# Building a Grid-Aware Energy Autonomy Agent with n8n

## Overview

This project creates an **AI-powered energy autonomy agent** using [n8n](https://n8n.io) – a free, self-hostable workflow automation tool. The agent monitors open energy data (electricity prices, grid load, weather forecasts) and suggests when a household or microgrid should use, store, or share power. We’ll start with basic data collection and gradually add intelligence and outputs, so even if you’re short on time you can implement the fundamentals first and enhance later. Importantly, all data sources and tools will be **free or open-source**, and the setup targets Europe in general (not just one country).

**Concept Recap:** The agent will fetch live electricity price and demand data (e.g. from ENTSO-E’s transparency platform), get local weather and solar irradiance forecasts (from Open-Meteo), use an AI model to predict next-hour grid “stress” and cost, then apply decision logic to suggest an optimal schedule for battery usage or grid consumption. Results can be logged to an Airtable base (for quick setup) and visualized on a Vue.js dashboard. We’ll also show how to send email notifications so you can receive the AI’s suggestions in your inbox.

## Prerequisites and Setup

Before building the workflow, make sure you have:

* **n8n installed or hosted**: The agent will run on n8n. You mentioned using Render to self-host n8n for free, which is perfect. Ensure your n8n instance is up and accessible.
* **ENTSO-E API access**: Register on the [ENTSO-E Transparency Portal](https://transparency.entsoe.eu/) for a free API key (this may require creating an account and generating an API token). ENTSO-E publishes real-time and historical power system data (load, generation, day-ahead prices, etc.) covering most European countries[[1]](https://www.gridx.ai/blog/resources-for-energy-data#:~:text=%EF%B8%8F%20Coverage%3A%20Most%20European%20countries,time%20Cost%3A%20Free%20Link%3A%20here). This will provide our electricity price and demand inputs. *(If you cannot get an API key quickly, you could start by hard-coding some sample data or using another open data source, but ENTSO-E is ideal for live data.)*
* **Open-Meteo API**: No sign-up required. Open-Meteo is a free, open-source weather API that provides forecast data without any API key for non-commercial use[[2]](https://open-meteo.com/#:~:text=Free%20Weather%20API). We’ll use it to get weather and solar irradiance forecasts. Just have the latitude/longitude of your location or region (for example, use your city coordinates).
* **Google Cloud account (optional, for AI)**: We’ll use an AI model (Google’s Gemini 2.5 **Flash-Lite** on Vertex AI) to predict grid stress. If you have a Google Cloud account, you can access Vertex AI’s API. Google’s Gemini 2.5 Flash-Lite model is very budget-friendly (only about $0.10 per million input tokens and $0.40 per million output tokens[[3]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=Today%2C%20we%E2%80%99re%20releasing%20the%20stable,ready%20for%20scaled%20production%20use)), effectively free for our small-scale usage. Make sure to enable the Vertex AI API and obtain credentials (or use Google’s **Vertex AI** n8n node if available). *Alternatively*, if you prefer not to use Google, you could use an OpenAI API (with their free trial or a local open-source model), but here we’ll assume Vertex AI since you specifically noted its generous free usage.
* **Airtable account**: Sign up for a free Airtable account to log results. Airtable provides an easy online database (spreadsheet-like) where n8n can insert records. It’s quick to set up and fairly customizable – you can define your own fields and retrieve data via Airtable’s API later. We’ll create a base and table to store the agent’s recommendations.
* **Email service (optional)**: If you want email alerts, have an email SMTP service or a Gmail account ready. n8n’s **Send Email** node (or Gmail node) can send emails via SMTP servers[[4]](https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.sendemail/#:~:text=The%20Send%20Email%20node%20sends,using%20an%20SMTP%20email%20server). For example, you might use Gmail’s SMTP or a service like SendGrid; ensure you have credentials (username, password or API key) for the email node.
* **Vue.js Dashboard Template (optional)**: To visualize data on a dashboard quickly, download a free Vue.js admin template. There are many open-source Vue dashboards that can accelerate development[[5]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=Moreover%2C%20free%20and%20open,effective%20solutions) – for example, **CoreUI Free** (Bootstrap-Vue based) or **TailAdmin Vue** (Tailwind-based) are popular choices[[6]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=,D2%20Admin%20for%20advanced%20visualizations). Having the template files ready will save time. You’ll need a development environment to run the Vue app (Node.js and npm installed to run the template’s build). This step is optional if you’re extremely time-crunched, but it’s there if you can allocate some time for front-end visualization.

With the prerequisites in place, we can start building the workflow step by step.

## Step 1: Fetch Live Electricity Price and Demand Data

**Goal:** Get the current (or day-ahead) electricity price and current grid load demand for your region. We’ll use ENTSO-E’s data for Europe.

* **Create an HTTP Request node in n8n:** In your new n8n workflow, add an **HTTP Request** node. This node will call the ENTSO-E API endpoint. Configure it as follows:
* Method: **GET**
* URL: Use the appropriate ENTSO-E API URL for the data you want. For example, to get day-ahead market prices you can use an endpoint like:
* https://api.transparency.entsoe.eu/api?documentType=A44&in\_Domain=10Y1001A1001A83F&out\_Domain=10Y1001A1001A83F&periodStart=20251014&periodEnd=20251015&securityToken=YOUR\_API\_TOKEN
* This is an example for the **Day-Ahead Prices** (documentType A44) for the ENTSO-E “10Y1001A1001A83F” zone (which is a code for a bidding zone, e.g. France). You would replace the domain codes and date parameters as needed (ENTSO-E uses codes for each country/zone and time in specific formats). **Tip:** Start with a simpler query if possible, like total load. For instance, to get **Actual Total Load** for a country, use documentType=A65. Refer to ENTSO-E’s API documentation for the exact parameters.
* **Auth**: Set up your authentication. ENTSO-E uses an API key (securityToken). You can append it in the URL as shown or use n8n’s authentication fields. Since we have the token, appending in URL is straightforward.
* **Response format**: Expect XML (ENTSO-E API returns XML by default). Enable the option in n8n to parse XML to JSON (there’s an option **“Automatically Convert XML to JSON”** in the HTTP node). This will make it easier to work with the data.
* **Test the node:** Execute the HTTP Request node (you can use n8n’s manual run for just this node). If configured correctly, it should return data. For example, for day-ahead prices, you’ll get a JSON object with hourly prices. For load data, you get timestamps and load values. If the data is nested, you might need to inspect the JSON structure to locate the values.
* *If you run into issues*: Double-check your URL parameters and token. If you cannot get the API working quickly, as a basic workaround you could use a static JSON or CSV of prices (perhaps from a recent day) just to have something to feed the next steps. But ideally, get live data. ENTSO-E provides free access to a wealth of grid data across Europe[[1]](https://www.gridx.ai/blog/resources-for-energy-data#:~:text=%EF%B8%8F%20Coverage%3A%20Most%20European%20countries,time%20Cost%3A%20Free%20Link%3A%20here), including **load (demand)** and **day-ahead prices**, so it’s worth the effort.
* **Parse or extract needed fields:** After the HTTP node, add an **Item Lists** node or a small **Function** node to extract the specific values we need:
* For example, from the price data, you might extract the *current hour’s price* (or a list of upcoming hourly prices).
* From the load data, extract the *current load* or *recent load trend*. If using Actual Total Load, you’ll get values for each timestamp – you might take the latest value.
* Store these in workflow variables or JSON fields (e.g. currentPrice, currentLoad, or even an array of next 24h prices if you pulled day-ahead data).
* **(Optional) Normalize units**: Ensure the price is in a uniform unit (EUR/MWh, for instance) and load in MW. ENTSO-E data is usually in MWh or MW and should be fine. Just note what you have (e.g., if day-ahead price is 100 EUR/MWh, that’s €0.1/kWh).

*By completing this step:* you have real energy data ready. For example, “The current wholesale electricity price is X, and current grid demand is Y.” This will feed our AI and decision logic. Even at this basic stage, you could log or output these values to see the live status.

## Step 2: Fetch Weather and Solar Forecast Data

**Goal:** Get weather conditions, especially solar irradiance, since this affects local generation/usage decisions. We use Open-Meteo’s free API.

* **Add another HTTP Request node**: Configure this one to call Open-Meteo’s forecast API. Open-Meteo’s base endpoint is https://api.open-meteo.com/v1/forecast. You need to provide:
* **Latitude & Longitude**: for your location or region of interest.
* **Parameters**: which weather variables you want. We at least need solar irradiance or cloud cover to estimate solar power. Open-Meteo offers *shortwave\_radiation* (global horizontal irradiance) and others. We can also fetch temperature or wind if relevant (for example, high temperature might correlate with high AC usage).
* **Example URL:**
* https://api.open-meteo.com/v1/forecast?latitude=48.85&longitude=2.35&hourly=temperature\_2m,shortwave\_radiation,cloudcover&current\_weather=true
* This would get hourly forecast for temperature, solar radiation, and cloud cover for Paris coordinates, plus current weather. Adjust coordinates and parameters as needed. The API returns JSON by default.
* **No API key needed:** Open-Meteo is open and doesn’t require an API key or auth for standard usage[[2]](https://open-meteo.com/#:~:text=Free%20Weather%20API). Just ensure your query stays within free usage (their free tier allows **10,000 calls/day** with no account, which is plenty).
* **Execute and parse:** Run this node. You should get a JSON with fields like hourly.time, hourly.shortwave\_radiation, etc. Confirm you receive data for the next hours (by default it gives 7 days of hourly data).
* **Extract useful info:** Decide what pieces you need for the AI and logic:
* For example, get the *forecasted solar irradiance for the next hour* (to know if solar generation will increase or decrease).
* Perhaps get a *summary of the day* (like peak sun hours, or if a storm is coming – but to keep it simple, maybe stick to numeric values for the next few hours).
* You could take the next 3–5 hours of solar data as an array, or compute an average. If using an AI, you might even feed it the sequence (e.g., “next hour irradiance = 200 W/m², following hour = 50 W/m² (clouds coming)”). The richer the context, the better the AI’s prediction can be.
* **Combine weather with energy data:** You can use an **IF node or Merge node** in n8n to combine the outputs of the ENTSO-E node and the Open-Meteo node into one consolidated item (merging on the current timestamp for example, or simply use a Function node to construct an object containing { price, demand, solarRadiation, temperature, ... }).

At this point, you have **real-time price, demand, and weather metrics**. This forms the **context** for the AI to analyze grid conditions. It’s worth logging these raw inputs (maybe output to console or temporary Airtable row) to verify they make sense (e.g., price not null, solar radiation values present, etc.).

## Step 3: Predict Grid Stress and Cost with AI

**Goal:** Use an AI model to analyze the data and predict the next hour’s grid stress level and cost trend. This essentially means having the AI interpret whether the grid will be under high load or high prices in the near future, and possibly *why*, given the data. We’ll use the **Google Vertex AI (Gemini 2.5 Flash-Lite)** model via n8n.

* **Add an AI node (or HTTP call):** n8n may have a built-in node for **Google Vertex Chat** models (check in the nodes panel – if not, we can call the REST API). Assuming you have credentials set up:
* If using the **Vertex AI node**: Configure your Google Cloud credentials (likely need a Service Account JSON or OAuth – the n8n docs describe how to set up Google OAuth2 credentials for API calls[[7]](https://community.n8n.io/t/how-to-connect-vertex-ai-with-http-node/48830#:~:text=Solution)). Select the **Gemini 2.5 Flash-Lite** model or specify the model name if required (projects/.../locations/.../publishers/google/models/text-bison for text, or a specific name for Gemini if available). Use the “Chat” or “Text” generation operation.
* If using **HTTP Request node**: Set the method to POST and URL to Google’s Vertex AI endpoint for text generation. For example:  
  https://us-central1-aiplatform.googleapis.com/v1/projects/PROJECT/locations/us-central1/publishers/google/models/gemini-2.5-flash-lite:predict (note: replace PROJECT and region accordingly). The body will contain the prompt and parameters. You’ll also need to include the auth token in headers (which the OAuth2 credential can handle).
* **Prompt engineering:** Craft the prompt to send to the AI. We’ll give it the data we gathered and ask for a prediction:
* For example:
* You are an AI energy analyst.   
  Current wholesale price: €{price} per MWh.  
  Current grid demand: {demand} MW (relative to typical peak).  
  Solar forecast next hour: {irradiance} W/m².  
  Weather: {temperature}°C, {cloudcover}% cloud.  
  Question: Will the grid be under high stress in the next hour? Is the electricity cost likely to be high or low? Please respond with a brief analysis and a classification of grid stress (Low/Medium/High) and cost (Rising/Falling/Stable).
* This prompt provides the model with structured facts and asks for a concise output. Adjust it to your liking – the key is to get a *useful but short prediction* that we can parse. For instance, the model might answer: *“Grid stress will likely be* *High* *due to high demand approaching peak hours, and cost is expected to remain* *High/Rising* *in the next hour.”*
* **Tip:** Keep the output format predictable if possible. You might ask the model to format the result in JSON (e.g., {"stress":"High","cost":"High"}) or a simple phrase containing keywords. Simpler output is easier for n8n to parse with a subsequent node. However, Gemini Flash-Lite is a large language model; while it’s fast and cheap, it’s primarily for natural text, so guiding it to structured output may require few-shot examples in the prompt.
* **Execute the AI node:** When it runs, it will send the prompt to the model and get a response. Verify the response text. The Vertex Gemini 2.5 Flash-Lite model is optimized for speed and cost-efficiency[[8]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=%2A%20Cost,from%20the%20preview%20launch) while still providing reasoning capabilities, so it should give a decent guess quickly. (For example, if demand is high and solar low, it might predict *high stress*; if demand is moderate and lots of solar is coming online, maybe *low stress*).
* **Parse the AI output:** Use a **Set** or **Function** node to extract the predicted categories from the AI’s response. If you asked for specific words like “High”/“Medium”/“Low,” find those in the text. Alternatively, if you managed to get JSON, parse it directly. The result should be something like:
* predicted\_stress = High
* predicted\_cost\_trend = High (or “Rising”, “Falling” etc. depending on how you asked).
* **Basic sanity check:** You might implement a quick check or fallback in case the AI returns something unexpected or incoherent (since freeform AI can sometimes stray). For instance, if no clear keyword is found, you could default to Medium stress. This ensures the next logic step always has a value to work with.

*Note:* Using an AI like this is a bit of a creative approach – effectively, we’re letting a trained model *infer* grid conditions from data without a formal algorithm. It’s a fun application for an energy autonomy scenario. Because Google’s Gemini 2.5 Flash-Lite is extremely low-cost per call[[3]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=Today%2C%20we%E2%80%99re%20releasing%20the%20stable,ready%20for%20scaled%20production%20use), you can use it generously within the free tier or free trial. If setting up Vertex AI takes too long, you can substitute with OpenAI’s GPT-3.5 (n8n has an OpenAI node) or even skip to Step 4 with a manual rule. But having the AI analysis gives the project a nice “smart agent” touch that a curious electrical engineer might experiment with.

## Step 4: Decision Logic – Simulate Optimal Storage/Usage Schedule

**Goal:** Based on the AI’s prediction (and raw data), decide what the household or microgrid should do in the next period: use power from grid, use battery storage, charge the battery, or even share/sell power. Essentially, this is the “control logic” of your agent. We simulate it in n8n (no actual hardware control, just recommendations).

Since time is short, start with **simple rules** and you can expand to a full schedule optimization if you have time:

* **Define scenario assumptions:** Let’s assume the household has a battery and some solar panels (common in microgrids or homes with PV). For example, a 10 kWh battery with current charge level known (you could maintain a state in n8n or Airtable). Also assume if “share power” is an option, it means sending excess to grid or neighbors when battery is full.
* **Add a Function (Code) node:** Use an n8n **Function** node to write custom JavaScript that decides an action. The input to this node will be the data we have so far: current price, demand, solar forecast, and the AI’s stress/cost prediction. In the code, you can implement something like:
* // Example pseudo-code for decision logic  
  const price = $input.item.price; // current price  
  const demand = $input.item.demand;  
  const solar = $input.item.solarForecast;  
  const stress = $input.item.predicted\_stress; // AI prediction  
  let action = "";  
    
  // Simple rule: if grid stress or price is high, minimize grid usage  
  if (stress === "High" || price > SOME\_THRESHOLD) {  
   if (/\* battery has charge available \*/ true) {  
   action = "Discharge battery to supply home load";  
   } else {  
   action = "Reduce usage if possible (grid is stressed and battery empty)";  
   }  
  } else {  
   // Grid stress low or price cheap  
   if (/\* battery not full \*/ true && solar < SOME\_SMALL\_VALUE) {  
   action = "Charge battery from grid (cheap power available)";  
   } else if (solar > SOME\_LARGE\_VALUE) {  
   action = "Use solar to supply loads and charge battery";  
   } else {  
   action = "Use grid power normally (no special action)";  
   }  
  }  
    
  return [{ recommendation: action }];
* This is a simple heuristic. You can refine thresholds (e.g., define what “high price” is – perhaps top 20% of prices, or a fixed value like €150/MWh). Also consider solar: if a lot of solar is expected soon, you might *not* charge the battery from grid now (save room for solar energy later). If solar is currently strong (exceeding load), the decision might be to **share power** (send excess to grid or to a neighbor, which in our simulation we denote as an action).
* **Simulate schedule vs. one-step:** The above logic is essentially deciding for the next hour. If time permits, you could extend this to simulate a schedule for the next several hours. For example, loop through the next 24 hours of forecast (price and solar) and create a plan (like charge battery from hour1-3, discharge at hour4, etc.). However, that gets more complex (essentially solving an optimization problem). For a time-constrained project, focusing on the immediate next action is perfectly fine and demonstrates the concept. You can mention that a full schedule optimization could be done with linear programming or advanced algorithms if this were a production project.
* **Output the decision:** Ensure the Function node outputs the recommendation (e.g., “Discharge battery in next hour to avoid high grid cost”) in a field, e.g., recommendation. You might also output any details used (for example, battery state if you track it, or the predicted values, to log context).

At this stage, our workflow has produced a **recommended action** for the household/microgrid. To recap a possible outcome: The AI predicted “Grid stress High, cost Rising”, and the logic, seeing battery is, say, half full, decides “Discharge battery for the next hour to supply the house, avoid drawing from grid.” Or if the AI predicted low stress and we have an empty battery: “Charge battery from grid while power is cheap.”

These are the kinds of decisions an “energy autonomy agent” would make locally. The system is decentralized (running for your household) but aware of grid conditions (prices, demand) and weather, aligning with the project’s spirit of **distributed intelligent energy management**.

## Step 5: Log Results to Airtable

**Goal:** Store the input data and the AI’s recommendation in Airtable for record-keeping and for use in the dashboard. Airtable will act as a simple database + UI for viewing the agent’s history.

* **Set up Airtable base:** Create a new base (perhaps called "EnergyAutonomy") and within it a table (e.g., "Recommendations"). Define fields to capture data. For instance:
* Date/Time (when the recommendation is made)
* Current Price (number)
* Current Demand (number)
* Solar Forecast (number or text, e.g., “200 W/m²”)
* AI Predicted Stress (single select or text)
* AI Predicted Cost Trend (text)
* Recommended Action (text)

You can customize as needed. Airtable is flexible; just ensure the field names match what you will send from n8n. - **Add Airtable node in n8n:** Use n8n’s **Airtable** integration node (operation: Create Record). Connect it after the decision logic node. Configure: - **Credentials:** Create a new Airtable credential in n8n using your API key or personal access token from Airtable. (Follow the instructions in n8n docs or Airtable node setup – you will need to generate a personal token in Airtable and give it access to the base[[9]](https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/#:~:text=In%20the%20Airtable%20node%20window%2C,configure%20the%20following%20parameters)). - **Base ID** and **Table**: Select the base and table you created. n8n should load your base schema after you authenticate. - **Field Mappings:** Map each field to the corresponding data from previous nodes. For example, Date = an expression like {{$now}} (current timestamp) or the timestamp from the data; Price = the price output from Step 1; Demand = demand from Step 1; Stress = the predicted\_stress from AI; Action = the recommendation text. - **Test the Airtable insertion:** Execute the Airtable node. A new row should appear in your Airtable table with all the data. This confirms n8n can connect to Airtable successfully. n8n’s Airtable node simplifies the process of inserting data[[10]](https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/#:~:text=In%20this%20step%20of%20the,Airtable%20using%20the%20Airtable%20node) – you just point it to your table and provide the values. (Note: If a field name doesn’t match or is missing, the node might throw an error, so double-check the names.) - **Verify customization:** You mentioned concern about Airtable being “not completely uncustomizable”. In practice, Airtable allows a lot of flexibility in how you view and use the data (filters, formulas, even scripts). And because the data is now stored in Airtable, you have an API to retrieve it or feed it to other apps. It’s a quick solution for prototyping. Later, you could move to a more custom database if needed, but for now Airtable gives you a web UI to see all the agent’s suggestions at a glance and is easily connected to n8n data.

Now your workflow, each time it runs, will log an entry. If you schedule this workflow to run every hour (using n8n’s Cron trigger, for example), you’d accumulate a nice timeline of recommendations. That can be valuable for analysis and for powering the dashboard.

## Step 6: Notifications via Email (Optional)

If you want immediate alerts without checking Airtable or the dashboard, set up an email output:

* **Add an Email node (Send Email):** After logging to Airtable, you can branch to a **Send Email** node. Configure it with your SMTP credentials (or use the **Gmail** node if easier – with Gmail OAuth credentials or an app password).
* **Compose the email:** In the node, set the recipient (your email), subject, and body. You can include the variables from the workflow in the body using n8n expressions. For example:
* Subject: Energy Agent Update: {{ $json["predicted\_stress"] }} stress, {{ $json["recommendation"] }}
* Body: A short summary, e.g.:
* Hello,   
  The energy autonomy agent has a new recommendation:  
  - Grid stress next hour: {{$json["predicted\_stress"]}} (cost trend: {{$json["predicted\_cost\_trend"]}}).  
  - Suggested action: {{$json["recommendation"]}}.  
  Data: Price={{$json["price"]}} €/MWh, Demand={{$json["demand"]}} MW, Solar={{$json["solarForecast"]}} W/m².  
    
  - n8n Energy Agent
* This would produce an email like: “Grid stress next hour: High (cost trend: Rising). Suggested action: Discharge battery to supply load. Data: Price=120 €/MWh, Demand=45,000 MW, Solar=50 W/m².”
* **Send and test:** Execute the email node (you may hard-code your email first, to avoid spam). Check your inbox. If it doesn’t arrive, check n8n execution logs for errors (e.g., SMTP misconfiguration). Once working, you can rely on this for real-time notifications.

Email alerts ensure you don’t miss what the agent suggests, even if you’re not actively watching the dashboard. It’s an easy way to get the info “via email” as you requested, and it complements the logging.

## Step 7: Vue Dashboard for Visualization (Optional Advanced)

**Goal:** Create a quick web dashboard to visualize the data and recommendations, using a Vue.js template for speed. This step is if you have extra time and want a nice UI for demonstration.

* **Choose a Vue template:** As mentioned, there are many free Vue.js admin dashboard templates available[[5]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=Moreover%2C%20free%20and%20open,effective%20solutions). For example, **CoreUI Vue** or **Vue Notus** or the **TailAdmin Vue** are good choices. Download or clone the template project. Usually, you will:
* Install dependencies with npm install.
* Run the dev server with npm run serve (or similar) to see it locally.
* **Plan the data flow:** The Vue app needs to get the data (recommendations and possibly historical charts). Easiest path:
* Use Airtable as the data source. Airtable has a REST API that you can call from the frontend (though you might expose your API key – to avoid that, you could create a simple proxy or use only read-only shared views). Another approach: export the data as JSON.
* Alternatively, use n8n to expose an endpoint. For example, n8n could have a separate workflow with a Webhook trigger that queries the Airtable (or stores the latest recommendation in memory) and returns JSON. Then the Vue app could call this n8n webhook URL. This avoids putting API keys in the front-end.
* Given time constraints, a straightforward method: After each run, n8n could write a JSON file or update a small database that the Vue app reads. However, let’s assume using Airtable API directly for simplicity.
* **Implement data fetch in Vue:** In your Vue dashboard code, create a service or component that fetches the Airtable data. You can create an Airtable **API key** (or use the same personal token). Airtable’s API allows you to retrieve all records of a table in JSON (they even provide example code snippets). For example, using fetch in JavaScript:
* async function fetchRecords() {  
   const res = await fetch('https://api.airtable.com/v0/yourBaseId/Recommendations?sort%5B0%5D%5Bfield%5D=Date&sort%5B0%5D%5Bdirection%5D=desc', {  
   headers: { 'Authorization': 'Bearer YOUR\_AIRTABLE\_TOKEN' }  
   });  
   const data = await res.json();  
   return data.records;  
  }
* This would retrieve records from the "Recommendations" table, sorted by Date descending (latest first).
* **Visualize data:** Use Vue components or chart libraries to display:
* **Current status card:** e.g., a card showing the latest recommendation text, perhaps color-coded by stress level (green for low, red for high).
* **Historical chart:** Plot price over time and maybe where the recommendations changed (could use a line chart for price and bars or icons for when battery was used vs grid). Libraries like Chart.js or ApexCharts (some templates include these) can be plugged in. Since the template likely has examples of charts, adapt one of those charts to use your data array from Airtable. For instance, if you have last 24 recommendations, you can make an x-axis of time and lines for price (EUR) and maybe another line for demand, and highlight the high/low stress periods.
* **Weather forecast:** You could also show a small panel of the weather forecast (like “sunny next hour” or display the solar irradiance forecast curve).
* The Vue admin template will handle a lot of styling. You mainly need to inject your content. For speed, you can start by hardcoding a small JSON (one pulled from Airtable) to ensure the charts render, then replace the hardcoded part with actual fetch logic.
* **Plug and play approach:** Since you want plug-and-play, try to modify as little of the template as possible. Identify a page or component in the template that shows dummy data (maybe a dashboard page) and replace the dummy dataset with your real data. For example, if the template has a chart using an array [65,59,80,81,...], swap that with your price array.
* **Hosting the dashboard:** If really short on time, running it locally might suffice for demo. But if you need others to see it, you could quickly deploy it (Render can host static sites, or GitHub Pages for static builds). Many Vue templates can be built into a static bundle (npm run build) that you then serve. Since it’s mostly front-end calling Airtable, deployment is straightforward.

Now you have a **Vue dashboard** that complements the agent: - It shows the latest and historical recommendations. - You can see how often the agent said “use battery” vs “charge battery,” etc., and correlate with price spikes or drops on the chart. - It provides a visual validation that the agent is doing something sensible (or at least something interesting!).

## Wrap-Up and Next Steps

We’ve built a **Grid-Aware Energy Autonomy Agent** step by step: 1. *Data Collection:* pulling live open data for grid status (price & demand) and weather (solar potential). 2. *AI Analysis:* using a free/low-cost AI model (Google’s Gemini 2.5) to interpret and predict near-future grid conditions[[3]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=Today%2C%20we%E2%80%99re%20releasing%20the%20stable,ready%20for%20scaled%20production%20use). 3. *Decision Logic:* implementing simple rules to decide on using or storing energy, aligning with energy efficiency goals. 4. *Logging & Notification:* recording the decisions in Airtable and optionally notifying via email. 5. *Visualization:* optionally, creating a quick Vue.js dashboard from an open-source template to display the data and recommendations, which makes the project feel more complete and user-friendly.

This project fits the profile of an electrical engineer’s curiosity project – it combines energy data, decentralized decision-making, and AI. We used entirely free and open resources: ENTSO-E’s free data access for Europe[[1]](https://www.gridx.ai/blog/resources-for-energy-data#:~:text=%EF%B8%8F%20Coverage%3A%20Most%20European%20countries,time%20Cost%3A%20Free%20Link%3A%20here), Open-Meteo’s no-key weather API[[2]](https://open-meteo.com/#:~:text=Free%20Weather%20API), a cost-efficient AI model, and free tiers of n8n, Airtable, and open-source Vue templates[[5]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=Moreover%2C%20free%20and%20open,effective%20solutions).

**Next improvements (if you find more time or resources):** - **Fine-tune the AI or use a more specialized model:** The current approach treats the LLM as a heuristic predictor. With more time, one could train a simple machine learning model on historical data to predict price spikes or grid stress more rigorously. However, given Gemini Flash-Lite’s speed, you could even prompt it with more data or let it plan a schedule (prompt it to output a plan for the next 6 hours). - **Advanced optimization:** Implement a true optimization for battery scheduling (using linear programming solvers via an n8n code node, or an algorithm like dynamic programming to maximize self-consumption and minimize cost). - **Expand data sources:** Incorporate more data such as grid frequency or carbon intensity (electricityMap API, for instance), to have the agent also optimize for carbon footprint (e.g., use battery when grid is carbon-intense). This could be interesting for a more holistic “green energy” decision maker. - **User interface:** If the Vue dashboard needs to be more interactive, you could add controls (like simulate different battery sizes, or manually override decisions, etc.). Also, ensure mobile responsiveness if you plan to check it on the go – most admin templates are responsive out of the box. - **Europe-wide adaptation:** Since the company is French but we made it European, you can adapt the agent to any country by changing the ENTSO-E zone code and coordinates. You might even run multiple workflows for different regions or households. - **Self-hosting considerations:** Running n8n on Render for free is great for development. If you end up running this agent 24/7 (cron every hour), ensure the instance is stable and consider persistence for any state (like battery charge level, if you simulate that over time – storing that in Airtable or a file might be necessary).

By following these steps, you start with a basic working core (Steps 1–4 can stand alone: data -> decision), and then add logging, alerts, and UI as time allows. Good luck with building your energy autonomy agent, and enjoy the process of seeing AI and automation applied to real-world energy data! Each incremental step will already provide value, and together, they showcase a powerful DIY smart grid assistant running on completely free infrastructure.

**Sources:**

* ENTSO-E Transparency data (open access to European power system data, including load and prices)[[1]](https://www.gridx.ai/blog/resources-for-energy-data#:~:text=%EF%B8%8F%20Coverage%3A%20Most%20European%20countries,time%20Cost%3A%20Free%20Link%3A%20here)
* Open-Meteo free weather API (no API key required for non-commercial use)[[2]](https://open-meteo.com/#:~:text=Free%20Weather%20API)
* Google Vertex AI Gemini 2.5 Flash-Lite (fast, low-cost AI model suitable for such tasks)[[3]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=Today%2C%20we%E2%80%99re%20releasing%20the%20stable,ready%20for%20scaled%20production%20use)
* n8n Airtable integration (easy to insert workflow data into Airtable)[[10]](https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/#:~:text=In%20this%20step%20of%20the,Airtable%20using%20the%20Airtable%20node)
* Open-source Vue dashboard templates (accelerate UI development at no cost)[[5]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=Moreover%2C%20free%20and%20open,effective%20solutions)

[[1]](https://www.gridx.ai/blog/resources-for-energy-data#:~:text=%EF%B8%8F%20Coverage%3A%20Most%20European%20countries,time%20Cost%3A%20Free%20Link%3A%20here) gridX – 14 great energy resources

<https://www.gridx.ai/blog/resources-for-energy-data>

[[2]](https://open-meteo.com/#:~:text=Free%20Weather%20API) ️ Free Open-Source Weather API | Open-Meteo.com

<https://open-meteo.com/>

[[3]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=Today%2C%20we%E2%80%99re%20releasing%20the%20stable,ready%20for%20scaled%20production%20use) [[8]](https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/#:~:text=%2A%20Cost,from%20the%20preview%20launch) Gemini 2.5 Flash-Lite is now stable and generally available - Google Developers Blog

<https://developers.googleblog.com/en/gemini-25-flash-lite-is-now-stable-and-generally-available/>

[[4]](https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.sendemail/#:~:text=The%20Send%20Email%20node%20sends,using%20an%20SMTP%20email%20server) Send Email | n8n Docs

<https://docs.n8n.io/integrations/builtin/core-nodes/n8n-nodes-base.sendemail/>

[[5]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=Moreover%2C%20free%20and%20open,effective%20solutions) [[6]](https://tailadmin.com/blog/free-vue-admin-dashboard#:~:text=,D2%20Admin%20for%20advanced%20visualizations) 25+ Free Vue Admin & Dashboard Template of 2025 | TailAdmin

<https://tailadmin.com/blog/free-vue-admin-dashboard>

[[7]](https://community.n8n.io/t/how-to-connect-vertex-ai-with-http-node/48830#:~:text=Solution) How to connect Vertex AI with HTTP node - Built with n8n - n8n Community

<https://community.n8n.io/t/how-to-connect-vertex-ai-with-http-node/48830>

[[9]](https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/#:~:text=In%20the%20Airtable%20node%20window%2C,configure%20the%20following%20parameters) [[10]](https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/#:~:text=In%20this%20step%20of%20the,Airtable%20using%20the%20Airtable%20node) Inserting data into airtable | n8n Docs

<https://docs.n8n.io/courses/level-one/chapter-5/chapter-5.2/>