# NYC Taxi Demand Prediction with Weather and Flight Data Integration

## Value Proposition

**NYC Taxi Demand Prediction with Weather and Flight Data Integration** is a project aimed at improving urban mobility through better forecasting of taxi demand. Using historical NYC Taxi and Limousine Commission (TLC) trip records enriched with weather and flight arrival data, we will predict hourly taxi pickup counts in New York City. By anticipating surges or lulls in demand (for example, due to rain or incoming flights), taxi fleets and dispatchers can proactively allocate drivers, reducing passenger wait times and avoiding oversupply in low-demand periods. This leads to more efficient service for passengers and higher utilization for drivers, addressing urban transit needs.

The business impact is significant i.e. better demand forecasts mean improved rider satisfaction and driver revenue, and city authorities can use these insights for traffic management. We will evaluate our models with Root Mean Square Error as the primary metric, to quantitatively measure prediction accuracy in number of trips. A lower RMSE on held-out data will indicate a better model, and this metric directly ties to business goals by representing how close our predictions are to actual taxi usage.

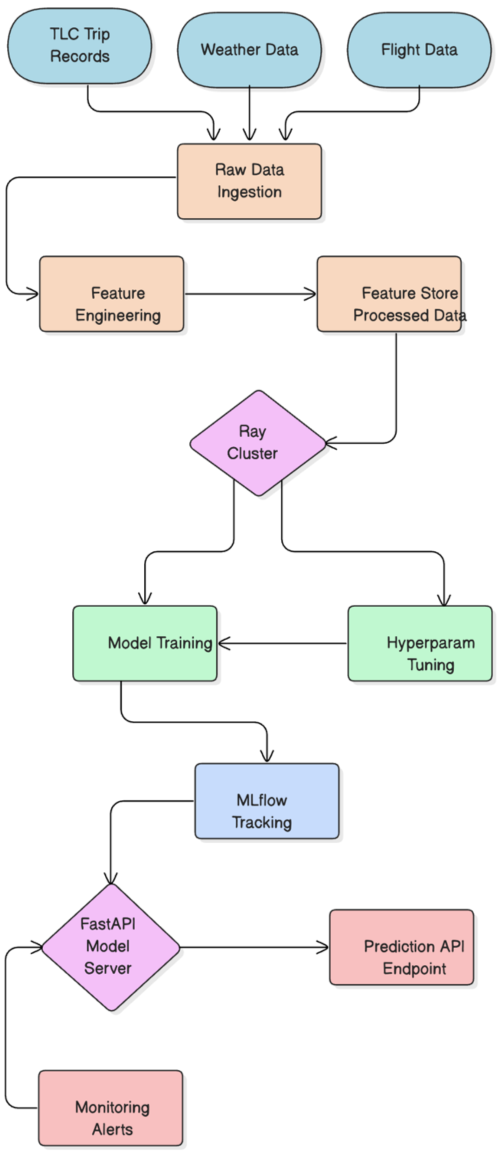
**Contributors**

| Name | Role | Course Unit Focus |
|---|---|---|
| Jeel Patel | **Model Training** | Units 4 & 5 – Model development, feature engineering, and hyperparameter tuning. |
| Shashank Dugad | **Serving & Monitoring** | Units 6 & 7 – Developing the FastAPI service for model serving, and seting up logging/monitoring. |
| Amarnadh Reddy | **Data Pipeline** | Unit 8 – Designing and implementing data ingestion and processing pipelines for taxis, weather, flights. |
| Anshi Shah | **CI/CD & Infrastructure** | Unit 3 – Handling infrastructure-as-code, continuous integration, and deployment automation. |

Note: All team members will collaborate on system integration, testing, and documentation to ensure the components work seamlessly together.

## System Diagram

The diagram below shows the end-to-end system architecture, including data sources, the ETL pipeline, the training cluster, model registry, and the serving and monitoring components:



**Explanation:** TLC, weather, and flight data flow into an **ETL pipeline** which processes raw inputs into engineered features stored in a feature table. A **Ray cluster** trains the ML model and logs results to **MLflow**. The best model is packaged in a **FastAPI** service inside a Docker container for deployment. End-users or systems can send requests to a **prediction API endpoint**, and the service returns predicted taxi demand. A **monitoring module** collects logs and metrics from the running service and triggers alerts in case of data drift or performance issues. The **CI/CD pipeline** underpins everything: infrastructure is provisioned as code, model training can be scheduled, and new model versions are deployed via canary releases on the FastAPI service.

## Data Sources

Our project integrates three external datasets. The table below summarizes each source, its origin, licensing and ethical considerations, and usage conditions:

| Data Source | Origin & Description | Licensing | Ethical & Usage Considerations |
|---|---|---|---|
| **NYC TLC Taxi Trip Records** | Provided by NYC Taxi & Limousine Commission. This dataset contains detailed records of taxi trips in NYC from 2009 onward Each record includes pickup/drop-off timestamps and locations, trip distance, fare, payment type, etc. We use this data to derive hourly taxi demand. | Public NYC Open Data. Usage is governed by NYC Open Data terms of use. TLC provides the data "as is" without guarantees of accuracy. | Data is aggregated and anonymized. Privacy risk is low, since individual passengers or drivers cannot be identified. We will handle it responsibly by using aggregate demand counts per hour. Ensure compliance with NYC Open Data terms. |
| **Weather Data (NYC)** | Sourced from Visual Crossing. Includes hourly historical weather observations for NYC. This data provides context on how weather conditions affect taxi demand. | **Visual Crossing API:** Freemium license – up to 1,000 weather data records per day are available for free. | Weather data has no personal data, so privacy concerns are minimal. We must respect API usage limits and attribution requirements. For Visual Crossing, if using the free tier, we'll ensure we stay within daily call limits and acknowledge Visual Crossing in our project. |
| **Flight Arrival Data (NYC)** | Obtained from the U.S. Bureau of Transportation Statistics **Airline On-Time Performance** data. This dataset logs every commercial flight in the US, including departure/arrival times and delays. From this, we will extract the number of flight arrivals to NYC airports each hour, as a feature to predict taxi demand. | Public dataset collected by the US Department of Transportation. As a government data source, it is free to use for the public. No explicit license required. | Flight data contains operational details but no personal passenger info, so privacy issues are negligible. Ethically, we use this data to improve transportation services. We will ensure data is used in aggregate and will update it periodically to reflect current travel patterns. |

## Infrastructure Requirements

We will utilize the **Chameleon Cloud** testbed to deploy our system. The table below outlines the cloud resources we plan to reserve on Chameleon, including virtual machine types, storage, and networking. For each resource, we list its purpose and the timeline of usage during the project:

| Resource | Specs / Size | Purpose | Usage Timeline |
|---|---|---|---|
| **Ray Cluster Head VM** | 1× VM (OpenStack flavor: m1.large – 4 VCPUs, 8 GB RAM) + 100 GB Volume storage | Acts as the head node for the Ray cluster and also runs data processing jobs. This VM will coordinate distributed training tasks and run the ETL pipeline code. It will also host services like the MLflow tracking server and Jupyter notebooks for development. The attached volume stores the datasets and experiment artifacts. | **Timeline:** Used throughout the project. Provisioned early for data preparation and model development (Units 4-5), and remains active to run periodic retraining jobs and coordinate workers. |
| **Ray Worker VMs** | 2× VMs (m1.xlarge – 8 VCPUs, 16 GB RAM each) | These are worker nodes in the Ray cluster for distributed training. They execute model training tasks and hyperparameter tuning trials in parallel. With two workers, we can parallelize experiments. More workers could be added if needed. | **Timeline:** Primarily used during model training & tuning phases (Unit 5). We will spin up the worker VMs when running intensive training or Ray Tune jobs, and they can be released or shut down when not in use to conserve resources. During serving phase, these may be idle or re-used for retraining jobs on schedule. |
| **Model Serving VM** | 1× VM (m1.small – 1 VCPU, 2 GB RAM) | Hosts the FastAPI application for serving predictions. This VM will run the Docker container that loads the trained model and listens for HTTP requests. The specs are modest because the prediction workload is light. It also handles a small volume of requests typical of a prototype service. Optionally, this VM can also run lightweight monitoring agents. | **Timeline:** Deployed after model training is complete, during the deployment phase (Units 6-7). It remains active 24/7 once the service is live, to respond to prediction queries and perform monitoring. It will continue to be used through the end of the project for serving the latest model. |
| **Monitoring/Logging VM** (optional) | 1× VM (m1.small – 1 VCPU, 2 GB RAM) | If needed, a separate VM to host the monitoring stack – e.g., Prometheus for scraping metrics, Grafana for | **Timeline:** Starts during deployment (Unit 7) if used. Runs continuously to collect metrics/logs while |

|  |  |  |  |
|---|---|---|---|
|  |  | dashboards, and a centralized log collector. In a small deployment, we might co-locate these on the Serving VM to save resources. However, isolating monitoring can be useful to avoid any interference with the model service performance. | the serving system is live. Not strictly required if we use managed services or the serving VM for monitoring, but available if needed for separation of concerns. |
| **Persistent Storage Volume** | 1× Volume (100 GB) attached to Head VM | Stores datasets and artifacts persistently. We will load the TLC trip data, weather data, and flight data onto this volume. Processed feature files and trained model artifacts will also reside here for persistence. Using a volume ensures data isn't lost if a VM is reprovisioned or rebooted. 100 GB is ample for several years of taxi data and associated features. | **Timeline:** Allocated at the start (Unit 4) when we begin data ingestion. Used throughout for reading/writing data in ETL and training. It remains mounted and in use through model development and serving. |
| **Floating IP** | 1× Public IP address | Allows external access to the FastAPI service and SSH access to VMs. The floating IP will be associated with the Serving VM. We may also assign a floating IP to the head VM for direct SSH and monitoring access. | **Timeline:** Allocated during the deployment phase when the service is about to go live (Unit 6). In use for the remainder of the project to enable API calls from outside the cloud and remote management. |
| **GPU Node** (optional) | 1× GPU-enabled VM (e.g., m1.xxxlarge with NVIDIA Tesla T4) | Not required for our baseline, but we include the option of a GPU node for experimentation. A GPU could accelerate training if we try deep learning models or if we use XGBoost's GPU training to speed up on very large data. This would be a single powerful VM with a GPU. | **Timeline:** Only used if needed in Unit 5 for experiments beyond the scope of the basic model. We would provision it on a short-term basis and release it afterwards. If not used, we will stick to CPU resources only. |

## Detailed Design Plan

### Model Training and Platform (Units 4 & 5)

- **ETL & Feature Engineering:** Our first step is transforming raw data into model-ready features. Using the TLC trip records, we aggregate trip counts per hour. We may focus on high-demand areas or do a citywide total demand; for simplicity, an initial approach is citywide hourly pickups. We then join **weather data** to each hourly record: for each hour timestamp, we attach weather features such as temperature, precipitation, snow, etc., retrieved from the Visual Crossing API or NOAA datasets. Similarly, we integrate **flight data** by adding a feature for the number of flights arriving in NYC during that hour. We also create time-based features: hour of day (0-23), day of week, whether the day is a holiday or not, and perhaps monthly seasonality indicators. Additionally, we include **lag features** – e.g., the previous hour's demand, and demand 24 hours prior – to help the model recognize temporal patterns. These features are assembled into a training dataset where each row corresponds to a specific hour with all known inputs and the output. Any missing data will be imputed or filled with reasonable defaults. The ETL pipeline performing these steps will be implemented in Python, using libraries like pandas or PySpark for efficiency. We will store intermediate results in a **Silver layer** to avoid re-processing raw data repeatedly.
- **Model Selection:** We plan to experiment with two models: a simple baseline and a more powerful regressor. The baseline can be a **Random Forest Regressor** trained on the feature set. This provides a benchmark to improve upon. Our main model will be **XGBoost**, a gradient boosting decision tree model known for strong performance on structured data. We choose XGBoost for its ability to model complex interactions and its efficiency on large datasets via boosting. XGBoost also has built-in handling for missing values and can naturally incorporate categorical features via one-hot encoding. We'll train the model to predict the next hour's demand given the features of the current and previous hours. The model training will be performed on the Ray cluster to leverage parallelism. We will split our data into training and validation sets to evaluate how well the model generalizes to unseen periods. The **evaluation metric** is RMSE – we will compute RMSE on the validation set to compare models. RMSE directly measures the typical error in number of trips. We will also monitor Mean Absolute Error for insight into median error.
- **Distributed Hyperparameter Tuning:** After establishing a working model, we will improve it through hyperparameter tuning using **Ray Tune**. Ray Tune allows us to launch multiple training trials in parallel across our cluster, each with different hyperparameters. Key XGBoost hyperparameters to tune include: max_depth of trees, learning_rate, n_estimators, subsample and colsample_bytree. We'll define a search space for these and use Ray Tune to try a combination of random search and Bayesian optimization to efficiently explore this space. Thanks to the Ray workers, we can run many trials concurrently. **Parallelizing hyperparameter search** drastically cuts down experimentation time. Ray Tune will manage the scheduling and resource allocation for each trial, and we'll specify a stopping criterion. For example, we might run 50 trials of XGBoost with different parameter settings in parallel on the cluster; Ray Tune will report back the best combination found. We will also try to tune the baseline Random Forest for fairness, though the emphasis is on optimizing XGBoost.

- **Experiment Tracking:** Throughout model development, we will use **MLflow** to track experiments. For each training run, we log parameters and performance metrics to MLflow. For instance, we record which features are used, the hyperparameters, training start/end times, and the resulting RMSE/MAE on validation data. MLflow will save these in a backend store and keep the trained model artifacts in an artifact store. By doing so, we ensure reproducibility: we can always refer back to a run ID and see what settings produced a given model. Once the hyperparameter tuning identifies the best model, we will **register** that model in MLflow's Model Registry. This registry entry can be versioned, so future models can be compared and we maintain a history of model versions. MLflow also helps with team collaboration, as all members can view the experiment results and compare notes on which features or parameters improved performance. We will take the top-performing model and prepare it for deployment.

## Model Serving and Monitoring (Units 6 & 7)

- **Serving Architecture:** The chosen model will be deployed as a RESTful service using **FastAPI**. We implement a /predict endpoint that, when called, will return the predicted taxi demand for a given input query. In a simple scenario, the service could always predict the next hour demand using the latest available features. A client might call /predict with no parameters and get the forecast for the upcoming hour in NYC, or optionally specify a future timestamp or location of interest. For our scope, we assume predictions are for the immediate next hour citywide, updated hourly. The FastAPI app loads the trained model artifact at startup. It also loads any necessary data normalization or encoding steps that were used in training. On each request, the API will gather the required features: in practice, this means we need access to current weather and flight information and recent demand. Since this is an hourly forecast, one strategy is to have the pipeline continuously update a feature row for the current hour, and the API simply reads that latest feature set to produce a prediction for the next hour. Alternatively, the API could itself fetch the latest weather/flight info from their sources in real time. For simplicity, we may decouple this by having a background job update a "live features" table every hour. The FastAPI service will respond with a JSON containing the predicted demand. This service allows any external system to get predictions on demand.
- **Containerization:** We will containerize the FastAPI service using Docker. The Docker image will encapsulate the Python environment, the FastAPI app code, and the model file. By containerizing, we ensure consistency between the development environment and deployment environment. The image will be built as part of our CI pipeline and tagged with a version. This image can then be run on the Serving VM. We will expose the FastAPI on a specific port and use the floating IP so it's reachable. If needed, we might use an Nginx reverse proxy in front for more robust routing, but given the low traffic in a prototype, it might not be necessary. The container will have resource limits set. We also consider scalability: if demand for predictions grew, we could run multiple containers and load-balance, but for now a single instance is sufficient. The FastAPI app will include basic error handling and a health check endpoint so we can easily monitor if it's up.

- **Logging & Monitoring:** In deployment, **monitoring is crucial** to catch issues and evaluate performance. We will instrument the FastAPI app with logging at each request. Each prediction request will generate a log entry containing the input and the output. These logs can be written to a file or stdout. For metrics, we integrate **Prometheus** for monitoring. We can use the prometheus-client library in Python to define custom metrics. For example, we maintain a counter for number of predictions made, a histogram for response latency, and perhaps a gauge for the latest predicted value. The FastAPI app can expose a /metrics endpoint where these metrics are available. Prometheus, running either on the same VM or a monitoring VM, will scrape this endpoint at intervals. We will monitor system metrics as well using Node Exporter or similar. With Prometheus data, we can set up **Grafana** dashboards to visualize key stats: e.g., a time-series of predicted vs actual demand, or the distribution of errors. Logging is also used for debugging; we'll adjust log levels to ensure we record warnings or errors. In addition, the model server could log the feature values it used for each prediction – this is useful for post-analysis and also for detecting anomalies.
- **Predictive Performance Tracking:** Since the ground truth will be known after the fact, we plan to implement **closed-loop monitoring** of the model's accuracy. This might be done offline: for instance, each day, when TLC releases the previous day's trips, we compare the predictions made for each hour to the actual outcomes. We can then compute error metrics for that day and log them. This can be automated with a scheduled job that runs daily, pulling yesterday's actuals and computing RMSE, MAE, and maybe a breakdown of errors during peak vs off-peak. Those results can be logged to MLflow or sent to a monitoring dashboard. Over time, this gives us a trend of model accuracy. If we see the error metrics creeping up, it's a signal that the model might be getting stale or that data patterns have changed. We will set thresholds for acceptable performance. This forms the basis of **continuous evaluation** of the model in production.
- **Data Drift Detection:** Beyond just tracking prediction error, we also keep an eye on input data drift (Unit 7 concept). Our model relies on certain statistical relationships learned from historical data; if the input distribution shifts, those relationships may no longer hold. We will implement a drift detection component focusing on key features:
  - For continuous features like temperature or flight count, we employ a two-sample **Kolmogorov–Smirnov (K-S) test** comparing the distribution of recent values to the historical distribution. The K-S test is non-parametric and sensitive to differences in both the shape and location of distributions, and it yields a p-value indicating if differences are likely due to chance. For example, we can take the last 30 days of hourly temperatures used by the model and compare it to the temperature distribution in our training dataset; a significant p-value would suggest a drift.
  - For categorical/time features, we can monitor their frequency. E.g., if suddenly we get a lot more "holiday" hours not represented well in training, that's a context drift.
  - We will also monitor the relationship between features and the target in recent data. This is more complex, but for instance, if the correlation between flight arrivals and taxi pickups changes drastically, the model might need adjustment.

  Implementation-wise, a small **monitoring script** can run periodically to perform these statistical tests using recent logged data. If drift is detected beyond a threshold, it will trigger an alert indicating which feature drifted. We might use an existing library or service for drift detection, but given our scope, a simple statistical test approach is sufficient to demonstrate the concept. Detecting drift gives us advance notice that the model may require retraining or revision.

- **Alerts and Incident Response:** We connect our monitoring to an alerting mechanism. Using Prometheus Alertmanager or simple custom scripts, we will set up notifications for conditions such as:
  - **Service downtime:** if the FastAPI service is unreachable or if HTTP error rates go above certain level.
  - **High latency or errors:** if prediction requests are consistently slow or erroring, indicating something is wrong.
  - **Data pipeline failures:** if our streaming simulation or data updating job fails.
  - **Drift or accuracy alerts:** as discussed, if drift is detected or if daily RMSE exceeds a threshold.
- **Canary Deployment Strategy:** We plan to mitigate deployment risks by using a **canary release** approach for new models. Instead of immediately replacing the running model with a new version, we will deploy the new model in parallel and route a small portion of traffic to it initially. In practice, since our traffic is low and maybe not live user-driven, we can simulate this by sending a fraction of requests to the new model. Another method is "shadow mode," where the new model gets all the same inputs as the old model but its results are not used operationally – instead, we log and compare them; however, a true canary actually serves a subset of users. For our case, we can implement canary by running two instances of the FastAPI service: one with the old model (v1) and one with the new model (v2). The API gateway, we collect performance data: is v2's error lower, same, or higher? Are there any anomalies? Only after confirming the new model performs well and stable do we increase its traffic share to 50%, then 100%, essentially phasing out the old model. This incremental rollout ensures that if the new model has an issue, most users are unaffected and we can rollback easily by simply redirecting all traffic back to v1. In our CI/CD process, we'll automate the canary setup: deployment scripts will deploy the new container alongside the old one, and our testing harness can act as the "traffic splitter." While full production systems might use sophisticated tools for canary, our implementation will be simpler but conceptually identical. This addresses the risk of deploying untested models – we **test in production on a small scale** before full release, catching potential problems early. For example, if the new model's predictions lead to higher error or if it fails to respond for some inputs, we will notice that in the canary phase and hold off promotion. Only when it proves at least as good as the current model will it be fully deployed. Canary deployments thus give us confidence in continuously improving the model without causing service disruptions or regressions in accuracy.

## Data Pipeline (Unit 8)

- **Batch Data Ingestion:** We design our data pipeline to handle both one-time historical data load and streaming updates. Initially, we perform batch ingestion of historical data to build our first model. The pipeline will retrieve the TLC trip record data for the period we're interested in. These may come as monthly files which we can download from the NYC

TLC website or AWS Open Data bucket. The pipeline reads these files and writes them to our storage in a raw zone. Then it processes them to compute hourly aggregates. We will likely use Python with pandas for this if data volume is manageable. If needed, we can employ Spark on our Ray cluster or Dask for scaling. The **aggregation step** groups trips by pickup datetime and possibly by location. Each group's size is the demand count. We'll create a table where each row is a unique hour and columns for demand count and maybe breakdown by taxi type if that granularity helps. This becomes our base demand time series. We ensure time continuity. This batch process also ingests **external data**: for weather, we might call the Visual Crossing API in a loop for each day/hour in history. A better approach is to use their bulk query if available or use NOAA's Local Climatological Data for NYC which can be downloaded in bulk. We will get weather for the same timeframe. Similarly, we obtain flight data: BTS provides the on-time performance data which we can download as monthly CSVs. We will likely pre-compute an hourly flight arrivals time series for each airport. The pipeline then merges these datasets on the timestamp key. This merged dataset is written to the **Silver layer**. This forms the training data input for the modeling step described earlier.

- **Joining and Cleaning Data:** During the join, the pipeline will handle any misalignments or missing entries. For example, if weather data has missing hours, we interpolate or forward-fill as appropriate. Flight data may not have flights every hour, but that would just show as zero arrivals which is fine. We ensure time zones are consistent. We also add any derived features in this stage. For instance, from the raw timestamp we derive hour-of-day and day-of-week as separate columns. We may add a binary feature for "rush hour" to help model capture rush patterns. Weather categorical conditions could be one-hot encoded or turned into flags. The cleaning process also looks at anomalies: e.g., if an hour shows an extremely low or high demand compared to neighbors, is it an outlier or data error?. We might smooth or remove blatant errors if identified. We document any such cleaning. **Ethical note:** since this is aggregate mobility data, we ensure not to introduce bias – e.g., when we join external data, we treat all areas uniformly. The output of this stage is a coherent, cleaned dataset ready for analysis and modeling.
- **Streaming Simulation:** After the batch data build, our pipeline will shift to "streaming" mode to simulate real-time operation. In a real deployment, we would have streaming data inputs: a feed from TLC giving us trips in real-time, and possibly APIs for weather and flights giving updates frequently. To emulate this, we use our historical data but feed it in incrementally. One approach is to write a small scheduler that sleeps or ticks and processes data hour by hour. For example, suppose we want to simulate the model in production for January 2022 data. We would start at January 1, 2022 00:00: the pipeline would take all trips that happened between 00:00 and 00:59, count them, but for prediction of 1 AM, we need data up to midnight. There's a subtle point: predicting the next hour's demand typically could use the current hour's demand as a feature. But in real-time you don't know the current full hour's demand until the hour is over. We might structure it such that we predict one hour ahead using data up to the previous hour. That means at 00:00, we predict demand for 01:00 using data up to 00:00. This is a detail in how we include lag features. For our simulation, we can do the following: loop through each hour of the day – at hour H, assume we have all data for hour H. We then predict demand for H using data of H-1. Then when actual demand of H becomes available, we can record the error. This is essentially a rolling prediction. Our pipeline code can implement this loop. It will use the

model to generate predictions at each step, and then log those predictions alongside the actual. This will mimic how the system would work in production with live data. We can also simulate **streaming features**: e.g., have a data structure that holds "latest weather" and update it as we progress hour by hour. If we wanted to be more granular, we could even simulate minute-by-minute, but hourly is sufficient.

- **Pipeline Structure (Bronze/Silver/Gold):** We organize our data pipeline following a **medallion architecture**:
  - **Bronze layer (Raw data):** This layer contains raw, unprocessed data exactly as obtained from sources. For us, that means the original TLC trip record files, the raw weather API outputs, and raw flight data. We store these in a file system or database without modifications. This preserves a source of truth. For instance, all trip records for 2021 are kept in a bronze/taxi/2021/ directory in Parquet or CSV format. Raw weather data JSONs or CSVs are stored under bronze/weather/, etc. The pipeline ensures these files are accessible and records metadata for reproducibility.
  - **Silver layer (Cleaned & joined data):** In this layer, we transform and clean the data. The hourly aggregation of taxi trips is a silver-level product. We also clean the weather data. Then we join the datasets by time. The result—hourly records with all features and the target—is stored as the Silver output. We might have a table like silver/hourly_demand_nyc with columns. This table is the one the model actually trains on. Storing it means if we retrain or experiment with different modeling techniques, we don't need to recompute the joins each time. We maintain code to regenerate Silver if needed.
  - **Gold layer (Feature sets for ML):** In some definitions, the Silver we have might already be considered Gold since it's ready for ML. We can treat our joined dataset as Gold. Or we might create further aggregated gold data, for example, if we decide to aggregate by day or zone for some analysis. But most likely, **Gold = Silver** in our case. In MLOps, Gold could also refer to the final outputs of the model that are fed to business apps, but here we focus on input to model. We ensure the Gold data used in training is versioned. This ensures if the pipeline logic changes, we can compare the old and new feature sets.
- **Persistent Storage & Workflow:** The pipeline will run on our head node. It reads/writes from the attached volume where data resides. We will use directory partitioning in storage for efficiency: for example, partition trip data by year and month, as mentioned, so if we only need to update the last month, we process just that partition. We also consider using a small relational database for some data. However, given the time series nature, files might be enough. The pipeline code will be under version control. We will ensure that running the pipeline end-to-end is achievable with a single script or notebook, which can be re-run whenever new data arrives.
- **Integration with Training & Serving:** The output of the data pipeline (Gold data) feeds into model training (Units 4-5). When the system is deployed, the pipeline's streaming component effectively feeds the serving system: it keeps updating the latest features and actuals. If we implement it as a continuous process, it might run as a background service on the head node, sending data to the model service. Alternatively, if the serving service fetches data itself, the pipeline ensures that external data sources are queried and cached regularly. For example, a small job could hit the weather API every hour and save the latest

weather to a file that the FastAPI reads. This division of labor will be clearly defined to avoid duplication.

- **Testing the Pipeline:** We will test the pipeline with a subset of data to verify that joins and calculations are correct. We'll also verify that the timeline alignment is correct. These tests ensure our features truly reflect only past or concurrent information, not future. Once the pipeline is verified, we run it on the full dataset. The pipeline is an evolving component: as we add features, we can extend it. The layered design helps integrate new data sources with minimal disruption.

## Continuous Integration, Deployment & Infrastructure (Unit 3)

- **Infrastructure as Code:** To manage our various cloud resources and deployments, we adopt Infrastructure-as-Code practices. We will write configuration scripts describing the VMs, network, and storage needed. For example, a Terraform file will specify one openstack_compute_instance_v2 for the head node, two for workers, etc., and an openstack_blockstorage_volume_v2 for the 100GB volume, plus the openstack_compute_floatingip_associate_v2 to attach a floating IP. This allows us to spin up the entire environment in one command and tear it down when not needed, which is especially useful on a shared testbed like Chameleon with time-limited allocations. Alternatively, using the python-chi library, we can script the resource provisioning in Python. The benefit of python-chi is that it's tailored to Chameleon and can orchestrate experiments programmatically. We could write, for instance, a Python script that uses chi.server.create_instance(...) calls to create the VMs with desired specs, set up SSH keys, and configure the network. This script can be run as part of our CI pipeline or manually. We'll store these IaC scripts in our repo so that environment configuration is tracked. When someone new wants to run the project, they can use these scripts to allocate resources on Chameleon. Additionally, using IaC means any changes are done by editing the config and re-applying, avoiding configuration drift or undocumented changes.
- **Environment Configuration Automation:** Once the VMs are provisioned, we need to install the necessary software. We will automate this with either cloud-init scripts or Ansible playbooks. For example, we can have a shell script that updates apt, installs Python3, Docker engine, and pulls our code from GitHub. Particularly for the Ray cluster, we need to ensure Ray is installed on head and workers and that workers can connect to the head node. We'll automate that setup. All these steps can be encapsulated in our IaC process. By doing this, we minimize manual setup errors and make redeployment faster.
- **Continuous Integration (CI):** Our GitHub repository will have multiple components. We will set up **GitHub Actions** as our CI service. Key CI steps include:
    - Running **unit tests**: We will write tests for functions like data cleaning, for model training, and for the API. These tests will run on each push to the repo to catch regressions.
    - **Code style checks**: Use flake8 or black for Python code formatting to maintain consistency.
    - **Build Docker image**: We will have a Dockerfile for the FastAPI service. The CI can attempt to build this image to ensure it's working.

    - **Integration test** (if feasible): We might simulate a small end-to-end run within CI. For example, take 1 day of data, train a tiny model, deploy it locally in the CI runner, and query the API. This would be an extensive test and might be too slow for each push, but we could schedule it nightly or on specific branches.
- **Continuous Delivery & Deployment (CD):** For deployment, we integrate with our CI pipeline or a separate CD pipeline. After CI tests pass, we will trigger a deployment workflow. Because we're deploying on Chameleon and not a traditional cloud with easy APIs, our approach will likely be:
    1. Use the IaC scripts to ensure the infrastructure is up. This could be manual for initial setup, but later we assume the environment is running.
    2. Build and push the Docker image of the model server to a container registry. We might use Docker Hub or GitHub Packages for this. For example, the CD pipeline could build taxi_demand:latest and push it.
    3. Deploy the new container to the Serving VM. We can do this via an SSH action: the pipeline can SSH into the VM and run docker pull ... && docker run .... Another way is to use Ansible in the pipeline to do rolling update of the service.
    4. Once deployed, run a quick smoke test: the pipeline can call the /healthz endpoint of the API via the public IP to confirm the service is responding.
- **Scheduled Retraining (Continuous Training):** In addition to code and model deployment, MLOps involves continuous training when new data arrives. We plan to incorporate a **retraining schedule**. For example, schedule a workflow to run the model training pipeline every week or month. This job would:
    - Pull the latest TLC data and append to the Bronze data.
    - Regenerate the Silver/Gold feature set for that period.
    - Retrain the model on an updated window.
    - Evaluate the new model's metrics and compare with the current one.
    - If improved or necessary, trigger the deployment of this new model.

  We will likely implement a simplified version of this: perhaps manually triggering retraining as new data is available. But we will demonstrate the process. The CI/CD pipeline could integrate with this by treating the trained model artifact as part of the build – e.g., an Action that runs training and if a new model is better, it automatically goes into the Docker image for deployment.

- **CI/CD for Infrastructure:** The continuous processes aren't only for code; if we change infrastructure, we will update our Terraform and could even have Terraform automation run. However, often infra changes are applied manually in a controlled way. We will keep it under version control though. The idea is to treat infrastructure similarly to code – any changes are reviewed and applied systematically.
- **Canary and Blue-Green in CD:** As part of our CD pipeline, we incorporate the **canary logic**. When a new model image is deployed, the pipeline could automatically set it as "canary". For instance, we could have an environment variable in FastAPI like MODEL_VERSION and an extra endpoint or parameter to choose which model to use. The deployment might deploy the new model on a different endpoint. Our test suite could then hit both old and new endpoints with the same sample inputs and compare outputs. This could be an automated A/B test. If the difference in outputs is acceptable, then the

pipeline proceeds to switch traffic. This level of automation is advanced; due to time, we may end up doing some of this evaluation manually. Nonetheless, we strive to automate as much as possible to align with a true CI/CD practice where human intervention is minimal and mostly for approval steps.

- **Continuous Monitoring & Logging:** Finally, as part of operations (Unit 7), the continuous aspect extends to monitoring. We will ensure logs from the application and pipeline are aggregated. This way, the team can continuously monitor the system's health and performance. If any component crashes, we consider using a supervisor or systemd to auto-restart it, thus making the system self-healing to a degree.

## Difficulty Points Addressed

- **Distributed Hyperparameter Tuning with Ray (Unit 5):** We tackled the challenge of efficiently tuning model parameters by using a Ray cluster to parallelize hyperparameter search. Instead of serial grid search which could take an impractical amount of time, Ray Tune allows us to run many trials at once, leveraging multiple CPUs across nodes. This approach demonstrates how to scale model development horizontally. By doing so, we can explore a wide range of models and settings in a fraction of the time, ensuring we find a well-optimized model. This addresses the difficulty of tuning complex models on large data – our solution shows competence in distributed training frameworks and reflects Unit 5 concepts. We cited the benefit that distributed computing enables trying many hyperparam combinations in parallel to **tune faster without compromising quality**.
- **Data Drift Monitoring & Alerts (Unit 7):** The project includes a robust monitoring component that goes beyond basic uptime checks – we actively monitor the statistical properties of incoming data. By implementing drift detection, we directly address the problem of model degradation due to changing data. If the live data starts to differ significantly from the training data, our system will detect this and alert the team. This is a key Unit 7 difficulty: ensuring model reliability over time. We've planned for automated alerts on drift and performance drop, so we won't be caught off guard by silent model decay. In essence, the system is self-monitoring its predictive validity. This demonstrates our ability to implement data monitoring and trigger retraining or investigation when needed, which is critical in production ML systems. The use of the **K-S test** exemplifies one method to quantify drift for continuous features, fulfilling the requirement of incorporating statistical vigilance into our pipeline.
- **Canary Deployment & CI/CD Integration (Units 3 & 6):** To safely deploy new models, we integrated a **canary deployment strategy** into our CI/CD pipeline. This directly addresses the difficulty of pushing model updates without disrupting the service or regressing performance (Units 3 and 6). Our plan to release new model versions incrementally – first to a small subset of requests/users, then gradually increasing – allows us to verify the model in production before full rollout. We described how the CI/CD pipeline automates this process, deploying a new container alongside the old and routing traffic in a controlled manner. This shows we understand continuous delivery of ML: not just automating deployment, but doing so in a way that includes **continuous verification**. If the new model underperforms, we can quickly rollback with minimal impact. By

implementing canary releases, we demonstrate mastery of advanced deployment techniques. This approach significantly de-risks the ML release cycle, which is often a pain point in MLOps. We also tied this into our infrastructure automation, showing end-to-end capability from code commit to deployment and monitoring.

## References

1. NYC Taxi and Limousine Commission (TLC) Trip Records. NYC Open Data. Retrieved from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
   Data Source: Public dataset provided by the NYC Taxi & Limousine Commission, containing detailed records of taxi trips in New York City.
2. Visual Crossing Weather API. Visual Crossing. Retrieved from https://www.visualcrossing.com/weather-api
   Data Source: Historical weather observations for NYC. Free tier available for non-commercial use with a daily API call limit.
3. U.S. Bureau of Transportation Statistics (BTS) Airline On-Time Performance Data. U.S. Department of Transportation. Retrieved from https://www.bts.gov/
   Data Source: Public dataset that logs commercial flight performance, including flight arrival times for NYC airports.
4. Anyscale, Ray Tune Documentation. Retrieved from https://www.anyscale.com/
   Reference: Best practices in distributed hyperparameter tuning using Ray, enabling parallel model training and optimization.
5. NannyML, Data Drift Detection with Kolmogorov-Smirnov Test. Retrieved from https://www.nannyml.com/
   Reference: Implementation of statistical tests like the Kolmogorov-Smirnov (K-S) test for detecting data drift in predictive models.
6. MLflow Documentation. Retrieved from https://mlflow.org/
   Reference: Experiment tracking and model management using MLflow for managing training runs, logging hyperparameters, and versioning models.
7. Terraform Documentation. HashiCorp. Retrieved from https://www.terraform.io/docs
   Reference: Infrastructure-as-Code practices for automating cloud resource provisioning, ensuring consistency and repeatability in infrastructure setup.
8. Prometheus Monitoring Documentation. Retrieved from https://prometheus.io/docs/
   Reference: System monitoring and alerting using Prometheus for tracking model performance and detecting potential issues in real-time