users will send a request containing the location of a model and its training data in a Google Cloud object storage bucket.

Cloud Services: We have used GCP's Cloud Build and Artifact Registry. When we receive a request from a user, Google Cloud Build will build a container image that has the model, dataset, and files needed to run the server. Once the image is built, it is pushed to Google Artifact Registry and then deployed in our Kubernetes cluster.

Architecture Diagram:



Software and Hardware Interaction

There are three main interactions with this system: Deploying a model, viewing the performance dashboard for that model, and sending a prediction request to that deployed model. We will overview each of them here.

When a user wants to deploy a model, they will send a prediction request to the Flask server containing the Google Cloud object storage location of the saved model and the JSON-formatted dataset that was used to train the model. When the Flask server receives this request, it will build a container image using Google Cloud Build that has the model, dataset, and files needed to run the server. It builds on top of a base image that has some large dependencies that all model servers will need, such as Tensorflow. Once the image is built, it is pushed to Google Artifact Registry and then deployed in our Kubernetes cluster. When the gRPC server running in the container is initialized, it first loads the saved model, then parses the JSON training dataset, creates a new table for that server in Cloud SQL, and loads the dataset into that table. Meanwhile, the Flask server is waiting for all of this to happen. When it is finished, it returns to the user the status of the deployed server and the public IP address of the server, so that users can make requests to it.

When a prediction request is sent over to the newly-deployed gRPC server, its model makes a prediction and then stores that prediction along with the request into MySQL before returning the prediction to the user.

When a user wants to view the performance dashboard of their model, from the home page they will click on the name of the model they are interested in. The React server will send a request to the Flask server which makes a query to the database table corresponding to that model. After retrieving the data, it computes some column-wise statistics on the training and production slices of the data, and sends the result back to the React server to get displayed.

Testing and Debugging

All of the React code was developed locally and then containerized and deployed to Kubernetes at the very end. For the Flask and gRPC servers, our development process consisted of testing specific functions in a REPL, then testing servers locally, then testing servers in the cloud. To make testing the cloud easier, we built the images using Cloud Build rather than on our own machines, which in one of our cases used a different hardware platform, leading to confounding errors in Docker. Kubernetes pod logs were also tremendously helpful in debugging services in the cloud.

Bottlenecks

Spinning up the gRPC servers is one aspect of the system that was quite slow, and there are a few things we could have done to speed this up. First, we could have been more clever about how we cached images. The gRPC servers had to pull the base image with all the dependencies every time they got built. There is probably a way to keep that base image on the server so that it doesn't need to get downloaded every time. Second, instead of using the full GCP CLI to download the models and training data files to the gRPC servers, we

could have used a Python client with a much smaller dependency. Lastly, we do the processing to compute the statistics for the dashboard in Python on the Flask server because the SQL query required to do these calculations would have been pretty hairy. This was fine for the relatively small dataset we were testing with, but would most likely be problematic for a much larger one.