Estimate 25 endpoints across 6+ ponds, with the following breakdown of endpoint counts:

- **Atmospheric**: 12 endpoints (~48%)
- **Oceanic**: 4 endpoints (~16%)
- **Terrestrial**: 2 endpoints (~8%)
- **Spatial**: 2 endpoints (~8%)
- **Multi-Type**: 3 endpoints (~12%)
- **Metadata**: 1 endpoint (~4%)
- **Restricted**: 7 endpoints (~28%, overlapping)

The three most popular or numerous data ponds by endpoint count and prominence are:

1. **Atmospheric** (12 endpoints)
2. **Restricted** (7 endpoints)
3. **Oceanic** (4 endpoints)

Based on external sources, the most used services are:

- **Atmospheric**: NWS API (widely accessed for weather forecasts and observations).
- **Oceanic**: Tides & Currents API (commonly used for real-time oceanic data like tides and water levels).
- **Restricted**: EMWIN (key for emergency weather alerts, with restricted access controls).

These services will be prioritized in the MVP for ingestion, ETL, and API routing.

Technology Stack

- **Backend**: Node.js with Express
- **Data Processing**: Databricks
- **AI Layer**: TensorFlow.js
- **Cloud Infrastructure**: Cloud provider (S3-compatible storage, serverless compute, monitoring)
- **Database/Cache**: Redis
- **Security**: OAuth 2.0
- **Frontend**: React with Tailwind CSS
- **Recommended Web Services**: NWS API (Atmospheric), Tides & Currents API (Oceanic), EMWIN (Restricted)

Web Service Integration

Phase #1: Endpoints to Data Ponds

1. **Set Up Node.js, Databricks, and Cloud Infrastructure** (8 days)
    - **Tasks**: Install Node.js, set up Express, provision Databricks workspace, configure cloud storage for 6+ ponds, initialize Redis, and set up initial access to NWS API, Tides & Currents API, and EMWIN.
    - **Configuration**: Install Node.js, `npm install express redis`, configure Databricks with cloud credentials, set up storage buckets, and define API endpoints for the three web services.
    - **Time**: 8 days.
2. **Implement ETL with Medallion Architecture (Bronze Layer)** (9 days)
    - **Tasks**: Use Databricks to ingest raw data (Bronze) from NWS API, Tides & Currents API, and EMWIN into respective ponds, alongside other endpoints.
    - **Configuration**: Set up Databricks notebooks, define Bronze schemas, connect to storage.
    - **Time**: 9 days.
3. **Create Metadata Catalogs and Silver Layer Transformation** (6 days)

- **Tasks**: Build metadata catalogs in Databricks for NWS API, Tides & Currents API, and EMWIN data, transform Bronze to Silver.
- **Configuration**: Integrate Repository API with Redis cache, map web service metadata.
- **Time**: 6 days.

4. **Establish Access Controls and Gold Layer Enrichment** (7 days)
    - **Tasks**: Implement OAuth 2.0 in Node.js for EMWIN, enrich Silver to Gold in Databricks for all three web services.
    - **Configuration**: Install `npm install passport passport-oauth2`, configure Databricks Gold pipelines.
    - **Time**: 7 days.
5. **Handle Traffic Management** (6 days)
    - **Tasks**: Design ingestion (tens TB/day from NWS API-related data) and inter-pond transfer (1-10 TB/day) with Databricks Delta Lake.
    - **Configuration**: Optimize Databricks with Delta Lake, monitor with Node.js logging.
    - **Time**: 6 days.

Phase #2: Data Ponds to Personas

6. **Develop Federated API with Node.js** (10 days)
    - **Tasks**: Build a unified API integrating with Databricks, with specific endpoints for NWS API, Tides & Currents API, and EMWIN data.
    - **Configuration**: Define API routes (e.g., `/data?service=nws`), connect to Databricks SQL.
    - **Time**: 10 days.
7. **Integrate AI Layer with TensorFlow.js** (12 days)
    - **Tasks**: Develop a BERT-based NLP model to interpret queries for the three web services (e.g., "weather forecast from NWS API").
    - **Configuration**: Install `npm install @tensorflow/tfjs @tensorflow-models`, train with web service data.
    - **Time**: 12 days.
8. **Set Up React Frontend and Test Responses** (9 days)
    - **Tasks**: Create a React app with Tailwind CSS, test queries for NWS API, Tides & Currents API, and EMWIN with personas.
    - **Configuration**: Install `npx create-react-app my-app`, `npm install tailwindcss`, connect to Node.js API.
    - **Time**: 9 days.
9. **Deploy and Monitor with Serverless Compute** (7 days)
    - **Tasks**: Deploy Node.js API and React app on serverless compute, monitor web service traffic.
    - **Configuration**: Package Node.js for serverless, host React on storage with CDN, use monitoring tools.
    - **Time**: 7 days.
10. **Iterate Based on Feedback** (7 days)
    - **Tasks**: Analyze feedback with Databricks analytics, enhance Atmospheric pond (e.g., NWS API) and UI.
    - **Configuration**: Update Databricks pipelines, refine React components.
    - **Time**: 7 days.

Total Estimated Time: 73 days

Node.js Backend Code (server.js)

```
const express = require('express');
const redis = require('redis');
const { DatabricksSQL } = require('databricks-sql');
const passport = require('passport');
const OAuth2Strategy = require('passport-oauth2').Strategy;
const tf = require('@tensorflow/tfjs');

const app = express();
app.use(express.json());

const client = redis.createClient({ url: 'redis://localhost:6379' });
client.on('error', (err) => console.log('Redis Client Error', err));

const dbConfig = {
  host: 'noaa-federated-databricks.cloud.databricks.com',
  path: '/sql/1.0/endpoints/your-endpoint',
  token: process.env.DATABRICKS_PAT
};
const dbConnection = new DatabricksSQL(dbConfig);

passport.use(new OAuth2Strategy({
  authorizationURL: 'https://auth.noaa.gov/oauth2/authorize',
  tokenURL: 'https://auth.noaa.gov/oauth2/token',
  clientID: 'YOUR_CLIENT_ID',
  clientSecret: 'YOUR_CLIENT_SECRET',
  callbackURL: '/auth/callback'
}, (accessToken, refreshToken, profile, done) => done(null, { accessToken })));

app.use(passport.initialize());

// API Endpoint for Web Services
app.get('/data', async (req, res) => {
  const { service, region } = req.query;
  const cacheKey = `data:${service}:${region}`;

  const cached = await client.get(cacheKey);
  if (cached) return res.json(JSON.parse(cached));

  let query;
  switch (service) {
    case 'nws':
      query = `SELECT * FROM gold.atmospheric_aggregated WHERE region = '${region}'`;
      break;
    case 'tides':
      query = `SELECT * FROM gold.oceanic_aggregated WHERE region = '${region}'`;
      break;
    case 'emwin':
      if (!req.user?.accessToken) return res.status(403).send('Unauthorized');
      query = `SELECT * FROM gold.restricted_aggregated WHERE region = '${region}'`;
      break;
    default:
      return res.status(400).send('Invalid service');
  }

  const session = await dbConnection.openSession();
  const result = await session.executeStatement(query);
  const data = await result.fetchAll();
  await client.setEx(cacheKey, 3600, JSON.stringify(data));

  res.json({ data });
});

// AI Integration (Simplified)
app.post('/query', async (req, res) => {
  const { query } = req.body;
  const model = await tf.loadLayersModel('file://path/to/bert/model.json');
  const prediction = model.predict(tf.tensor2d([query.split(' ')]));
  const intent = prediction.argMax(-1).dataSync()[0];
  res.json({ intent, query });
});

// Start Server
app.listen(3000, () => console.log('Server running on port 3000'));
```

React Frontend (App.js)

```
import React, { useState, useEffect } from 'react';
import './App.css';

function App() {
  const [data, setData] = useState(null);
  const [query, setQuery] = useState('');

  useEffect(() => {
    fetch('http://localhost:3000/data?service=nws&region=CA')
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  const handleQuery = async () => {
    const response = await fetch('http://localhost:3000/query', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ query }),
    });
    const result = await response.json();
    setData(result);
  };

  return (
    <div className="container mx-auto p-4">
      <h1 className="text-2xl font-bold mb-4">NOAA Data Pond Dashboard</h1>
      <input
        className="border p-2 mr-2"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Enter query (e.g., weather forecast in California)"
      />
      <button
        className="bg-blue-500 text-white p-2 rounded"
        onClick={handleQuery}
      >
        Search
      </button>
      <pre className="mt-4">{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default App;
```

CSS (App.css with Tailwind)

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Databricks Configuration

Workspace Setup

- **Cluster Configuration**:
  - **Type**: All-Purpose Cluster
  - **Instance Type**: General-purpose (e.g., Standard_DS3_v2) with 4 workers, scalable to 10 for high traffic.
  - **Runtime**: Latest Databricks Runtime (e.g., 11.3 LTS) with Spark 3.3.
  - **Auto Scaling**: Enable to handle ~tens TB/day ingress.
  - **Initialization Script**: Install required libraries (`databricks-sql-connector`, `pyspark`, `delta`).
- **Storage Configuration**:
  - **Mount Point**: Mount cloud storage (e.g., `dbfs:/mnt/noaa-ponds`) using a service principal or access key.
  - **Pond Structure**: Create directories for each pond (e.g., `dbfs:/mnt/noaa-ponds/atmospheric`, `dbfs:/mnt/noaa-ponds/oceanic`).
  - **Delta Lake**: Enable Delta Lake for all ponds to manage versioning and optimize queries.
- **Security Configuration**:
  - **Access Control**: Use Unity Catalog or table ACLs to restrict access to Restricted Pond data.
  - **Authentication**: Configure Personal Access Token (PAT) or OAuth 2.0 for API integration with Node.js.
  - **Network**: Set up VPC peering or private endpoints for secure data transfer.

Medallion Architecture Implementation

- **Bronze Layer**:
    - **Schema**: Raw ingestion of endpoint data (JSON, NetCDF, GeoJSON) into partitioned tables (e.g., `bronze.atmospheric_raw`).
    - **Configuration**: Use Databricks Autoloader to stream data from storage into Bronze tables.
    - `python`

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("BronzeIngestion").getOrCreate()
raw_df = spark.readStream.format("cloudFiles") \
    .option("cloudFiles.format", "json") \
    .load("dbfs:/mnt/noaa-ponds/atmospheric/raw")
raw_df.writeStream \
    .option("checkpointLocation", "dbfs:/mnt/checkpoints/bronze") \
    .partitionBy("date") \
```

    -     .table("bronze.atmospheric_raw")
- **Silver Layer**:
    - **Schema**: Cleaned and normalized data (e.g., Parquet format) with consistent schemas.
    - **Configuration**: Use Spark SQL to transform Bronze data.
    - `python`

```
%sql
CREATE TABLE silver.atmospheric_cleaned
AS SELECT explode(data) as record
FROM bronze.atmospheric_raw
```

    - WHERE record IS NOT NULL
- **Gold Layer**:
    - **Schema**: Enriched, aggregated data for API consumption.
    - **Configuration**: Aggregate and enrich with metadata.
    - `python`

```
%sql
CREATE TABLE gold.atmospheric_aggregated
AS SELECT region, AVG(value) as avg_value, metadata
FROM silver.atmospheric_cleaned
GROUP BY region, metadata
```

Integration with Node.js

- **Databricks SQL Connector**:
    - **Installation**: Add `databricks-sql-connector` to Node.js (`npm install databricks-sql`).
    - **Configuration**: Set up connection in Node.js.
    - `javascript`

```
const { DatabricksSQL } = require('databricks-sql');

const dbConfig = {
  host: 'noaa-federated-databricks.cloud.databricks.com',
  path: '/sql/1.0/endpoints/your-endpoint',
  token: process.env.DATABRICKS_PAT
};

async function queryDatabricks(sqlQuery) {
  const connection = new DatabricksSQL(dbConfig);
  const session = await connection.openSession();
  const result = await session.executeStatement(sqlQuery);
  const data = await result.fetchAll();
  await session.close();
  return data;
```

    - }
    - **Usage**: Call from the Node.js API endpoint to fetch Gold layer data.
- **Job Scheduling**:
    - **ETL Jobs**: Schedule Bronze-to-Silver and Silver-to-Gold transformations using Databricks Jobs API.
    - **Configuration**: Define a job with a Spark task pointing to the ETL notebooks.

Traffic and Performance

- **Delta Lake Optimization**:
    - Enable Z-order indexing on frequent query columns (e.g., `region`, `date`).
    - Use `OPTIMIZE` and `VACUUM` commands to manage data files.
    - `python`

```
%sql
OPTIMIZE gold.atmospheric_aggregated
```

    - ZORDER BY (region)
- **Monitoring**:
    - Use Databricks System Tables or custom logging to track ingress (~tens TB/day) and egress (10-100 GB).

Notes

- **Databricks Configuration**: Tailored for medallion architecture, scalable traffic, and Node.js integration. Replace placeholders with actual values.
- **Deployment**: Use serverless compute for Node.js, storage/CDN for React, and Databricks jobs for ETL.