**Workflow Name: Daily Ops Coordinator**

**🧠 Goal:**

Prepare daily task lists, optimize routes, and update task boards using local and free tools.

**🎯 Objective**

Automate the process of:

* Ingesting daily delivery data.
* Grouping deliveries by optimal driver allocation.
* Sorting routes for each driver.
* Exporting delivery manifests for execution (Trello/File).

**🛠️ Demo Tools:**

* n8n
* Nominatim (for geocoding)
* Google Sheets
* Trello (optional)

**🧩 PROCESS STAGES (High-Level)**

**1. 📥 Ingest Delivery Data**

* **Source**: Google Sheets (lab) or Postgres (bonus)
* **Fields**: Delivery ID, Name, Address, Window, Priority, Size, etc.
* **Optional Enhancements**:
  + Use a local LLM (Ollama) to validate/flag issues (missing data, ambiguous time windows, duplicate addresses).

**2. 👥 Determine Delivery Capacity**

* Default: use a static config (e.g. num\_drivers = 3).
* **(Optional)** LLM-Driven Insight: Prompt Ollama with the delivery count, delivery zones, constraints → infer estimated number of drivers.
  + "Given 45 delivery locations across these zip codes and time constraints, estimate how many drivers are needed for < 7 hr routes."

**3. 📦 Group Deliveries for Routing**

* **Clustering with K-Means**: Cluster by location (lat/lng) into N groups, where N = driver count
* Optional: weight clustering by size or delivery time window.

**4. 🧭 Sort Each Route (TSP/Nearest Neighbor)**

* + Sort each delivery group to minimize travel distance.
  + Assign optimized route order.
* **LLM Role** (Optional):
  + Provide a summary/alert of possible route issues:
    - "Driver 2 has a 90-minute stop that may exceed expected shift time."

**5. 🧾 Output Final Route Assignments**

* **Trello Cards** – 1 List per Driver with cards for each delivery stop.
* **File Export** – JSON or Markdown saved to disk (free option).
* **Optional Streamlit Dashboard** to visualize routes.

**🔄 Design Principles**

* **Reproducible**: Anyone with n8n can run this.
* **Extendable**: Easily adapted for trash pickup, inspections, or home services.
* **Local-First**: Keeps data on your machine.
* **Free-to-Use**: Zero cost (OpenStreetMap, Trello Free Tier, etc.).

📦 Prerequisites

1. A working n8n environment (Docker or desktop).
2. A base understanding of how to test and trigger n8n workflows.
3. Google authentication credentials for the Sheets node.

📝 Work Instructions

1. ✅ Step 1: Trigger Setup
   1. Replace later with a scheduled trigger.
   2. Use a ‘Manual Trigger’ for testing.

A computer screen shot of a computer mouse

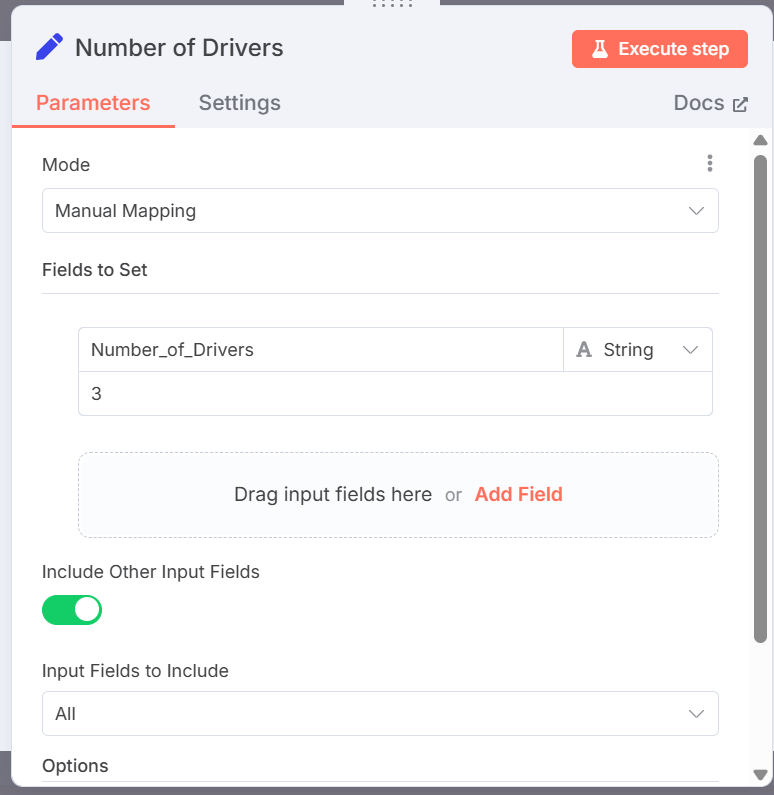
AI-generated content may be incorrect.

1. ✅ Step 2: Connect Google Sheets (Ingest Delivery Data)
   1. Use a “**Google Sheets**” node.
   2. Authenticate with your Google account.
   3. Select:
      1. ‘Document’ from available google sheets (google authentication)
      2. ‘Sheet’ from available sheets within the document
      3. Click ‘Test Step’ to ensure expected outputs.

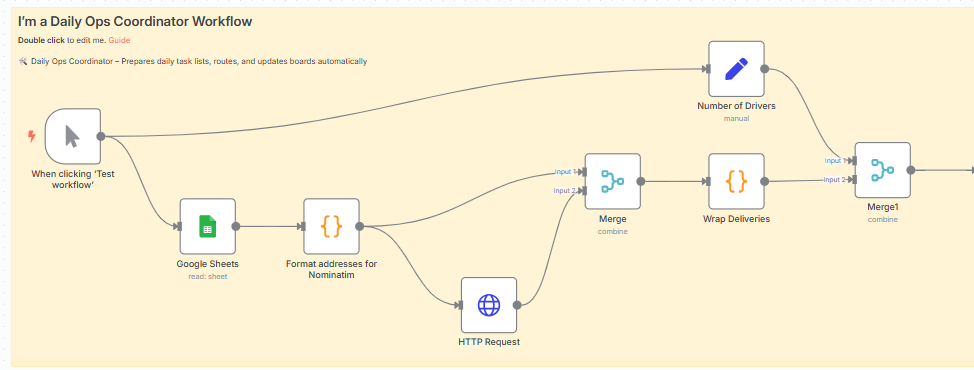
A screenshot of a computer

AI-generated content may be incorrect.

1. ✅ Step 3: Format Addresses
   1. Insert a \*\*Code\*\* node.
   2. Format addresses into a single string.
   3. Encode the address for geocoding via `Nominatim`.
2. ✅ Step 4: Determine Delivery Capacity.
   1. > 🔧 This example uses a static driver count. *You may optionally replace this with an LLM-based prompt to estimate based on time and distance constraints*.
      1. Add a \*\*Set\*\* node named `"Number of Drivers"`
      2. Create a new field:
         1. Name: `Number\_of\_Drivers`
         2. Type: String
         3. Value: `3` (or your desired number of drivers)



* + 1. Connect this node in parallel with the Google Sheets node → merge both streams before clustering.



1. ✅ Step 5: Geocode Delivery Addresses.
   1. Add a ‘Code’ node named "Format addresses for Nominatim".

return items.map(item => {

const { Address, City, ["Zip Code"]: ZipCode } = item.json;

// Correct template literal interpolation

const fullAddress = `${Address}, ${City}, TX ${ZipCode}`;

const encodedAddress = encodeURIComponent(fullAddress);

const nominatimUrl = `https://nominatim.openstreetmap.org/search?q=${encodedAddress}&format=json&addressdetails=0&limit=1`;

return {

json: {

...item.json,

address: fullAddress,

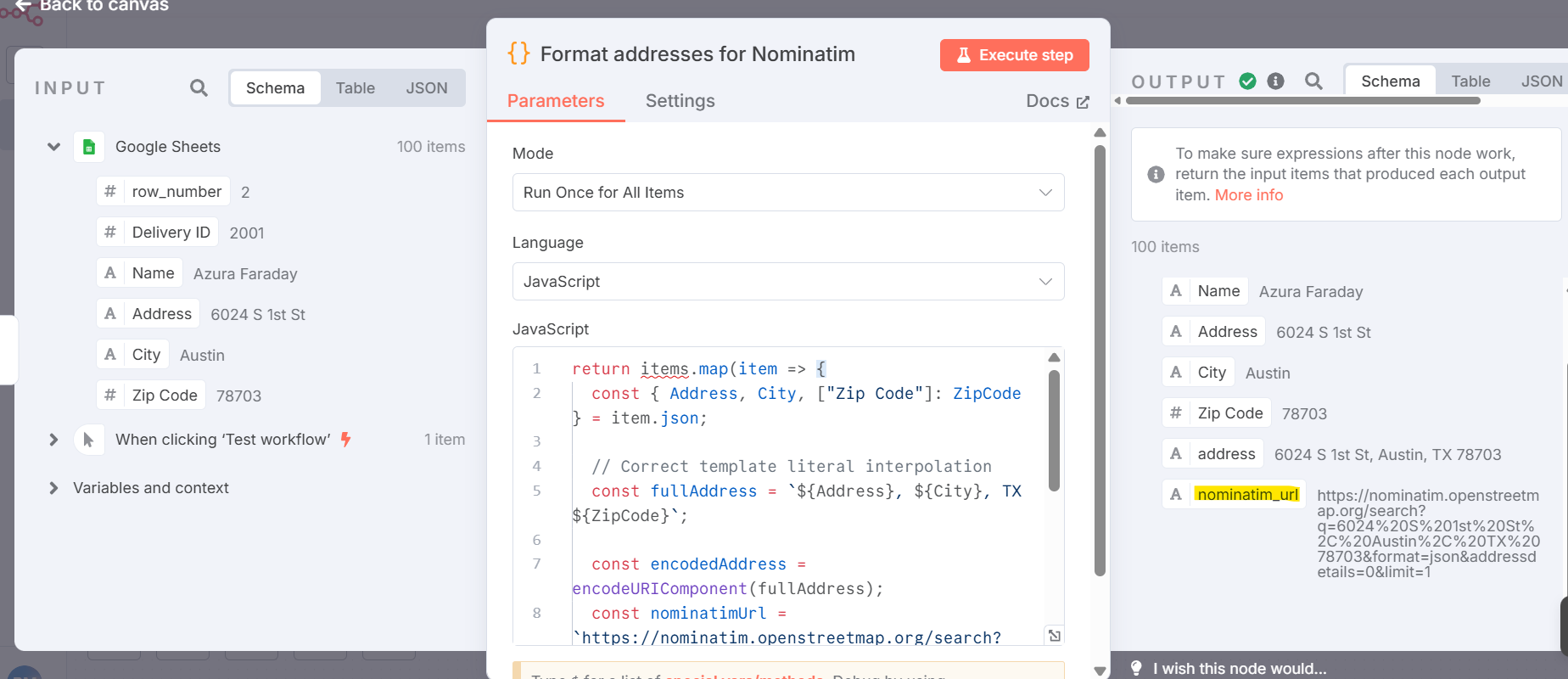
nominatim\_url: nominatimUrl

}

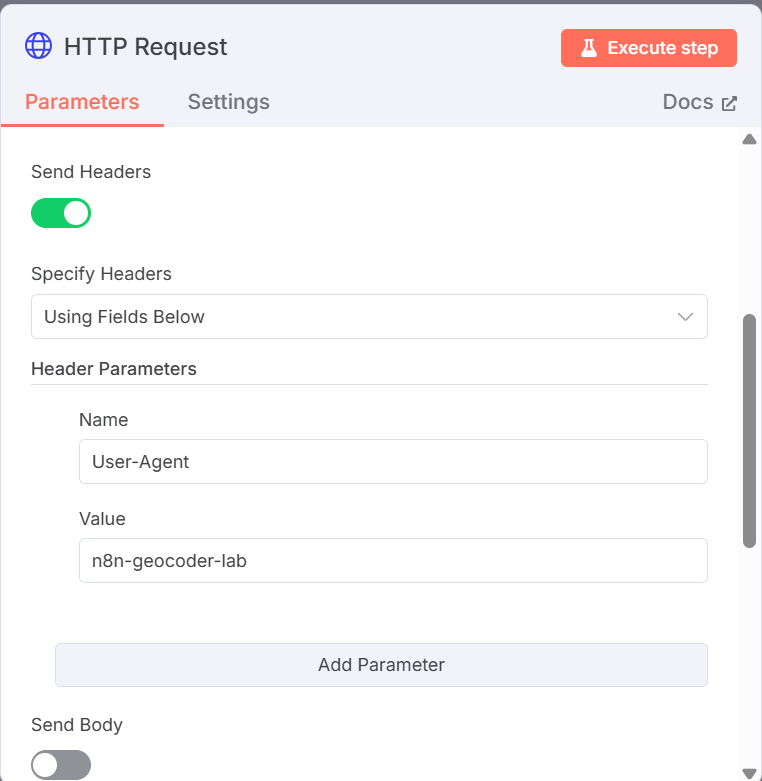
};

});

* 1. Generate a URL to query OpenStreetMap’s Nominatim API.



* 1. Add an \*\*HTTP Request\*\* node to perform the GET request.
     1. Set header `User-Agent` to `n8n-geocoder-lab`.



* 1. Use a \*\*Merge\*\* node to recombine delivery data with geocoding result.
     1. Mode: Comine
     2. Combine By: Position

A screenshot of a computer

AI-generated content may be incorrect.

1. ✅ Step 6: Group Deliveries by Driver (Clustering).
   1. Add a \*\*Code\*\* node `"Wrap Deliveries"`.

return [

{

json: {

deliveries: items.map(item => item.json)

}

}

];

* 1. Add a \*\*Merge\*\* node to combine it with `"Number of Drivers"`.
     1. Mode: Comine
     2. Combine By: Position

A screenshot of a computer

AI-generated content may be incorrect.

* 1. Add a \*\*Code\*\* node `"K-Means"` to assign `DriverGroup` using lat/lng.

function distance(a, b) {

return Math.sqrt((a[0] - b[0]) \*\* 2 + (a[1] - b[1]) \*\* 2);

}

function kMeans(points, k, maxIterations = 100) {

const centroids = points.slice(0, k).map(p => [...p]);

let labels = new Array(points.length).fill(0);

for (let iter = 0; iter < maxIterations; iter++) {

labels = points.map(p => {

const dists = centroids.map(c => distance(p, c));

return dists.indexOf(Math.min(...dists));

});

const newCentroids = Array.from({ length: k }, () => [0, 0]);

const counts = Array(k).fill(0);

points.forEach((p, i) => {

const label = labels[i];

newCentroids[label][0] += p[0];

newCentroids[label][1] += p[1];

counts[label]++;

});

for (let i = 0; i < k; i++) {

if (counts[i] === 0) throw new Error(`Cluster ${i + 1} has no assigned points.`);

centroids[i][0] = newCentroids[i][0] / counts[i];

centroids[i][1] = newCentroids[i][1] / counts[i];

}

}

return labels;

}

// Extract structured input

const input = items[0].json;

if (!input.deliveries || !Array.isArray(input.deliveries)) {

throw new Error("Missing or invalid 'deliveries' array");

}

const numDrivers = parseInt(input.Number\_of\_Drivers, 10);

if (isNaN(numDrivers) || numDrivers < 1) {

throw new Error("Invalid or missing 'Number\_of\_Drivers'");

}

// Convert coordinates

const coords = input.deliveries.map((d, i) => {

const lat = parseFloat(d.lat);

const lon = parseFloat(d.lon);

if (isNaN(lat) || isNaN(lon)) {

throw new Error(`Invalid lat/lon at delivery index ${i}`);

}

return [lat, lon];

});

// Run clustering

const labels = kMeans(coords, numDrivers);

// Return updated deliveries with DriverGroup

return input.deliveries.map((d, i) => ({

json: {

...d,

DriverGroup: `Driver ${labels[i] + 1}`

}

}));

1. ✅ Step 7: Sort Each Route (Nearest Neighbor).
   1. Add \*\*Code\*\* node `"List Sort"` to group by `DriverGroup`.

const grouped = {};

for (const item of items) {

const d = item.json;

const group = d.DriverGroup;

if (!group) {

throw new Error(`Missing 'DriverGroup' on delivery ID ${d["Delivery ID"]}`);

}

if (!grouped[group]) {

grouped[group] = [];

}

grouped[group].push(d);

}

// Wrap into one output item

return [

{

json: grouped

}

];

* 1. Add \*\*Code\*\* node `"Nearest Neighbor"` to assign `StopOrder`.

function distance(a, b) {

const dx = a.lat - b.lat;

const dy = a.lon - b.lon;

return Math.sqrt(dx \* dx + dy \* dy);

}

function nearestNeighbor(deliveries) {

const unvisited = [...deliveries];

const ordered = [];

// Start at the first delivery

let current = unvisited.shift();

ordered.push(current);

while (unvisited.length > 0) {

let nearestIndex = 0;

let minDistance = distance(current, unvisited[0]);

for (let i = 1; i < unvisited.length; i++) {

const d = distance(current, unvisited[i]);

if (d < minDistance) {

minDistance = d;

nearestIndex = i;

}

}

current = unvisited.splice(nearestIndex, 1)[0];

ordered.push(current);

}

return ordered;

}

// Sort each group

const grouped = items[0].json;

const sortedGroups = {};

for (const driver in grouped) {

const deliveries = grouped[driver];

const enriched = deliveries.map(d => ({

...d,

lat: parseFloat(d.lat),

lon: parseFloat(d.lon)

}));

const sorted = nearestNeighbor(enriched);

// Optionally add a stop index

sorted.forEach((d, i) => d.StopOrder = i + 1);

sortedGroups[driver] = sorted;

}

return [

{

json: sortedGroups

}

];

1. ✅ Step 8: Output Final Assignments.
   1. Option A: Export to Trello
      1. Add a \*\*Trello node\*\*
      2. Create 1 list per `DriverGroup`.
      3. Add delivery cards per stop.
   2. Option B: Export to File
      1. Add a \*\*Write to File\*\* node.
      2. Output `.json` or `.md` to local disk.

🔚 Final Output Fields Per Delivery

|  |  |
| --- | --- |
| Field | Description |
| `Delivery ID` | Unique identifier |
| `Name` | Customer or task name |
| `Address` | Full formatted address |
| `lat/lon` | Geocoded location |
| `DriverGroup` | Assigned driver (e.g. Driver 1) |
| `StopOrder` | Sequence within route |