After exploring the model’s behavior and the features, I deciphered that there are three major components of the model that we have to focus on while making it scalable and serving it to a bigger user base.

1. **Maintaining Model Performance:** I prioritized ensuring consistent prediction quality for similar data, even as the user base grows. This meant mitigating potential accuracy drops under increasing load. Therefore, I have used the same mean values used during training for imputation, and used the same Standard Scaler for the purposes of standardization, besides of course pickling the finalized model.
2. **Preserving Data Quality:** Secondly, we have to make sure that quality of data does not deteriorate on reading internally within the application. To ensure no degradation of quality, I have made sure to handle all edge cases where the number of features input to the model is constant. This involves handling missing values, different data types as well as outliers. While the model is susceptible to outliers, the standard scalers do a pretty good job of normalizing the data. **Limitation:** Addressing missing features was deferred due to the focus on containerization. I plan to revisit this aspect in future iterations.
3. **Enabling Efficient Model Replication:** I aimed to facilitate seamless model replication to deliver timely responses to end users. This involved strategies to efficiently create and deploy additional instances of the model to handle growing demand.

For handling massive influx of data, it is always wise to modularize each component of the pipeline, especially pre-processing, model-predictions, and post-processing. While predictions and post-processing are handled together due to relatively lighter model, separation of these two components becomes indispensible for larger and heavier models like LLMs and Transformers.

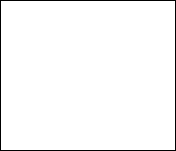
Application can be “Dockerized” (containerized). For containerization of the application, we create layers of images.



Container

Application Mongo DB

……………………..



Dependencies



Python

Linux

**Potential Enhancements:**

* **Adaptive Batching:** I plan to implement adaptive batching techniques to process multiple requests simultaneously, leveraging GPU capabilities for efficiency. This can significantly improve throughput and reduce response times under heavy load.
* **Metrics Endpoint:** I intend to integrate a "metrics" endpoint to provide valuable insights into server performance, including request volume, processing times, and other relevant metrics. This information will be crucial for further optimization and performance monitoring.

After Implementing FastAPI for getting response. When executed directly from the venv, and POST request can be observed as functional on local machine, for one request (a dict) or a batch of requests (a list of dicts style json).  
  
I then decided to containerize the application. But before that I save dependencies on requirements.txt file using pip freeze > requirements.txt (this has since been modified.

Next step is to dockerize the predictor application created using FAST API. It is during this step and on repeated failure on creation of the docker image that I realized that FastAPI is not compatible with python 3.7, which is the version in which the model was trained. For this reason I chose to go with a base image of 3.8.10-slim, as that is the lowest version on which FastAPI is still supported and requires minimum deviation from the package versions in GLM Model 26.ipynb.

The reason I opted for FastAPI in the first place is because of an experimentation within my current organization involving testing the NRT tool of the BERT based Question Answering Model. This experiment revealed that FastAPI was better performance for smaller loads such as the one we are dealing with, than other APIs like Flask (even though flask API did prove to be more stable for over 16 concurrent users)

After writing the commands to install the dependencies on Dockerfile, I created the .docignore file to leave behind the environment related files and folders. Next I created a docker-compose.yaml for container testing purposes, named our service –“web”, and included the uvicorn command discussed earlier. For testing, I mapped the host system’s 8000 port to the container’s 8000 port, and then built the container to see if it works using  
docker compose up –build  
  
To test if the hot reloading works, we have to sync our host machine with the docker container so the uvicorn server can listen to the changes in the files. For that we have to mount the volume in Docker compose file as  
 volumes:

* . :/app

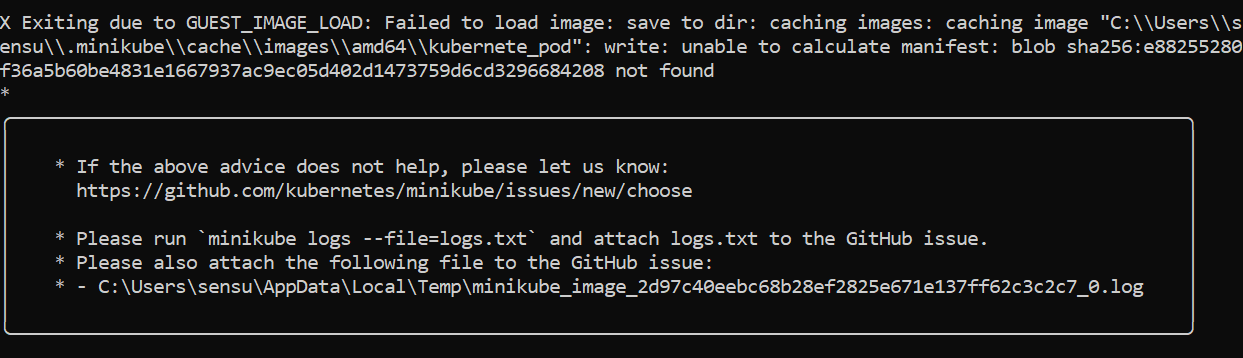
And restart the docker container to have the server up again. On verifying functionality, the image is finally created with all pkg versions compatible with the model, using:

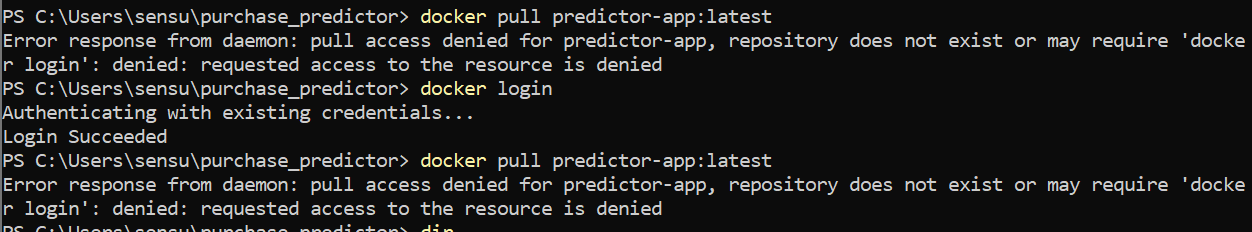
docker build -t predictor-app .

by running docker ps we can see how many containers are running  
We can also observer how we are accessing from the host port to the container port. Since in the app.py file I mentioned host=”0.0.0.0” , we can use the localhost or 127.0.0.1 or we can also use local IP address.

After stopping the docker image using docker stop 67b8c8ffc609 (Container ID)  
  
With the curl command it runs exactly like the raw API, but now we’ve managed to containerize the whole application with all its dependencies!

**Curl -X 'POST' 'http://127.0.0.1:8080/predict' -H 'accept: application/json' -H 'Content-Type: application/json' -d '**{"features":[{"x0":0.042317,"x1":-3.3447210000000003,"x2":4.635124212161472,"x3":-0.5983959993003629,"x4":-0.6477715045570444,"x5":"monday","x6":0.184902,"x7":46.690015,"x8":3.034132,"x9":0.364704,"x10":14.260733,"x11":-1.559332,"x12":"$5,547.78","x13":0.520324,"x14":31.212255,"x15":4.891671,"x16":0.357763,"x17":14.766366,"x18":-17.467243,"x19":0.224628,"x20":0.096752,"x21":1.305564,"x22":0.353632,"x23":3.909028,"x24":-91.273052,"x25":1.396952,"x26":4.401593,"x27":0.443086,"x28":14.048787,"x29":-0.932243,"x30":5.255472,"x31":"germany","x32":0.54199153,"x33":2.98948039,"x34":-1.78334189,"x35":0.80127315,"x36":-2.60231221,"x37":3.39682926,"x38":-1.22322646,"x39":-2.20977636,"x40":-68.69,"x41":522.25,"x42":-428.69,"x43":381.37,"x44":0.0197503,"x45":0.75116479,"x46":0.8630479007977094,"x47":-1.0383166613479036,"x48":-0.27261876352216863,"x49":-0.3430207259042951,"x50":0.31090086655652394,"x51":-0.7978419740300581,"x52":-2.0390175152938923,"x53":0.87182889,"x54":0.14373012,"x55":-1.15212514,"x56":-2.1703139704,"x57":-0.26784296202800567,"x58":0.21211063295318527,"x59":1.6926559406621045,"x60":-0.9522767913493433,"x61":-0.8625864974123282,"x62":0.07484871579558282,"x63":"36.29%","x64":3.47125327,"x65":-3.16656509,"x66":0.65446814,"x67":14.60067029,"x68":-20.57521013,"x69":0.71083785,"x70":0.16983767,"x71":0.55082127,"x72":0.62814576,"x73":3.38608078,"x74":-112.45263714,"x75":1.48370808,"x76":1.77035368,"x77":0.75702363,"x78":14.75731742,"x79":-0.62550355,"x80":"NaN","x81":"October","x82":"Female","x83":-0.7116680715420765,"x84":-0.26535598920105635,"x85":0.5175495906776872,"x86":-1.0881027091519955,"x87":-1.8188638198005096,"x88":-1.3584469526619427,"x89":-0.6549951949654567,"x90":-0.493304226235515,"x91":0.373853,"x92":0.94143481,"x93":3.54679834,"x94":-99.8574882,"x95":0.403926,"x96":1.65378726,"x97":0.00771459,"x98":-32.02164582,"x99":-60.3127828},{"x0":-1.03316,"x1":-0.34014,"x2":5.871823267093174,"x3":"NaN","x4":0.12213313902703374,"x5":"tuesday","x6":0.997773,"x7":51.581411,"x8":1.709219,"x9":0.844079,"x10":14.105233,"x11":-2.5433,"x12":"$-5,483.24","x13":0.28742,"x14":35.743492,"x15":4.415237,"x16":0.110435,"x17":14.944933,"x18":-18.755392,"x19":0.075041,"x20":0.214917,"x21":-4.123163,"x22":0.72766,"x23":3.006255,"x24":-113.387174,"x25":1.229528,"x26":3.748232,"x27":0.840611,"x28":14.441707,"x29":-0.120849,"x30":5.015867,"x31":"america","x32":0.821254,"x33":2.55354495,"x34":-1.84863863,"x35":1.6950307,"x36":1.51292076,"x37":3.51865516,"x38":-1.5228092,"x39":-2.2853103,"x40":-1576.44,"x41":-1109.88,"x42":-158.62,"x43":-1828.94,"x44":0.49520292,"x45":-2.20971925,"x46":-0.690305716163116,"x47":-1.2241275979842796,"x48":-1.2228860901205947,"x49":-1.8189779439382878,"x50":0.020347397416074088,"x51":2.3119441850454394,"x52":-0.07435673356064304,"x53":-0.0576265,"x54":0.1030348,"x55":0.59564248,"x56":1.626547456,"x57":-1.0763060909840458,"x58":0.591320350072612,"x59":-1.0877765187234452,"x60":1.27014337340356,"x61":0.6933246356337982,"x62":0.7255990172130561,"x63":"6.25%","x64":39.62417626,"x65":-0.34046576,"x66":0.81113762,"x67":14.36705547,"x68":-19.82612589,"x69":0.33424312,"x70":0.21591107,"x71":-4.88384906,"x72":0.76257143,"x73":3.63539031,"x74":-81.42302571,"x75":0.01061658,"x76":0.08064429,"x77":0.2150197,"x78":14.84293049,"x79":-2.10231995,"x80":6.35264328,"x81":"November","x82":"Male","x83":0.7997773236062501,"x84":-0.1304857222272298,"x85":-0.212689557461508,"x86":-0.284131119606305,"x87":-0.45520687219275524,"x88":-0.6754269968594849,"x89":-0.10510873880386888,"x90":0.5211188486643189,"x91":0.148424,"x92":0.92530093,"x93":3.83042567,"x94":-101.1057483,"x95":0.05577509,"x96":0.56488992,"x97":0.05171552,"x98":-32.5406118,"x99":-266.72579479},{"x0":2.029367,"x1":-3.2393009999999998,"x2":4.724436418197158,"x3":2.211831107546117,"x4":0.551611146361754,"x5":"tuesday","x6":0.492405,"x7":87.179042,"x8":4.333755,"x9":0.513789,"x10":14.317604,"x11":-3.31471,"x12":"$5,515.72","x13":0.295417,"x14":12.520591,"x15":4.0877,"x16":0.918884,"x17":14.743044,"x18":-20.548698,"x19":0.481351,"x20":0.179629,"x21":0.438461,"x22":0.491454,"x23":3.196985,"x24":-97.282469,"x25":1.192911,"x26":4.280097,"x27":0.30187,"x28":"","x29":-1.899614,"x30":"","x31":"germany","x32":0.48734341,"x33":2.82584466,"x34":-2.06608076,"x35":0.04402067,"x36":-0.49498194,"x37":3.59107998,"x38":-1.86201173,"x39":-0.64685302,"x40":-232.37,"x41":527.08,"x42":-9.46,"x43":582.15,"x44":0.9764149,"x45":-0.10400614,"x46":0.20532893791223,"x47":0.6360683088773246,"x48":0.9560161368002782,"x49":-0.16721893898638476,"x50":0.4640296781901955,"x51":2.37439498157146,"x52":0.6381296568457049,"x53":0.41124767,"x54":-0.97818721,"x55":0.77879075,"x56":-0.7632954603999998,"x57":-0.45660813031654224,"x58":0.5503756467463191,"x59":-0.13253421720719075,"x60":0.043386075596591675,"x61":0.8808301812718534,"x62":1.2922155590008249,"x63":"60.38%","x64":12.77168333,"x65":-1.74509859,"x66":0.90191424,"x67":14.37913333,"x68":-18.91037928,"x69":0.80395677,"x70":0.08923535,"x71":3.98044166,"x72":0.08806195,"x73":3.82664117,"x74":-98.52120473,"x75":0.93689644,"x76":0.07905681,"x77":0.02858207,"x78":"","x79":0.01852967,"x80":5.00093131,"x81":"July","x82":"Female","x83":0.2377332918442134,"x84":-0.0323918657232083,"x85":-1.1758573271462776,"x86":-1.3652342194057445,"x87":-1.1263427465053837,"x88":-1.1527352561391448,"x89":0.5137473000021247,"x90":0.15449155166013726,"x91":-0.034504,"x92":0.90404228,"x93":3.64296778,"x94":-107.4764874,"x95":1.04671797,"x96":1.49412263,"x97":0.23108405,"x98":-32.74095405,"x99":-4.3278873}]}’

For the purposes of this assignment, I am only creating a single node cluster on my laptop using minikube.  
  
Update (02/26/2024) due to setup issues with minikube and sign in problems with docker CLI, I was not able to create a cluster of pods within the allocated time. I am actively working on resolving the issues.  


but here are the commands for minikube implementation of single pod:

minikube start

kebctl get all -A #this will show what the cluster contains

Next, we load the docker daemon into the minikube docker daemon (the step that is causing problems): minikube image list

minikube image load predictor-app:latest # failing with error message after some time

minikube image *list*

All the pods will be created from this image

# next we create a deployment which will act as parent to all the pods

kubectl create deploy predictor-deploy --image=predictor-app:latest

kubectl get deploy   #1/1

kubectl get pod      # 1/1

We have one deployment *object*

and we also have one pod which was automatically created by this deployment. And what's important  is that here there is always random hash.  Deployments are used in a case where the  pods don't have any state so there is no notion of order. And if this given pod dies, then

the deployment will make sure there's a new one with a completely new hash. Next, we can

even create a service which will take all the pots we have in our deployment (which  is just one single pod in this case) and it will load balance the traffic among these pods.

We can check logs with: kubectl logs -f predictor-deploy-containerID

**Leveraging Kubernetes:**

* **Horizontal Pod Autoscaler (HPA):** I'll use HPA to automatically scale my application pods based on resource usage, ensuring it can handle increased traffic without performance drops.
* **Ingress Controller:** I'll integrate an ingress controller to manage external traffic routing, simplifying load balancing and enabling features like path-based routing and SSL termination.

**Optimizing the application:**

* **Caching:** I'm considering implementing caching mechanisms like Redis or Memcached to store frequently accessed data and reduce database load for repetitive API requests.
* **Asynchronous Processing:** For resource-intensive tasks like complex predictions, I might explore using asynchronous processing tools like Celery or RabbitMQ to prevent blocking the main request-response cycle and improve overall throughput.
* **Monitoring Tools:** I'll integrate monitoring tools like Prometheus and Grafana to track key metrics like application health, resource utilization, and API request volume, allowing me to proactively identify and troubleshoot potential issues.
* **Logging and Alerting:** I'll set up comprehensive logging and alerting systems to capture application logs, errors, and performance metrics, along with alerts notifying relevant teams of critical issues requiring immediate attention.

**Database optimization:**

* **Database Sharding:** For large datasets, I might consider sharding the database across multiple servers to distribute the load and improve performance. This approach would require careful planning and implementation.
* **Database Caching:** I'll explore utilizing database-specific caching mechanisms like query caching to reduce database load and improve response times.

**Utilizing additional technologies:**

* **API Gateway:** I might implement an API Gateway like Amazon API Gateway handle API requests, manage traffic flow, and enforce security policies. This can offload processing from my application and improve scalability.
* **Content Delivery Networks (CDNs):** For geographically distributed user bases, we can consider using a CDN to serve static content like images and JavaScript closer to users, reducing latency and enhancing user experience.

I intend to continuously monitor and analyze the application's performance to identify bottlenecks and further refine the deployment strategy for optimal scalability and reliability in an enterprise production environment