Technical Questions - Answers

1.Detecting unusual spikes in usage

To detect unusual spikes in usage, I'd rely on a combination of simple statistical methods that work well with time-series data.

I'd start by using a 7-day moving average to create a stable baseline that smooths out normal daily fluctuations. From there, I'd use standard deviation to identify true anomalies, for example, flagging any day where usage sits more than two standard deviations above the mean.

To catch sharper, sudden jumps, I'd also look at the percentage change from one day to the next. If a day is, say, 50% higher than the previous day, that's worth flagging even if it hasn't yet broken the longer-term statistical pattern. This helps detect equipment issues or unexpected load spikes early.

For a more robust measure that isn't thrown off by extreme outliers, I'd also consider the Interquartile Range (IQR) method, marking anything above Q3 + 1.5×IQR as unusual.

By combining these three approaches, I can flag both gradual abnormal increases and sudden unexpected spikes, giving a more complete and reliable anomaly-detection strategy.

2. Managing application state in the frontend

If I were using Vue (which I'm comfortable with), I'd manage state using **Pinia** because it's lightweight, easy to reason about, and integrates cleanly with the Composition API.

My approach would be:

- Store token, processed energy data, weather data, and chart state in the Pinia store
- Fetch raw data once, store it centrally, and let components subscribe reactively
- Use getters to derive computed values (totals, averages, spikes, comparisons)
- Persist only what's necessary (like the token) either in memory or secure storage

This helps avoid prop drilling and keeps the app predictable, especially with multiple components using the same data.

3. Exploring whether temperature affects energy usage

I would start by aligning both datasets by **date**, since both energy and weather data are time-based.

Then I'd calculate two series:

- temperature values per day
- energy usage (kWh difference between min and max) per day

To explore the relationship, I would:

- plot them together on a combined chart (for example, line chart with two axes)
- compute the **correlation coefficient** between temperature and usage
- look for patterns such as high usage on very hot or very cold days

Even without full statistical analysis, a visual overlay often reveals whether temperature trends match energy patterns. If needed, I could go deeper using a simple linear regression later.

4. Adding a machine learning model in the future

If we were to integrate a machine learning model, a suitable starting point would be a **regression model** that predicts future energy usage based on historical energy readings, weather conditions, seasonality, and time of day.

A reasonable choice could be:

- **Linear Regression** for a simple baseline
- **Random Forest Regressor** for better performance on nonlinear patterns
- Or even a **small LSTM** if we wanted a time-series model later

To integrate it:

- The model would run on the backend as an endpoint (e.g., `/api/predict-usage`)
- The frontend would call the endpoint and display predictions in charts
- The dashboard could show predicted vs actual usage or alert users about expected spikes

I would start with a simple model first so that the infrastructure remains manageable.

5. Refactoring to support 500+ sites and live updates every 15 minutes

Scaling to hundreds of sites requires restructuring both the backend and frontend.

On the backend, I would:

- introduce a **centralised data aggregation service** to fetch and store data every 15 minutes
- cache processed results so that the frontend doesn't hit the API directly
- use async job queues or scheduled workers (e.g., cron jobs)
- store results in a fast datastore such as Redis or a time-series database

On the frontend, I would:

- load only the site the user is viewing rather than all sites
- use pagination or lazy loading if multiple sites need to be compared
- update charts in real time using WebSockets or polling at a safe interval

This structure keeps the app responsive, reduces repeated processing, and ensures the system can scale without increasing load per user.

6. My approach to understanding and solving this task

My Approach: Understanding and Solving the Task

1. Understanding the Problem

Goal: Display energy usage for a device over a date range with weather data

Requirements: Authentication → Data fetching → Processing → Visualization

2. Breaking it Down into Components

Authentication
Get identity token

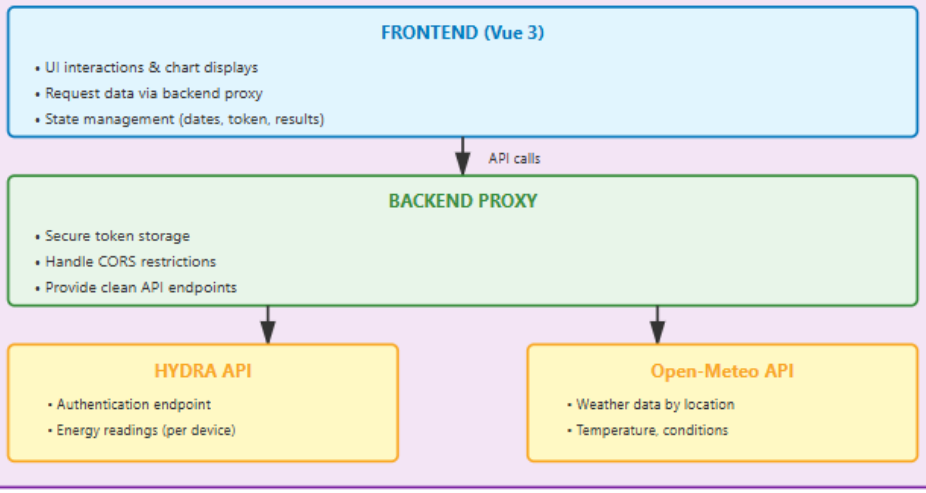
Data Fetching
Energy + Weather APIs

Data Processing
Convert to daily kWh

Frontend UI
Charts & summaries

State Handling
Token, data, dates

3. Layer Responsibilities & Data Flow



4. Trade-offs & Assumptions

- **Daily granularity:** Assumed daily data sufficient (no minute-level needed)
- **Lightweight proxy:** Simple proxy instead of full backend service
- **Chart clarity:** Built for readability rather than complexity
- **Fallback handling:** Weather errors fall back to mock values
- **CORS solution:** Backend proxy handles cross-origin issues
- **Security:** Tokens never exposed to frontend

This approach keeps the solution structured, scalable, and easy to follow