Write-Up Questions

1. How long did you spend working on the problem? What difficulties, if any, did you run into along the way?

I spent approximately 6–9 hours in total: around 4–5 hours understanding and experimenting with the GRIB2 data format, and 2–4 hours actively coding the CLI tool. An extra 1.5 hour for the documentations.

Difficulties:

One major difficulty I faced was working with the GRIB2 file structure and understanding how to extract variables using `cfgrib`. Initially, I tried using the `.idx` index file to download only the necessary byte ranges of the GRIB2 file to reduce bandwidth and memory usage. However, I decided to switch to downloading the full GRIB2 file for the following reasons:

- If multiple variables are needed, the partial-download approach requires `n` HTTP range requests per forecast file, which is inefficient.
- Caching partial GRIB2 files adds considerable complexity compared to caching complete files.

I chose full-file downloads as a simpler and more scalable default, but retained the partial-fetch logic as an alternative implementation worth pursuing if file size or bandwidth becomes a bottleneck later on.

Another challenge was understanding how GRIB variables are interpreted and transformed during processing. For example, the wind component variable `10v` ends up being renamed to `v10` in the output dataset — not due to inconsistent naming in the GRIB file itself, but due to how `cfgrib` maps parameters internally. This kind of behavior, along with interpreting GRIB metadata correctly and verifying that the extracted values made sense, took time to fully understand.

---

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.

I used the following AI assistants during development:

1. Gemini Gemini was my primary assistant for conceptual guidance. It was especially helpful in navigating unfamiliar packages such as `cfgrib`, `xarray`, and the structure of GRIB2 data. Specifically, it helped me:

- Understand the overall architecture of GRIB data ingestion by clarifying how the GRIB2 file, `.idx` index file, `cfgrib`, HRRR GRIB2 inventory, and ECMWF parameter database fit together.
- Interpret the HRRR GRIB inventory documentation and explore GRIB content using tools like `grib_ls`.
- Identify quirks in variable handling. e.g., understanding why `10v` appears as `v10` after parsing, not due to naming inconsistencies but due to internal processing.
- Use `shortName` attributes to selectively access specific GRIB fields.
- Sanity-check my overall approach to ensure it aligned with the intended goal.

In short, Gemini was valuable for unblocking me when I was stuck or unsure, and for accelerating my learning curve with the GRIB2 ecosystem.

2. Local models (Gemma 3-4B-IT-QAT and Qwen2.5-Coder:1.5B)

I also experimented with running these models locally for code completion. This was my first time using them in a "real" project. They were helpful for generating basic boilerplate and small snippets. However, their utility was limited compared to tools like GitHub Copilot.

3. ChatGPT

Because English isn't my first language, I almost ALWAYS use ChatGPT with the prompt `fix grammar: {whatever text I have}`, which is really helpful for me."

---

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?

For deployment, I'd containerize the ingestion logic with Docker and run it via something lightweight like **Airflow**, or even **cron** jobs using **AWS Lambda** or **Fargate**, depending on needs.

Scheduling: Jobs can run every few hours or get triggered when new HRRR data becomes available. The approach depends on how the upstream data is released.

Storage: I'd use **PostgreSQL** + **PostGIS** for geospatial support, hosted on **RDS** for simplicity. If traffic increases, a read-replica setup can help separate ingestion from analyst queries. I'd also plan for **indexing**, though which fields to index would depend on how the data ends up being queried.

Monitoring: Include basic validation checks (like variable presence, value sanity, and grid alignment), along with logs and alerts using tools such as **Sentry** or **CloudWatch**. I'd also track metrics like ingestion time and failure rates using **Prometheus**.

---

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?

Scaling Strategy (Prioritized by Impact)

1. Download Parallelization (First Bottleneck) The biggest bottleneck is downloading. Each GRIB2 file is about 100–200 MB, and there are 48 files per run. For a year of data, that adds up to over 17,000 files. Even if parsing is slow, downloads can dominate runtime without concurrency.

To speed this up, we can parallelize downloads:

- Run multiple ingestion jobs at the same time, each handling different dates or forecast hours
- In a cloud setup, I'd use something like AWS Batch to launch jobs independently and scale up as needed

2. GRIB Parsing Optimization (Bug fix) The current approach opens each GRIB file once per variable. Instead, we can open the file once and extract all variables together. Fixing this should noticeably improve performance during parsing.

3. Database Design Improvements As the dataset grows, writing and querying efficiently becomes more important. A few things I'd consider:

- Switching to a more production-ready database like PostgreSQL for better concurrency and long-term durability
- Partitioning the table by something like run date to manage growing data more easily
- Adding indexes based on expected query patterns, like `valid_time_utc`, `variable`, or location fields

4. Parallel Execution Strategy To scale ingestion further, we can break up the work into chunks and run them in parallel. A simple way is to split by date and run multiple jobs in parallel, either on a single machine or distributed in the cloud.

Other Considerations

- It's helpful to be able to track progress during large backfills so jobs can resume if interrupted. Build in retry logic in case issue happens.
- Adding some basic validation on parsed values can help catch data issues early.