# Comity Coding Challenge: High Resolution Weather Data

| | |
|---|---|
| Owner | Ⓐ Alex Favaro |
| Tags | |



## 📖 Background

At Comity, in service of our mission to make our energy systems more efficient and sustainable, we're building systems that unify data from a range of diverse and complex sources. A key factor affecting the dynamics of the energy system, both on the demand and increasingly the supply side, is weather and the climate. An important challenge for us,

therefore, is to integrate high quality weather forecasts into our system in a way that supports transparent analytics and reliable downstream modeling.

This challenge invites you to simulate a small part of our system by building a CLI tool that ingests and transforms a key weather forecasting dataset: the **NOAA High-Resolution Rapid Refresh (HRRR)** model, writing a program to ingest this data and work with it in a way that supports analytical use cases.

At Comity, we are also believers that emerging AI technology is fundamentally changing how we approach software development and quantitative research. As a result, we encourage you to use AI coding assistants such as ChatGPT, Cursor, Claude Code, etc. to complete this assignment. Nevertheless, you should view the assistants as collaborators and not crutches - your ability to organize, justify, and document your solution, as well as to explain how you used these tools, will be evaluated as much as your code. You are also welcome to use these tools to write the documentation accompanying your solution.

# 🎯 Goals

Build a CLI program that ingests and transforms a real-world weather forecasting dataset, the HRRR Forecasting Model, which contains high-resolution meteorological predictions published by NOAA daily. Your CLI should support:

- Storing data in a persistent DuckDB database
- Ingesting new data and updating the database according to the requirements below

# 👨‍💻 Requirements

## General

- Use any programming language, third-party libraries, and coding assistants that you want to implement your solution. In particular, you will almost certainly want to use an existing package for parsing the GRIB2 files in the raw HRRR dataset.
- Use DuckDB to persist data in a local file named `data.db`.
- Your code should be modular, readable, and easy to test and modify.

## HRRR Ingestion

- Ingest HRRR forecast data from the HRRR AWS Open Dataset in the GRIB2 format for a specified forecast run date, list of lat-long points, and specific set of variable names. You can assume that all provided points will be in the continental United States.

- Your solution should ingest **up to 48 hours** of forecasted data for the **06z** forecast run on each of the dates in the date range. You should ingest the **closest available point** in the forecast data for each input point.
- The HRRR data contains hundreds of variables. In practice we often care about only a subset of these, so to make your solution manageable you will only be required to support the following variables. The names that are used for these variables can be opaque and inconsistent among GRIB2 parsing packages, so we will specify below the name that should be used for each variable when you store the data. Part of your task is to figure out how to extract these variables from the data files. Documentation describing how these variables are referenced in the GRIB index files can be found here, though you should also use utilities such as those provided in the ecCodes package to inspect the contents of the files directly.

| Variable Name to Store | Description |
|---|---|
| `surface_pressure` | Surface pressure |
| `surface_roughness` | Surface roughness |
| `visible_beam_downward_solar_flux` | Visible beam downward solar flux |
| `visible_diffuse_downward_solar_flux` | Visible diffuse downward solar flux |
| `temperature_2m` | Temperature at 2m above ground |
| `dewpoint_2m` | Dew point temperature at 2m above ground |
| `relative_humidity_2m` | Relative humidity at 2m above ground |
| `u_component_wind_10m` | U-Component of wind at 10m above ground |
| `v_component_wind_10m` | V-Component of wind at 10m above ground |
| `u_component_wind_80m` | U-Component of wind at 80m above ground |
| `v_component_wind_80m` | V-Component of wind at 80m above ground |

- The ingestion command should be **idempotent** — i.e. it should skip ingestion of data that is already downloaded and avoid storing duplicate data.
- The data should be stored in a table named `hrrr_forecasts` with the following "long format" schema:

| Column | Type | Description |
|---|---|---|
| `valid_time_utc` | `TIMESTAMP` | A UTC timestamp of the "valid time" of the forecast, i.e. the timestamp of the forecasted data |
| `run_time_utc` | `TIMESTAMP` | A UTC timestamp of the "run time" of the forecast, i.e. the time at which the forecast was made |
| `latitude` | `FLOAT` | Latitude of the forecasted data |

| Column | Type | Description |
| --- | --- | --- |
| `longitude` | `FLOAT` | Longitude of the forecasted data |
| `variable` | `VARCHAR` | The human-readable variable name |
| `value` | `FLOAT` | The forecasted value |
| `source_s3` | `VARCHAR` | An S3 path to the file from which this data was sourced, of the form `s3://noaa-hrrr-bdp-pds.s3.amazonaws.com/hrrr.20250101/conus/hrrr.t06z.wrfsfcf00.grib2` |

## CLI

Your solution should be a command line program invoked as `hrrr-ingest [points.txt]` with the following options:

- `--run-date`: The forecast run date of the data to ingest. Defaults to the last available date with complete data.
- `--variables`: A comma separated list of variables to ingest. The variables should be passed using the human-readable names listed above. Defaults to all supported variables.
- `--num-hours`: Number of hours of forecast data to ingest. Defaults to 48. This will be useful for testing so that you can work with smaller amounts of data.

The command's only required argument is a text file with a list of lat-long pairs of points for which to ingest the data. For each of the input points your solution should ingest the point available in the forecast data which is closest to that point. An example input file is provided below:

```
31.006900,-88.010300
31.756900,-106.375000
32.583889,-86.283060
32.601700,-87.781100
32.618900,-86.254800
33.255300,-87.449500
33.425878,-86.337550
33.458665,-87.356820
33.784500,-86.052400
```

For example, to ingest 2 hours of forecast data including temperature at 2m and surface pressure for the forecast run date of April 24, 2025, you would run:

```
hrrr-ingest points.txt \
    --run-date 2025-04-24 \
    --variables temperature_2m,surface_pressure \
    --num-hours 2
```

# 📦 Deliverables

You must submit with your solution:

- The code implementing your solution in any language of your choice
- A README including instructions for setting up, building, and running your solution
- A write-up answering the questions below

## Write-Up Questions

1. How long did you spend working on the problem? What difficulties, if any, did you run into along the way?
2. Please list any AI assistants you used to complete your solution, along with a description of how you used them. Be specific about the key prompts that you used, any areas where you found the assistant got stuck and needed help, or places where you wrote skeleton code that you asked the assistant to complete, for example.
3. Describe how you would deploy your solution as a production service. How would you schedule the ingestion routines as new data becomes available? What data storage technology would you use to the make the data more readily available to analysts and researchers? What monitoring would you put in place to ensure system correctness?
4. As specified here, your solution will only ingest data for a single run date. How would you scale it up to support large-scale backfills of many data points across years worth of data? What performance improvements would you likely need to implement? Try to be as specific as possible.

# ⭐ Evaluation Criteria

Your solution will be evaluated based on the following criteria, in decreasing order of importance:

1. *Correctness* - Does your program correctly implement the specified CLI command and store the data in the right format? Is it properly idempotent?

2. *Clarity* - Is your code and write-up well-organized and easy to read?
3. *Adaptability* - Is your solution architected in a manner that makes it easy to understand and modify by other engineers, allowing them to add and change features in a consistent, testable way?
4. *Bonus Features* - Any ways in which your solution goes above and beyond the requirements outlined here.

## 📓 Notes and Suggestions

- As noted above, you are expected to use AI coding assistants to develop your solution. Make sure, however, that you record the prompts you used so that you can include them in your write-up.
- You are welcome to use any programming language you like.
- You are also welcome to use any third party package that you find useful. In particular, you should use an existing package for parsing the GRIB2 files in the raw dataset.