SIH25040

FloatChat

Description

The ocean is huge and full of data. Every day, thousands of Argo floats (robotic instruments in the sea) measure things like temperature, salinity, oxygen, and other properties of seawater. This data is very valuable for climate studies, weather prediction, fisheries, and ocean research.

The problem is that this data is stored in NetCDF format, which is complicated and not user-friendly. Only people with strong technical and scientific skills can access and analyze it.

With modern AI and Large Language Models (LLMs), we can make this much easier. Instead of learning coding or data science, a person should be able to just ask a question in plain language, and the system should give them an answer with visualizations.

Example Queries the System Should Handle

- "Show me salinity profiles near the equator in March 2023."
- "Compare BGC (bio-geochemical) parameters in the Arabian Sea for the last 6 months."
- "What are the nearest Argo floats to this location (lat/lon)?"

Why this system is important

- Makes complex ocean data accessible to non-technical people.
- Saves time and effort for scientists and policymakers.
- Helps in decision-making for fisheries, climate monitoring, and research.
- Can be extended in the future to include gliders, buoys, and even satellite datasets.

In short:
This project, FloatChat, is about building an AI-powered conversational tool that turns the difficult task of exploring oceanographic data into a simple chat and visualization experience.

Problem Statement

The Argo program places thousands of autonomous floats across the world's oceans. These floats regularly dive down into the water, measure things like temperature, salinity, oxygen, and other bio-geochemical parameters, and then send this data back. The data is stored in NetCDF files – a scientific data format that is not easy for non-experts to use.

Currently, if someone wants to work with Argo data, they need to:

- Understand oceanography,
- Know how to program in Python/R/Matlab,
- Be comfortable with NetCDF, SQL, and plotting tools.

This makes it hard for policymakers, students, and even some scientists to quickly explore and understand the data.

With the rise of AI and Large Language Models (LLMs), we now have an opportunity to make this data more accessible. Instead of writing code, a user should be able to ask a question in plain English like:

- "Show me salinity profiles near the equator in March 2023."
- "Compare oxygen levels in the Arabian Sea for the last 6 months."
- "Which Argo floats are closest to Mumbai?"

The system should then find the right data, query it, and show easy-to-understand visualizations like graphs, maps, or tables.

In short:
We want to build an AI-powered conversational tool (FloatChat) that lets anyone ask questions about the ocean in plain language and immediately get data-driven answers with visualizations from Argo float datasets.

Proposed Solution

We will build FloatChat, an AI-powered conversational system that makes it easy for anyone to explore Argo float ocean data using plain language.

How it works (step by step)

1. Read and organize the data
    - Take Argo NetCDF files (temperature, salinity, oxygen, etc.).
    - Convert them into a structured format like SQL tables and Parquet/CSV files.
2. Store data and metadata
    - Keep all profile summaries in a vector database (FAISS/Chroma) for quick search.
    - Use a relational database (Postgres) to store detailed measurements.
3. AI-powered question answering
    - Use LLMs with RAG (Retrieval-Augmented Generation) to translate user questions into SQL queries.
    - Example: User asks *"Show salinity profiles near the equator in March 2023"* → system creates SQL → fetches relevant data.
4. Chat interface for users
    - Provide a chatbot-style interface where users just type questions in natural language.
    - The system guides them if needed.
5. Interactive visualization
    - Show results in dashboards (maps, plots, tables).
    - Examples:
        - Map of float locations.
        - Depth vs temperature/salinity plots.
        - Profile comparisons over time or region.
6. Export and sharing
    - Allow users to download subsets of data in CSV or NetCDF for their own research.

Key Benefits

- Simple for everyone → no need for coding or technical tools.
- Fast insights → from raw data to visualizations in one step.
- Scalable → can be extended to other ocean data (gliders, buoys, satellites).
- Trustworthy → provenance and QC flags included for scientific reliability.

What the solution will do

1. Read and organize Argo data (from NetCDF files into SQL databases and searchable formats).
2. Use AI (LLMs with RAG) to translate user questions into database queries.
3. Provide a chat-style interface where users can simply type questions.
4. Show results in interactive dashboards with maps and plots.
5. Allow data export so researchers can download subsets as CSV or NetCDF.

Why it matters

- Democratizes ocean data → non-technical people can use it.
- Faster insights for scientists & policymakers → no need to preprocess manually.
- Supports decision-making in areas like climate monitoring, fisheries, shipping, and environmental research.
- Scalable → can be extended to other data (gliders, buoys, satellites).

In short:
The proposed solution is to create an end-to-end AI + dashboard system where users can chat with ocean data and instantly get answers, plots, and exports, without needing expert technical knowledge.

Step 0 — Prepare workspace

Purpose: create a reproducible, isolated development workspace so all team members can run the PoC without environment issues.

Why it matters: consistent environments prevent “works-on-my-machine” problems and make CI/CD and deployment simpler.

Detailed sub-steps

1. Create project folder and initialize git:
2. mkdir floatchat && cd floatchat
3. git init
4. Create and activate a Python virtual environment:
    - Linux / macOS:
    - python3 -m venv .venv
    - source .venv/bin/activate
    - Windows (PowerShell):
    - python -m venv .venv
    - .\.venv\Scripts\Activate.ps1
5. Install base dependencies (PoC-level; you may add extras later):
6. pip install xarray netCDF4 pandas sqlalchemy psycopg2-binary sqlalchemy-utils sentence-transformers faiss-cpu fastapi uvicorn python-dotenv plotly streamlit shapely geopandas sqlparse
    - Save versions:
    - pip freeze > requirements.txt
    - If you prefer conda because of geo-libs, create a conda env instead.
7. Create recommended repository layout:
8. floatchat/
9. ├– data/ # incoming raw files
10. ├– data/normalized/ # CSVs produced by ETL
11. ├– etl/
12. ├– services/
13. ├– webapp/
14. ├– notebooks/
15. ├– infra/

16. ├– tests/
17. ├– README.md

Checks

- python --version and pip list to confirm imports installed.
- git status shows clean repo.

Common pitfalls & fixes

- faiss-cpu installation may fail on some Windows machines — use conda or install faiss-cpu wheel matching your Python version or use a lightweight vector store like Chroma for the PoC.
- Geo packages (geopandas) often require system libs; use conda if you hit install errors.

Deliverable: project repo with .venv, requirements.txt, and folder structure.

Step 1 — Inventory & quick inspection of uploaded files

Purpose: know exactly which raw files you have, their sizes, and a quick variable listing so you can plan ingestion.

Why it matters: ARGO NetCDFs vary; variable names and dimensions differ across versions. Inventory prevents surprises in ETL.

Detailed sub-steps

1. List files in working data directory:
2. ls -lh /mnt/data/argo-profiles-*.nc /mnt/data/argo_data_text-merged.pdf
3. For each NetCDF, print a summary (use Python / xarray):
4. import xarray as xr
5. ds = xr.open_dataset("/mnt/data/argo-profiles-1901740.nc", decode_times=False)
6. print(ds) # prints dims, coords, data variables, attributes
7. ds.close()
8. Record into a small CSV or Markdown the variables and dims found per file:
    - columns: filename, size, dims (names & lengths), sample variables, any QC var names, notes.

Checks

- Confirm which files contain LATITUDE, LONGITUDE, JULD, PRES, TEMP, PSAL or variants.
- Note files that have extra BGC variables (oxygen, nitrate).

Pitfalls & fixes

- Some NetCDF use uppercase names, others lowercase: record all variants. The ETL will canonicalize using a mapping (next step).
- If xarray.open_dataset errors with decode_times=True, try decode_times=False and handle time decoding yourself.

Deliverable: data_inventory.md listing each file and variables.

Step 2 — Open one sample NetCDF and print variables

Purpose: get a deep look at structure of a typical file so you can design mapping and schema.

Why it matters: ETL code must be tailored to actual variable names, array shapes, and dimension ordering.

Detailed sub-steps

1. Open NetCDF in Python and inspect attributes for the time variable and QC variables:
2. import xarray as xr
3. ds = xr.open_dataset("/mnt/data/argo-profiles-1901740.nc", decode_times=False)
4. print("Dimensions:", ds.sizes)
5. for v in ds.variables:
6. print(v, ds[v].dims, ds[v].attrs.get('long_name'), ds[v].attrs.get('units'))
7. ds.close()
8. Look especially for:
    - profile-level coords (N_PROF, N_LEVELS) and their names;
    - time units in JULD: check ds['JULD'].attrs.get('units') — **do not assume epoch**; parse the units attribute to convert to datetimes properly.
9. Determine array shapes: e.g., TEMP may be (N_PROF, N_LEVELS) or flattened differently.

Checks

- Confirm presence of per-profile arrays (2D) vs per-float scalar arrays (1D).
- Confirm presence of *_QC variables (e.g., TEMP_QC, PSAL_QC) — note their names.

Pitfalls & fixes

- Some ARGO NetCDFs store variables as masked arrays or use fill values — check _FillValue and missing_value attributes and handle them in ETL (replace with NaN).
- Time conversion: use netCDF4.num2date(ds['JULD'][:], units=ds['JULD'].units, calendar=ds['JULD'].attrs.get('calendar','gregorian')) to convert JULD to datetimes if decode_times=False.

Deliverable: short note (notebooks/sample_inspect.ipynb or markdown) with outputs and key variable names recorded.

Step 3 — Build variable mapping file

Purpose: create a canonical mapping from variable-name variants in different NetCDF files to consistent internal names.

Why it matters: avoids scattered if var in ds logic; central mapping makes ETL code readable and maintainable.

Detailed sub-steps

1. Create var_map.yaml (example content):
2. latitude: ["LATITUDE","latitude","LAT"]
3. longitude: ["LONGITUDE","longitude","LON"]
4. time: ["JULD","juld","JULD_DECIMAL"]
5. pressure: ["PRES","PRES_ADJUSTED","pressure"]
6. temperature: ["TEMP","TEMP_ADJUSTED","temperature"]
7. salinity: ["PSAL","PSAL_ADJUSTED","PSAL"]
8. temp_qc: ["TEMP_QC","temp_qc"]
9. sal_qc: ["PSAL_QC","psal_qc"]
10. Add code to load the mapping and a helper to pick the first present candidate:
11. import yaml
12. vm = yaml.safe_load(open("var_map.yaml"))
13. def find_var(ds, canonical):
14. for cand in vm[canonical]:
15. if cand in ds.variables:
16. return cand
17. return None

Checks

- Run the helper against each file and ensure each canonical variable resolves in your sample files.

Pitfalls & fixes

- New files may have unexpected names — keep var_map.yaml under version control and update as new files come in.

Deliverable: var_map.yaml and small test script tools/test_varmap.py.

Step 4 — Design relational schema (DDL)

Purpose: design the database schema that can store the required information in normalized form.

Why it matters: schema affects query speed, ease of analysis, and ability to generate SQL from LLMs safely.

Detailed sub-steps

1. Choose DB: Postgres with PostGIS (if you want spatial indexing) is recommended.
2. Create schema.sql. Example (richer than earlier snippet):
3. -- enable PostGIS if spatial queries are required
4. CREATE EXTENSION IF NOT EXISTS postgis;
5.
6. CREATE TABLE floats (
7. float_id TEXT PRIMARY KEY,
8. wmo TEXT,
9. platform_type TEXT,
10. deploy_date TIMESTAMP,
11. owner TEXT,
12. meta JSONB);
13.
14. CREATE TABLE profiles (
15. profile_id BIGSERIAL PRIMARY KEY,
16. float_id TEXT REFERENCES floats(float_id),
17. cycle INTEGER,
18. juld TIMESTAMP,
19. lat DOUBLE PRECISION,
20. lon DOUBLE PRECISION,
21. summary TEXT,
22. qc JSONB,
23. geom geometry(Point,4326));

24. CREATE INDEX idx_profiles_time ON profiles (juld);
25. CREATE INDEX idx_profiles_geom ON profiles USING GIST (geom);
26. CREATE TABLE measurements (
27. meas_id BIGSERIAL PRIMARY KEY,
28. profile_id BIGINT REFERENCES profiles(profile_id),
29. level INTEGER,
30. pres DOUBLE PRECISION,
31. temp DOUBLE PRECISION,
32. sal DOUBLE PRECISION,
33. meas_qc JSONB);

34. CREATE INDEX idx_measurements_profile ON measurements(profile_id);
35. Add DDL for provenance table (maps profile → source file & byte offsets):
36. CREATE TABLE provenance (
37. profile_id BIGINT,
38. source_file TEXT,
39. source_path TEXT,
40. PRIMARY KEY (profile_id, source_file));

Checks

- Run psql -f schema.sql in a sandbox Postgres to confirm no syntax errors.

Pitfalls & fixes

- If using managed Postgres (RDS), enabling PostGIS sometimes requires extra steps — test locally first in Docker.

Deliverable: schema.sql.

Step 5 — Stand up a local Postgres (or Docker) instance

Purpose: provide a live DB to ingest into and query.

Why it matters: you need a DB for the LLM to generate SQL against and for the dashboard to query.

Detailed sub-steps

1. Start Postgres with Docker:
2. docker run --name floatchat-postgres \
3. -e POSTGRES_PASSWORD=pgpass \
4. -e POSTGRES_USER=pguser \
5. -e POSTGRES_DB=floatchat \
6. -p 5432:5432 -d postgres:15
7. Connect and create extensions + apply schema:
8. docker exec -it floatchat-postgres psql -U pguser -d floatchat -c "CREATE EXTENSION postgis;"
9. docker cp schema.sql floatchat-postgres:/schema.sql
10. docker exec -it floatchat-postgres psql -U pguser -d floatchat -f /schema.sql
11. Verify connectivity from your host Python script:
12. from sqlalchemy import create_engine
13. engine = create_engine("postgresql://pguser:pgpass@localhost:5432/floatchat")
14. conn = engine.connect(); print(conn.execute("select 1").fetchall()); conn.close()

Checks

- psql -h localhost -U pguser -d floatchat -c "\dt" shows the created tables.

Pitfalls & fixes

- Port conflicts: ensure 5432 free or map to another host port.
- If Postgres container restarts unexpectedly, check Docker logs docker logs floatchat-postgres.

Deliverable: running database with schema applied.

Step 6 — Implement an ETL script (single-run)

Purpose: parse NetCDF files and output normalized CSVs or directly insert into DB.

Why it matters: this converts raw, complex NetCDF into row-oriented data you can query quickly.

Very detailed sub-steps

1. Create file etl/ingest_netcdf.py.
2. High-level algorithm:
    - Load var_map.yaml.
    - For each NetCDF file:

- Open with xarray.open_dataset(file, decode_times=False, mask_and_scale=False).
- Identify variable names using mapping helper.
- Extract per-profile metadata:
  - lat, lon, juld (convert juld to timestamp; see below).
  - cycle_number (if provided).
- For each profile index i:
  - Assemble profiles row: float_id, cycle, juld_ts, lat, lon.
  - Extract measurement arrays for pres, temp, sal for profile i. They may be 1D (levels) or 2D.
  - Apply _FillValue and QC: mask fill values or extreme sentinel values (e.g., 99999) and mark as NaN.
  - Build measurements rows: one per level with level_index, pres, temp, sal, qc.
- Write profiles and measurements to CSV (or directly to_sql with SQLAlchemy).

3. **Time conversion (JULD) guidance**:
   - Inspect ds['JULD'].attrs.get('units') and calendar.
   - If units looks like "days since 1950-01-01 00:00:00", use:
   - from netCDF4 import num2date
   - units = ds['JULD'].attrs.get('units')
   - calendar = ds['JULD'].attrs.get('calendar', 'gregorian')
   - juld_vals = ds['JULD'][:] # numeric array
   - juld_dates = num2date(juld_vals, units=units, calendar=calendar)
   - If decode_times=True works, you can rely on xarray to give datetime objects.
4. QC flags:
   - If TEMP_QC or PSAL_QC present, carry them into meas_qc.
   - Do not auto-delete flagged points — instead annotate; allow user to decide to filter flagged points.
5. Save results:
   - Write data/normalized/floats.csv, profiles.csv, measurements.csv.
   - Also produce a data/normalized/profile_summaries.csv with summary text used later.

Sample minimal code (core idea):

```
import xarray as xr, pandas as pd, yaml, numpy as np
vm = yaml.safe_load(open("var_map.yaml"))
def find_var(ds, canonical): ...
for fname in list_of_files:
    ds = xr.open_dataset(fname, decode_times=False, mask_and_scale=False)
    latv = find_var(ds,'latitude'); lonv = find_var(ds,'longitude')
    # convert juld
    juldv = find_var(ds,'time')
    juld_vals = ds[juldv].values
    # convert to datetime with netCDF4.num2date as explained
    ...
    # iterate profiles
    for i in range(n_profiles):
        p_lat = float(ds[latv].values[i])
        ...
```

Checks

- After running ETL, count rows:
- SELECT COUNT(*) FROM profiles;
- SELECT COUNT(*) FROM measurements;
- Compare n_profiles detected in NetCDF dims to rows inserted.

Pitfalls & fixes

- Memory: large NetCDFs → process per-file or per-profile streaming rather than loading all into memory. Use chunked reads with xarray if needed.
- Fill values not documented: check variable attrs _FillValue and missing_value.

Deliverable: etl/ingest_netcdf.py and data/normalized/*.csv.

Step 7 — Load CSVs into Postgres (COPY)

Purpose: load the normalized CSVs into the DB quickly and reliably.

Why it matters: bulk COPY is much faster than row-by-row inserts.

Detailed sub-steps

1. If CSVs are on host and DB is in Docker, either copy CSVs into the container or use psql from host to run \copy:
2. psql -h localhost -U pguser -d floatchat -c "\copy floats FROM 'data/normalized/floats.csv' CSV HEADER"
3. psql -h localhost -U pguser -d floatchat -c "\copy profiles FROM 'data/normalized/profiles.csv' CSV HEADER"
4. psql -h localhost -U pguser -d floatchat -c "\copy measurements FROM 'data/normalized/measurements.csv' CSV HEADER"
5. For large imports, disable indexes temporarily and re-enable after load for faster performance.

Checks

- Run counts:
- SELECT COUNT(*) FROM profiles;
- SELECT COUNT(*) FROM measurements WHERE temp IS NULL;
- Spot-check a few rows with SELECT * FROM profiles LIMIT 5;

Pitfalls & fixes

- CSV quoting issues: ensure pandas.to_csv(..., index=False) used consistently.
- Data types mismatch (e.g., strings in numeric columns): inspect CSV and cast/clean before copy.

Deliverable: populated floats, profiles, measurements tables.

Step 8 — Run basic SQL sanity checks

Purpose: ensure data quality and detect anomalies before building indexes and models.

Why it matters: garbage in → garbage out in RAG/LLM outputs.

Detailed checklist & queries

1. Null checks:
2. SELECT COUNT(*) FROM profiles WHERE lat IS NULL OR lon IS NULL;
3. Range checks:
4. SELECT MIN(juld), MAX(juld) FROM profiles;
5. SELECT MIN(temp), MAX(temp) FROM measurements;
6. SELECT COUNT(*) FROM measurements WHERE sal > 40 OR sal < 0;
7. Profile counts vs measurements:

8. SELECT p.profile_id, COUNT(m.*) as meas_count FROM profiles p LEFT JOIN measurements m ON p.profile_id = m.profile_id GROUP BY p.profile_id ORDER BY meas_count DESC LIMIT 10;
9. QC summary:
10. SELECT meas_qc, COUNT(*) FROM measurements GROUP BY meas_qc ORDER BY 2 DESC;

Checks

- No extreme out-of-range values (or if present, annotate in ingest report).
- Measurement counts per profile are reasonable (e.g., typical ARGO 50–200 levels).

Pitfalls & fixes

- Time units interpreted incorrectly → results show dates far in past/future. Re-check JULD conversion.

Deliverable: ingest_report.json or sanity_checks.md with counts & notes.

Step 9 — Create per-profile textual summaries

Purpose: produce short human-readable summaries per profile that will be used for embeddings and provenance.

Why it matters: RAG needs small textual chunks that capture the essence of a profile for semantic retrieval.

Detailed sub-steps

1. For each profile, compute:
    - date (juld), lat/lon, min/max/mean of temp and sal, number of levels, notable QC flags.
2. Template:
3. "Float {float_id} cycle {cycle} at {lat:.3f},{lon:.3f} on {date}: temp {tmin:.2f}-{tmax:.2f}°C (mean {tmean:.2f}), sal {smin:.2f}-{smax:.2f} PSU, levels={n_levels}, qc_summary={qc_string}"
4. Output CSV profile_summaries.csv with columns: profile_id, float_id, juld, lat, lon, summary_text.

Checks

- For a few profiles, read their summaries and confirm they communicate the important bits.

Pitfalls & fixes

- Avoid very long summaries — keep each under ~200 tokens for embedding efficiency.

Deliverable: data/normalized/profile_summaries.csv or DB table profile_summaries.

Step 10 — Build embeddings for summaries (FAISS)

Purpose: create a vector index that allows semantic search over profile summaries.

Why it matters: enables RAG to find semantically relevant profiles even if user phrasing differs.

Detailed sub-steps

1. Choose embedding model (PoC): sentence-transformers/all-MiniLM-L6-v2 (fast & small). For domain specificity, later fine-tune or use larger models.
2. Example code:
3. from sentence_transformers import SentenceTransformer
4. import numpy as np, faiss, json
5.
6. model = SentenceTransformer("all-MiniLM-L6-v2")
7. summaries = [...] # list of summary_texts in same order as profile_ids
8. embeddings = model.encode(summaries, convert_to_numpy=True, show_progress_bar=True)
9. # normalize for cosine
10. faiss.normalize_L2(embeddings)
11. dim = embeddings.shape[1]
12. index = faiss.IndexFlatIP(dim) # inner product on normalized vectors = cosine similarity
13. index.add(embeddings)
14. # persist index
15. faiss.write_index(index, "faiss_index.bin")
16. # save mapping from index position -> profile_id / metadata
17. with open("profile_index_map.json","w") as f:
18. json.dump({"profile_ids": profile_ids, "lat": lats, "lon": lons}, f)
19. Consider using IndexIVFFlat for large datasets (requires training), but Flat index is OK for small PoC.

Checks

- Query a sample embedding and ensure the expected profile_id is among top results.

Pitfalls & fixes

- FAISS requires careful version matching on some OSes; if trouble, use Chroma or local SQLite-based nearest-neighbor as fallback.

Deliverable: faiss_index.bin and profile_index_map.json.

Step 11 — Create a retrieval API (hybrid)

Purpose: retrieve a small candidate set using metadata filters (time/geo) and rank by semantic similarity.

Why it matters: ensures retrieval respects hard constraints (spatial/time) while using semantic match for relevance.

Detailed steps

1. Design function retrieve(query_text, bbox=None, time_window=None, k=10):
    - **Step A — slot extraction**: parse user's natural-language query to extract optional filters:
        - Use a simple rule-based extractor (regex) or a light LLM to get lat_min, lat_max, start_date, end_date, variables.
    - **Step B — metadata filter**: run SQL on profiles table:
    - SELECT profile_id FROM profiles WHERE lat BETWEEN :lat_min AND :lat_max AND juld BETWEEN :start_date AND :end_date LIMIT 1000;

This reduces the candidate set to those satisfying hard constraints.

    - **Step C — semantic re-rank**:
        - Fetch summary_texts for candidate profiles.
        - Generate embedding for query_text.
        - Run FAISS search restricted to candidate vectors (you can either index all and then filter results by allowed profile_ids, or create a sub-index for candidates).
2. Return top-K profiles with profile_id, score, summary_text, lat, lon, juld.

Checks

- Test retrieve("salinity profiles near equator March 2023", bbox=(-2,2,-180,180), time_window=('2023-03-01','2023-03-31')) and inspect top results.

Pitfalls & fixes

- If candidate set is empty after metadata filter, relax spatial/time window or fall back to global semantic search and annotate that filter relaxation happened.

Deliverable: microservice or function retrieve() and an API endpoint /retrieve.

Step 12 — Implement MCP prompt template for SQL generation

Purpose: enforce a structured interface between user queries, retrieval context, and the LLM output so the model returns safe, reproducible SQL and metadata.

Why it matters: reduces hallucination, provides a fixed JSON output the system can parse and validate automatically.

Detailed sub-steps

1. Design a prompt skeleton mcp_template.txt with:
    - System role: model is an assistant that **must** return only JSON that follows the schema.
    - Allowed tables/columns: explicitly list profiles and measurements columns.
    - Few-shot examples: show 4–8 curated pairs of NL → JSON → SQL; keep SQL read-only and simple.
2. Required JSON schema (example):
3. {
4. "intent": "fetch_profiles | compare_parameters | find_nearest",
5. "filters": {"lat_min": null, "lat_max": null, "lon_min": null, "lon_max": null, "start_date": null, "end_date": null, "variables": []},
6. "sql": "SELECT ... ;",
7. "explain": "one-sentence explanation",
8. "visual_hint": "map|profile_plot|table"
9. }
10. Few-shot examples (careful, exact example):
    - NL: “Show me salinity profiles within 2° of the equator during March 2023.”
    - Model JSON+SQL (exact allowed columns):
    - {
    - "intent":"fetch_profiles",
    - "filters":{"lat_min":-2,"lat_max":2,"start_date":"2023-03-01","end_date":"2023-03-31","variables":["salinity"]},
    - "sql":"SELECT p.profile_id, p.float_id, p.juld, p.lat, p.lon, m.level, m.sal FROM profiles p JOIN measurements m ON p.profile_id = m.profile_id WHERE p.lat BETWEEN -2 AND 2 AND p.juld >= '2023-03-01' AND p.juld < '2023-04-01' AND m.sal IS NOT NULL LIMIT 1000;",
    - "explain":"select salinity measurements for profiles near equator during March 2023",
    - "visual_hint":"profile_plot"
    - }
11. Keep the prompt strict: instruct "Return only valid JSON. No commentary."

Checks

- Run a few sample queries through the prompt and inspect the raw model output to ensure it returns well-formed JSON.

Pitfalls & fixes

- LLM may ignore format; enforce strict JSON with a wrapper that extracts first JSON object substring and validates with json.loads. If model returns extra text, strip until the first {– matching } block.

Deliverable: mcp_template.txt and a small script tools/test_mcp.py to validate outputs.

Step 13 — LLM wrapper and validation

Purpose: call model (open or self-hosted), parse its JSON, validate SQL is safe, and execute read-only queries.

Why it matters: prevents the model from issuing unsafe SQL or referencing columns not in schema.

Detailed sub-steps

1. LLM call structure:
    - Inputs: system prompt from mcp_template, retrieved evidence (top-K summaries), user query.
    - Send as a single prompt with a hard cap on evidence tokens.
2. Parse and validate JSON:
3. import json, sqlparse
4. resp_txt = call_llm(prompt_text)
5. # isolate JSON substring and load
6. obj = json.loads(extract_json(resp_txt))
7. sql_str = obj['sql']
8. SQL safety checks:
    - Use sqlparse.parse(sql_str) to get tokens.
    - Enforce that only SELECT is present.
    - Ensure only allowed tables (profiles, measurements) and columns appear (maintain a whitelist).
    - Disallow ; beyond final SELECT, DROP, INSERT, UPDATE etc. Reject if found.
9. Parameterization & execution:
    - Replace string literals in SQL with parameter placeholders if user input contains untrusted strings.

    - Execute the SQL read-only and fetch results with a row limit (e.g., 1000).
10. Compose response: include answer_text (natural language summarization of results), sql, rows (or a pointer to a CSV), and provenance (the top-K retrieved profile summaries and scores that were included in context).

Checks

- Test with adversarial queries (e.g., “delete table ...”) to confirm SQL validation prevents execution.

Pitfalls & fixes

- Complex generated SQL (nested subqueries) may be valid but complex to validate; restrict MCP to require simple plain SELECT with optional JOIN to measurements.

Deliverable: llm_adapter.py (function handle_query(query_text, context)) and unit tests for SQL validation.

Step 14 — Wire /chat endpoint

Purpose: unify retrieval + MCP + SQL execution behind a single API endpoint that the frontend uses.

Why it matters: simplifies frontend and provides a single place for logging/auditing.

Detailed steps

1. Use FastAPI for quick dev:
2. # main.py
3. from fastapi import FastAPI
4. app = FastAPI()

5. @app.post("/chat")
6. async def chat_endpoint(req: dict):
7. query = req["query"]
8. # 1) extract slots (bbox/time)
9. # 2) retrieve candidates
10. # 3) call MCP via llm_adapter
11. # 4) validate SQL and execute
12. # 5) return JSON with answer_text, sql, rows, provenance, visual_hint
13. Support streaming responses: for long LLM responses, use Server-Sent Events (SSE) or WebSockets to stream partial answers to the frontend.

Checks

- curl -X POST -H "Content-Type: application/json" -d '{"query":"salinity equator march 2023"}' http://localhost:8000/chat returns JSON with sql and rows.

Pitfalls & fixes

- LLM latency: make the endpoint asynchronous and push partial updates; show spinner in frontend.

Deliverable: services/api/main.py with /chat implemented and uvicorn run command in README.

Step 15 — Create profile plotting utilities

Purpose: create reusable plotting functions to render depth vs temp/salinity for one or multiple profiles.

Why it matters: consistent visuals are required for the dashboard and for automated attachments in chat responses.

Detailed sub-steps

1. Use Plotly for interactive plots (saves as HTML or PNG):
2. import plotly.graph_objects as go
3. def plot_profile(depth, temp=None, sal=None, title="Profile"):
4. fig = go.Figure()
5. if temp is not None:
6. fig.add_trace(go.Scatter(x=temp, y=depth, mode='lines+markers', name='Temperature (°C)'))
7. if sal is not None:
8. fig.add_trace(go.Scatter(x=sal, y=depth, mode='lines+markers', name='Salinity (PSU)', xaxis='x2'))
9. fig.update_yaxes(autorange='reversed', title_text='Depth/Pressure')
10. fig.update_layout(title=title, xaxis_title='Temperature (°C)', legend=dict(x=0.8, y=0.1))
11. return fig
12. For side-by-side multiple profiles, overlay with transparency or use separate subplots.

Checks

- Test rendering into a PNG for embedding in chat:
- fig = plot_profile(depth, temp, sal)
- fig.write_image("/tmp/profile_plot.png", width=800, height=600)

Pitfalls & fixes

- Depth arrays must be monotonic or cleaned — invert axis for depth semantics.

Deliverable: services/visuals.py with tested functions.

Step 16 — Build minimal Streamlit PoC app

Purpose: provide an interactive UI combining chat, map, and profile viewer for quick demos.

Why it matters: easy-to-run demo for stakeholders and evaluators.

Detailed sub-steps

1. Create webapp/app.py skeleton:
2. import streamlit as st
3. st.set_page_config(layout='wide')
4. st.sidebar.title("FloatChat")
5. query = st.sidebar.text_input("Ask a question")
6. if st.sidebar.button("Run"):
7. res = call_api("/chat", {"query": query})
8. st.write(res["answer_text"])
9. if res["visual_hint"] == "profile_plot":
10. # get data or url and render
11. st.plotly_chart(load_plot_from_res(res))
12. # In main pane: map and list of profiles; clicking sends profile_id to chat
13. Map: use pydeck or streamlit-folium with points from profiles table.
14. Chat area: show conversation history, allow “Ask about this profile” button that pre-fills query with profile_id.

Checks

- streamlit run webapp/app.py opens in browser and the demo calls the local FastAPI /chat and renders result.

Pitfalls & fixes

- CORS: if FastAPI runs on another port, set CORS in FastAPI to allow Streamlit in dev.

Deliverable: webapp/app.py (single-file PoC).

Step 17 — Add CSV / NetCDF slice export

Purpose: let users download raw subsets for reproducibility and deeper offline analysis.

Why it matters: researchers expect to export exact data used to generate plots.

Detailed sub-steps

1. Implement /export endpoint:
    - Accept profile_ids and format=csv|netcdf.
2. For CSV:
    - Query measurements for these profile_ids and return pandas.DataFrame.to_csv() as streaming response.
3. For NetCDF slice:
    - For each profile, reassemble arrays into a small xarray.Dataset and call to_netcdf(path):
    - ds = xr.Dataset({
    - "PRES": (("profile","level"), pres_array),
    - "TEMP": ...
    - }, coords={"profile": profile_ids})
    - ds.to_netcdf("/tmp/slice.nc")
    - Return file as download link.

Checks

- Downloaded file contains the expected number of levels and headers.

Pitfalls & fixes

- NetCDF attributes: preserve _FillValue and variable attributes for fidelity.

Deliverable: /export implemented and tested.

Step 18 — Provenance & QC display

Purpose: make every model-generated answer auditable by listing which profile summaries and source files were used.

Why it matters: domain experts require traceability to trust model outputs.

Detailed sub-steps

1. Extend /chat response to include provenance array:
2. "provenance":[

3. {"profile_id":123,"score":0.92,"summary":"...","source_file":"/s3/argo/argo-profiles-1901740.nc"}
4.]
5. In UI, show provenance panel with:
    - clickable links to source file or to /export?profile_ids=....
    - QC flags displayed for plotted points; highlight flagged points in red or with special markers and include hover text showing QC codes.
6. Keep provenance size small (top-5) but include full metadata for each entry.

Checks

- Click a provenance entry and load raw profile data.

Pitfalls & fixes

- If source files are large and stored in S3, generate presigned URLs or display storage key instead of making the UI fetch the full file automatically.

Deliverable: provenance feature in API + UI.

Step 19 — Unit tests & integration tests

Purpose: ensure each module performs correctly and prevent regressions.

Why it matters: tests speed up development and make demos reliable.

Detailed test suggestions

1. ETL tests:
    - Given a small, known NetCDF fixture, ingest_netcdf.py should produce exactly N profiles and known aggregate values.
2. Retrieval tests:
    - For curated queries, ensure retrieve() returns expected profile_ids in top-3.
3. MCP / SQL tests:
    - For a list of NL inputs, assert that LLM+validation yields a safe SQL containing only whitelisted columns.
4. API tests:
    - Use pytest + httpx to call /chat with mocked LLM responses to validate full pipeline.
5. CI:
    - Add pytest step in GitHub Actions or GitLab CI.

Checks

- Run pytest and get all-green locally.

Pitfalls & fixes

- Flaky tests due to randomness in LLM outputs — mock LLM responses for deterministic tests.

Deliverable: tests/ folder with pytest suites.

Step 20 — Curate PoC demo queries & scripts

Purpose: prepare a short, repeatable demo that demonstrates core features and success criteria.

Why it matters: evaluators need reproducible steps to validate PoC.

Detailed sub-steps

1. Choose 6–10 queries that showcase features:
    - "Show salinity profiles within 2° of the equator in March 2023."
    - "Compare oxygen profiles in Arabian Sea between Jan–Jun 2024."
    - "Nearest ARGO floats to 15.0N, 72.5E."
2. For each query, provide:
    - expected visual_hint (map, profile_plot),
    - curl command calling /chat,
    - reference outputs (CSV, screenshot).
3. Add a demo_playbook.md that lists the steps and expected outcomes.

Checks

- Run each demo curl and confirm UI renders expected charts.

Deliverable: demo_playbook.md.

Step 21 — Logging, metrics, and a small dashboard

Purpose: capture performance and usage metrics for debugging and optimization.

Why it matters: helps spot slow queries, expensive LLM usage, or model drift.

Detailed sub-steps

1. Log per-query:
    - timestamp, user_id (if auth), raw query, sanitized SQL, LLM tokens used, response_time, success/fail, top provenance ids.
    - Write to structured JSON log or use structured logging library.
2. Metrics to capture:
    - requests per minute, average LLM latency, DB query latency, retrieval precision on known tests.

3. Build a minimal Streamlit dashboard to view logs (or use Prometheus + Grafana if you prefer).
    - Show histograms of latencies, counts by intent, top queries.

Checks

- Log volume and verify logs have required fields.

Pitfalls & fixes

- Avoid logging raw user PII. Mask sensitive data.

Deliverable: logs/queries.log and metrics_dashboard.py.

Step 22 — Domain-expert validation

Purpose: have oceanographers verify correctness and usefulness of the outputs.

Why it matters: LLMs may be syntactically correct but scientifically incorrect without domain checks.

Detailed sub-steps

1. Prepare evaluation packet:
    - 10–20 benchmark queries with ground-truth expected results (or example published analyses).
2. Ask domain experts to grade:
    - correctness (0/1), completeness, and whether provenance is sufficient.
3. Measure metrics:
    - SQL accuracy (did model generate a correct SQL?)
    - Retrieval precision@k
    - Human correctness rate
4. Record feedback and prioritize fixes (e.g., better QC handling, more conservative filters).

Checks

- For each benchmark query, confirm pass/fail counts and expert comments recorded.

Deliverable: expert_review.xlsx and update plan.

Step 23 — Hardening & production checklist

Purpose: prepare the system for safe, repeatable production use.

Why it matters: production needs reliability, security, and maintainability.

Checklist (implement these before production):

- Authentication & authorization (OAuth2 / API keys / RBAC).

- Rate limiting on LLM endpoints (protect budget).
- Encrypt data at rest (S3, DB) and in transit (HTTPS).
- Backups for Postgres and FAISS index snapshots.
- Automated ingestion pipeline (cron / Airflow) with idempotency.
- Error handling and alerting (Sentry / PagerDuty).
- CI/CD pipelines with automated tests & image builds.
- Data governance: tracking dataset versions, access logs.

Deliverable: production_checklist.md and implementation plan.

Step 24 — Optional: Add satellite overlays & external datasets

Purpose: enrich context by allowing in-situ (ARGO) vs satellite comparisons (e.g., SST).

Why it matters: many oceanographic analyses combine both data types.

Detailed sub-steps

1. Pick satellite SST dataset (e.g., GHRSST or NOAA OISST).
2. Store as tiles (XYZ) or precomputed raster files. For Streamlit/Leaflet, tile layers are easiest.
3. In the dashboard, allow overlay toggles (SST, chlorophyll).
4. If performing joins, be careful to convert coordinates, interpolation method, and time offsets.

Checks

- Overlay shows expected SST values where ARGO floats are located.

Pitfalls & fixes

- Large rasters: use tiling and caching. Be aware of different spatial resolutions.

Deliverable: demo overlay on map in webapp.

Step 25 — Packaging & deployment

Purpose: containerize services and provide manifests for reproducible deployment.

Why it matters: makes it easy to move PoC to staging or cloud.

Detailed sub-steps

1. Write Dockerfile for each service:
    - Dockerfile.api, Dockerfile.etl, Dockerfile.webapp.
    -

2. Create docker-compose.yml for PoC with Postgres, services, and optional reverse-proxy:
3. services:
4. db:

   image: postgres:15

   environment: ...
5. api:

   build: ./services/api

   ports: ["8000:8000"]
6. web:

   build: ./webapp

   ports: ["8501:8501"]
7. For production, write Kubernetes manifests using Deployments, Services, ConfigMaps, Secrets, and Persistent Volumes for DB.
8. CI/CD: GitHub Actions to build and publish images and run tests.

Checks

- docker-compose up --build brings up the PoC.

Pitfalls & fixes

- Secrets in repo — use .env and avoid checking into Git.

Deliverable: Dockerfile*, docker-compose.yml, k8s/ manifests.

Step 26 — Final deliverables for evaluation

Purpose: package everything needed for reviewers and to hand off to engineering.

Why it matters: clear deliverables make it easier to review and to continue development.

Deliverable checklist

- README.md with run instructions and architecture diagram.
- schema.sql, ingest_netcdf.py, profile_summaries.csv, faiss_index.bin.
- services/ (api, retrieval, llm_adapter) and webapp/app.py.
- demo_playbook.md with curl commands and expected outputs.
- evaluation_report.pdf summarizing accuracy and expert feedback.
- docker-compose.yml and k8s/ manifests for deployment.
- tests/ and CI workflows.