ChatGPT

NYDroneAI – Drone Logistics Simulation & Analytics Suite Documentation

Introduction

NYDroneAI is an interactive, AI-driven logistics simulation dashboard for drone delivery operations in New York City [1]. It allows users to simulate drone delivery missions across NYC's five boroughs and analyze performance through rich visual analytics. The platform is built with Streamlit for a web-based UI, and integrates data science and AI components to provide both realistic simulation of operations and intelligent decision support.

This documentation serves both as a **user guide** (how to deploy and use the dashboard) and a **developer guide** (explaining the system's architecture, code components, and how AI is integrated). It includes high-level overviews, technical details, architecture diagrams, and instructions for deployment across various environments.

Features and Capabilities

NYDroneAI comes with a comprehensive set of features to simulate and analyze drone logistics:

- **Drone Delivery Simulation:** Create a fleet of drones and a batch of delivery orders, then simulate assignment and execution of deliveries with realistic constraints (random route planning, potential blocked paths, battery limitations, etc.) [2]. Each drone delivery attempt can succeed or fail based on configurable probabilities (e.g. base failure rate, route block failures, low battery) and triggers maintenance cycles for drones after certain usage.
- **AI-Powered Dispatching:** Optionally utilize a Flan-T5 large language model (via LangChain) to assist in assigning drones to deliveries. The AI analyzes each delivery request and available drones to recommend the optimal drone for the job [2] [3]. This adds an intelligent decision-support layer on top of the baseline simulation.
- **Interactive Visual Analytics:** After each simulation run, the dashboard presents key metrics and charts. These include overall success vs. failure rates (in percentage and a visual pie chart)

, a breakdown of failure reasons, delivery outcomes per hub (bar charts), and distribution of payload weights. Users can also view time-based trends (e.g. deliveries per hour) and inspect a detailed log of every delivery attempt.

- **Geospatial Visualization:** The platform uses Folium maps to plot delivery outcomes geographically. An interactive map shows New York City with markers for each delivery drop location – green for successful deliveries and red for failed ones – as well as icons for hub locations (drone launch sites). Users can toggle a heatmap view to visualize the density of successful deliveries across the city [4]. This geospatial view helps in understanding regional performance and identifying any area-specific issues (e.g., certain hubs having more failures).
- **Drone Fleet Metrics:** The dashboard provides a tabular summary of each drone's performance (deliveries completed, current status, remaining battery, etc.) after a simulation. This helps in analyzing drone utilization and identifying drones that may be overworked or frequently failing.
- **Multi-Run Statistical Analysis:** Users can run the simulation multiple times in batch to gather statistics over many trials [5]. In multi-run mode, the dashboard will show the distribution of success rates across runs and compute the average success rate, helping to identify performance variability and reliability. This is useful for scenario testing and understanding probabilistic outcomes (e.g., how often the success rate falls below a threshold).
- **Real-Time Progress Mode:** The simulation can run in an "Instant" mode (fast execution with immediate results) or a "Real-time Progress" mode [6]. In real-time mode, a progress bar is displayed as deliveries are processed one by one, which can help demonstrate the simulation flow in a presentation or educational setting.
- **Configurable Parameters:** A wide range of simulation parameters can be adjusted via the sidebar before running a simulation [7]. These include the number of drones in the fleet, number of delivery orders, base failure probability, probability of route block failures, battery failure threshold, and the ability to filter results by specific hubs. There is also an "Advanced Analytics" toggle to show or hide more detailed data breakdowns.

With these features, NYDroneAI provides a full end-to-end suite for modeling a drone delivery operation – from dispatching and routing to failure handling and post-mission analytics – all within an easy-to-use dashboard interface.

Technology Stack

NYDroneAI leverages a modern Python-based tech stack to achieve its functionality [8] :

- **Language & Backend:** Python is the core language, powering the simulation logic (random processes, data handling with Pandas/NumPy) and integration with machine learning libraries.
- **Web Framework: Streamlit** is used for the web dashboard UI, enabling rapid development of interactive controls and live charts in a browser.
- **Data & Visualization: Pandas** is used for data manipulation (e.g., converting logs to dataframes for analysis). Visualizations are created with **Matplotlib** and **Seaborn** for charts (pie charts, bar graphs, histograms) and **Folium** (for interactive maps). The Folium integration (via `streamlit-folium`) allows embedding maps in the Streamlit app.
- **Geospatial & Route Planning:** The project includes geospatial tools such as **Geopy**, **GeoPandas**, and **Shapely** for handling coordinates and distances. For route planning, **NetworkX** is listed (with support for algorithms like Dijkstra or A* for shortest paths) [9] . In a realistic scenario, NetworkX could be used with city road network data to simulate actual flight paths and identify blocked segments dynamically. (The current simulation uses a simplified random route generator, but the infrastructure is in place for more advanced routing with NetworkX.)
- **Database: SQLite** is used for data persistence [10] . The simulation can log results (each delivery attempt and maintenance event) to a local SQLite database file (`nydroneai.db`). This allows historical analysis, querying past simulation runs, or persisting data between sessions. The `sqlite-utils` library is used to facilitate easy SQLite interactions.
- **AI & NLP:** The AI dispatching feature uses **Hugging Face Transformers** with a **Flan-T5** model, orchestrated via **LangChain** [11] . The HuggingFace `transformers` library loads the pre-trained Flan-T5 model (a small variant for language tasks), and LangChain provides prompt templating and chain management to structure the decision-making process (as described later in *AI Decision Module*). PyTorch (`torch`) and `accelerate` are utilized under the hood for model execution.

This diverse stack allows the project to blend data science, machine learning, and web interactivity seamlessly in one application.

Setup and Installation

To set up the NYDroneAI dashboard on your local machine for the first time, follow these steps:

1. **Clone the Repository:** Obtain the project code. For example, using Git:

   ```
   git clone https://github.com/<your-username>/NYDroneAI-Streamlit-
   Dashboard.git
   && cd NYDroneAI-Streamlit-Dashboard
   ```

   (If you've received the code via a ZIP or other means, simply ensure all files are in a single project directory.)
2. **Create a Virtual Environment (Optional but Recommended):** Create an isolated Python environment to avoid dependency conflicts. For example:

```
python -m venv env
```

Then activate it with `source env/bin/activate` on Linux/MacOS, or `env\Scripts\activate` on Windows.

3. **Install Dependencies:** Use pip to install required Python libraries:

   ```
   pip install -r requirements.txt
   ```

   This will install Streamlit, Pandas, Folium, Matplotlib, NetworkX, HuggingFace Transformers, LangChain, and all other necessary packages [12]. (Ensure you have an internet connection for pip to download packages.)
4. **Initial Model Setup:** The first run will download the pre-trained **Flan-T5 Small** model (around 300MB) if not already present. This happens automatically when the code invokes HuggingFace pipelines [13]. No action is needed from you except to be patient on the first run while the model downloads. Having a GPU is *not* required for the small model; CPU execution will work (though a GPU will speed up AI computations if available).
5. **Launch the Streamlit App:** Start the web app by running the Streamlit command:

   ```
   streamlit run Dashboard.py
   ```

   Streamlit will spin up a local web server and open a browser window. By default, you can access the app at **http://localhost:8501** [14]. You should see the NYDroneAI dashboard interface load in your browser.

After these steps, the application is ready to use. The project structure should look like this (files and their purpose) [15]:

```
  NYDroneAI-Streamlit-Dashboard/
├── Dashboard.py        # Main Streamlit app script containing UI and simulation
logic
├── requirements.txt    # Python dependencies list
├── data/               # Folder where simulation data/logs can be saved (if
enabled)
├── nydroneai.db        # SQLite database file (created at runtime for logging)
└── README.md           # Project overview and setup (this documentation is a
more detailed version)
```

(Note: The `nydroneai.db` file and `data/` folder will appear after you run a simulation that performs logging; they are not present initially.)

Using the Dashboard (User Guide)

Once the app is running locally (or deployed on a server), open it in your web browser. You will be greeted with the **NYDroneAI: Full Enterprise Drone Analytics Suite** interface (dark-themed dashboard). The screen is divided into two main sections: a left sidebar for **Simulation Controls** and the main panel on the right for **Simulation Output and Visualizations**.

Screenshot of the NYDroneAI Streamlit dashboard interface, showing the left sidebar controls and the main panel (before running a simulation). Users can adjust parameters like failure rates, number of drones, deliveries, etc., then click Run Simulation to execute.

1. Configuring Simulation Parameters

Use the **sidebar on the left** to configure the simulation settings before running. Key controls include:

- **Base Failure Rate (%):** The overall baseline chance that a delivery fails due to random issues (weather, drone malfunction, etc.). For example, a 9% base failure means roughly 9 out of 100 deliveries might fail on average purely due to random events [7].
- **Blocked Segment Fail Rate (%):** Probability of failure if a route segment is blocked. The simulation can randomly designate a segment of the drone's flight path as blocked (e.g., due to airspace restrictions or obstacles). This slider (e.g., 20% by default) represents the chance that a delivery will *actually fail* when such a detour is required [7]. Higher values make route disruptions more dangerous.
- **Battery Failure Threshold (%):** This simulates drone battery sufficiency. A delivery will fail if the required energy for the trip exceeds a certain percentage of the drone's battery. For instance, a 90% threshold means the drone must have at least 90% battery of the needed energy; otherwise the mission is considered a failure due to low battery [16]. Lowering this threshold makes drones more likely to abort if they don't have ample battery for a delivery.
- **Number of Drones:** Size of your drone fleet in the simulation. You can choose between 10 and 200 drones (default 70). Each drone is generated with random attributes (a home hub, a starting battery level between 60–100%, payload capacity, etc.) [17].
- **Number of Deliveries:** How many delivery orders to simulate in one run (between 50 and 1500, default 350). Each delivery will be assigned to a random hub (origin location) and has a random payload weight and destination coordinates near that hub [18].

- **Simulation Mode:** Choose **Instant** vs **Real-time Progress**. In Instant mode, the simulation runs as fast as possible and then displays results. In Real-time mode, you will see a progress bar updating as deliveries are processed one by one [19] [20], adding a slight delay for demonstration purposes.
- **Multi-Run Simulation:** If you want to run the simulation multiple times back-to-back (for statistics), set this number > 1. For example, setting *Multi-Run = 5* will execute 5 simulation runs in a row. The dashboard will then aggregate results (showing distribution of success rates, etc.) instead of just one run's outcome.
- **Filter by Hub:** By default, all five hubs (Manhattan, Brooklyn, Queens, Bronx, Staten Island) are included. You can select a subset – for example, only "Manhattan" and "Brooklyn" – to focus the results/visuals on those hubs [21]. This filters the output data (deliveries from other hubs will be ignored in charts and maps).
- **Show Advanced Analytics:** A checkbox that toggles the display of some additional detailed outputs. When checked, after a run the dashboard will show an "Advanced: Under-the-hood summary" which includes a pivot table of outcomes by hub and status (a quick table grouping deliveries by hub and whether they succeeded or failed) [22]. If you prefer a simpler view, you can uncheck this to hide that extra detail.

Feel free to adjust these parameters to design different scenarios. For example, you might simulate a small fleet with a large number of deliveries to see how overloaded drones perform, or increase failure probabilities to stress-test the system's robustness.

2. Running the Simulation

After setting the parameters, click the **"Run Simulation"** button at the top of the main panel [23]. The dashboard will begin the simulation:

- In **Instant mode**, results will appear almost immediately (the simulation loop runs to completion behind the scenes, which usually takes a couple of seconds for a few hundred deliveries).
- In **Real-time Progress mode**, a progress bar will appear and fill as each delivery is processed [24] [25]. This mode is slower but lets you observe the simulation in motion. It's useful for demonstrations or if you want to conceptually follow each step.

If you set **Multi-Run > 1**, the simulation will loop for the specified number of runs. In this case, the UI will display aggregated statistics (like a histogram of success rates across runs) instead of the usual single-run charts. After completing all runs, the results will be shown.

3. Viewing Results and Analytics

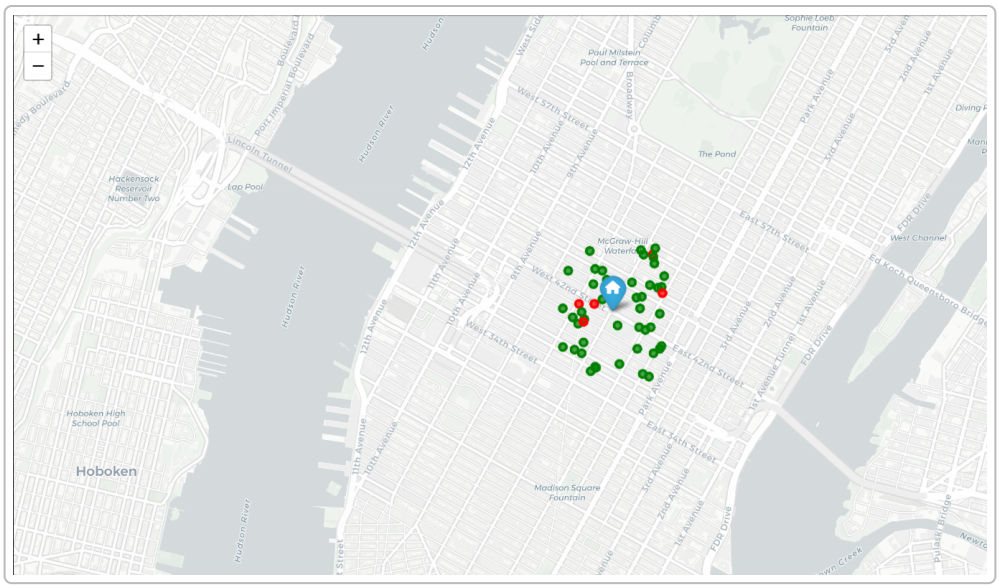
Once the simulation finishes, the dashboard populates the main panel with a variety of outputs. These outputs are organized with headings and expandable sections for clarity:

- **Simulation Summary Metrics:** At the top, you'll see key numeric indicators: total deliveries attempted, how many succeeded (delivered), how many failed, and the overall success rate as a percentage【2†L159-167】. For example, it might show *"Total Deliveries: 350, Delivered: 290, Failed: 60, Success Rate: 82.9%"*. It will also highlight the most common failure reason (e.g., "Most Common Failure: battery too low").
- **Mission Outcome Pie Chart:** A pie chart titled "Mission Success vs. Failure" visualizes the ratio of successful vs failed deliveries [26] [27]. Green represents successful deliveries and red represents

failed ones, with percentage labels. This gives a quick visual grasp of reliability. For instance, in a scenario, you might see about 83% green and 17% red (as in the example below).

Example pie chart output from the dashboard, illustrating the proportion of successful (green) vs failed (red) deliveries in a simulation run. In this sample, ~83% of deliveries succeeded and ~17% failed.

- **Failure Reasons Bar Chart:** A bar chart breakdown appears (if there were any failures) showing the count of each failure reason [28]. Common reasons include weather, drone malfunctions, GPS loss, "no path available" (if a route was completely blocked), "battery too low," etc. This helps diagnose why failures happened. If no failures occurred at all, the dashboard will simply note that there were no failures to show [29].
- **Deliveries by Hub:** Another bar graph groups deliveries by their assigned hub, splitting by delivered vs failed [30]. Each hub (Manhattan, Brooklyn, etc.) has a stacked bar: green portion for successful deliveries and red for failures. This allows you to compare performance across hubs – e.g., you might observe that one hub had significantly more failures than others, indicating possible location-specific challenges.
- **Geographical Delivery Map:** Under an expandable section "🗺 Show Map of Deliveries," you'll find an interactive map of NYC with markers for each delivery attempt [31] [32]. Each marker's color indicates success (green dot) or failure (red dot), and clicking a marker shows a popup with details: delivery ID, which hub it originated from, and failure reason if any. The five hub locations are marked with blue home icons on the map for reference. This spatial view (see image below) helps you see where successful deliveries clustered and if failures tend to happen farther from hubs or in certain areas.



Interactive map visualization of delivery outcomes in NYC. Green markers denote successful deliveries and red markers denote failed deliveries. Blue house icons mark the drone hub locations (Manhattan, Brooklyn, Queens, Bronx, Staten Island). This allows users to visually assess how deliveries fared geographically – for example, noticing any clusters of red indicating problem areas.

- **Delivery Density Heatmap:** In the " Show Delivery Heatmap" section, the dashboard can display a heatmap overlay on the map highlighting areas with higher concentrations of successful deliveries [33] . Areas where many deliveries succeeded will glow with warmer colors (yellow/red), indicating high throughput regions. This is another way to visualize coverage and performance intensity. (If no deliveries succeeded, the heatmap will be empty with just a note.)
- **Payload Distribution Histogram:** Under "⚖ Show Payload Weight Distribution," a histogram shows the distribution of package weights for the deliveries [34] . This uses Seaborn to illustrate how many deliveries fell into various weight ranges. It can be useful to verify that the random payload generator is producing a reasonable spread (e.g., if most payloads are small vs heavy) and to consider if weight correlates with failures.
- **Deliveries Per Hour Chart:** Under " Deliveries Per Hour," a bar chart shows how many deliveries occurred at each hour of the day (the simulation timestamps deliveries in the order they are processed) [35] . This is mainly interesting if you run in real-time mode or multiple runs – it will illustrate at what hours the simulated deliveries happened. Since the simulation starts all deliveries essentially at time 0 and runs quickly, this chart might show a spike at the hour corresponding to simulation start, but in extended or real-time simulations it could mimic a timeline of operations.
- **Drone Utilization Table:** In the "🏷 Drone Utilization Table" section, a sortable table lists each drone in the fleet and some end-of-simulation stats [36] [37] . Columns include Drone ID, Home Hub, Deliveries Completed, Final Battery %, and Status. This lets you see, for example, which drones completed the most deliveries or which ended up in "maintenance" status. It also helps identify if a subset of drones were doing most of the work (e.g., if the assignment logic favored certain drones).
- **Full Delivery Log & Download:** Finally, under "📝 Full Delivery Log / Download," you can inspect the raw log of deliveries. This is a table (capped at 500 entries shown for performance; if you simulated more, it notes that and suggests downloading) with each delivery attempt as a row [38] . Each row includes drone ID, delivery ID, status (delivered/failed), failure reason (if failed), timestamp, hub, and

payload weight. There is also a **Download button** that allows you to save the entire log as a CSV file [39]. This is useful if you want to do further analysis in Excel, Pandas, or another tool outside the app.

After reviewing results, you can adjust parameters and run another simulation to explore different scenarios. The dashboard is meant to be interactive – tweak something, run again, and immediately see how the outcomes change.

4. Multi-Run Results Interpretation

If you used the *Multi-Run Simulation* feature (with more than 1 run), the output will be slightly different:

- You will see a section **"📈 Multi-Run Simulation Stats"** with a histogram showing the distribution of success rates across all the runs [40]. For example, if you ran 10 simulations, it might show how many runs achieved 80–85% success, how many 85–90%, etc. This gives a sense of variability.
- The average success rate across all runs is displayed below the histogram [41]. You can use this to compare scenarios more objectively than a single run (since any single simulation run has randomness).
- The single-run charts (pie, failure reasons, etc.) will not be shown in multi-run mode, since each run could have different breakdowns. The focus in multi-run is on aggregate statistics.

If you want to see individual run details in a multi-run, you would need to run those simulations one at a time (or incorporate logging to the database for each run and analyze later).

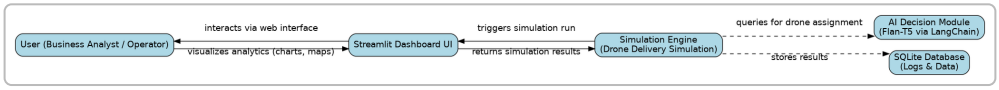
5. Advanced Mode

When **"Show Advanced Analytics"** is checked, after all the standard outputs, the dashboard displays an **"Advanced: Under-the-hood summary"**. This is essentially a pivot table grouping the results by hub and status [22]. It shows a small table with each hub as a row and two columns (Delivered, Failed) indicating the count of deliveries for that hub that succeeded or failed. This replicates information from the bar chart in a precise table form, which can be handy for cross-checking exact numbers or copying into reports.

This advanced section is mostly for those who want to see the raw data distribution. It can be turned off for a cleaner view if not needed.

Architecture and Implementation (Developer Guide)

Under the hood, NYDroneAI is composed of several logical components working in tandem. The high-level architecture is illustrated below:



High-level architecture diagram of the NYDroneAI system. The Streamlit Dashboard UI handles user interaction and visualization, communicating with the Simulation Engine that runs the drone delivery simulation. An optional AI Decision Module (Flan-T5 via LangChain) can be queried by the simulation to choose drones intelligently.

Simulation results can be stored in a SQLite Database for persistence. The UI then presents analytics (charts, maps) back to the User in the browser.

Streamlit Dashboard UI

The entire front-end interface is built using Streamlit, which runs Python code and renders interactive components in a web app format. Key aspects of the UI implementation:

- **Sidebar Controls:** Defined at the top of `Dashboard.py`, the sidebar uses Streamlit widgets like sliders, radio buttons, multiselect, and checkboxes to gather user input for all simulation parameters [7] [42]. Each control's value is stored in a Python variable (e.g., `FAILURE_RATE`, `NUM_DRONES`) for use in the simulation. Streamlit automatically re-runs the script when inputs change or the user clicks the Run button.
- **Run Button:** The **Run Simulation** button (`st.button("Run Simulation")`) triggers the main simulation execution block [25]. In Streamlit, clicking this button causes the app to go through the code under the `if st.button("Run Simulation"):` condition.
- **Dynamic Content Rendering:** Depending on whether a simulation is run and whether multi-run mode is selected, different content is shown. The app uses conditionals to either display multi-run stats or single-run results. Streamlit's ability to render charts (using `st.pyplot` for Matplotlib figures [27], or `st.dataframe` for tables [43], etc.) and maps (`st_folium` to render Folium maps [44]) is used extensively.
- **Expanders:** To keep the interface organized, less immediately critical visuals (maps, heatmaps, detailed tables) are placed inside `st.expander` sections [45] [46]. The user can click these to expand or collapse additional details.
- **Layout:** The app is set to wide mode (`st.set_page_config(layout="wide")` [47]), utilizing the full browser width for a better dashboard experience. Results are arranged using columns (for summary metrics) and subheaders for clarity.

In essence, the Streamlit UI ties everything together: it captures input, calls the simulation functions, and displays outputs. Developers can modify `Dashboard.py` to add new controls or visualizations. The UI code is reactive – meaning any change in inputs or state will recalculate and refresh outputs, which is convenient for interactive experimentation.

Simulation Engine (Drone Delivery Simulation Logic)

The core of the application is the simulation engine, which is responsible for generating drone and delivery data, simulating the dispatch of drones to deliveries, and determining outcomes. The simulation logic can be found in the functions within `Dashboard.py` (and was prototyped in a Jupyter notebook for development). Here's how it works:

- **Data Generation:** When a simulation run starts, the code generates a dictionary of drones and a list of deliveries based on the specified numbers:
- `gen_drones(num)` creates `num` drone objects [48], each with random attributes:
  - A unique ID (e.g., "D1", "D2", ...).
  - An assigned home hub (randomly one of the five NYC boroughs).
  - A starting battery level (random integer between 60% and 100%).
  - A payload capacity (random float between ~1.0 and 5.0 kg).

    - Status (initially "idle") and a counter for deliveries completed (initially 0).
- `gen_deliveries(num)` creates `num` delivery orders [18]. Each delivery has:
    - A unique delivery ID (e.g., "D0001", "D0002", ...).
    - An assigned hub (randomly chosen, representing which hub will dispatch the drone for this delivery).
    - A payload weight (random between 0.5 and 4.5 kg).
    - A delivery location (latitude/longitude coordinates) which is a random point roughly around the hub location (the code adds a small random offset to the hub's coordinates to simulate a drop-off point in the vicinity).
- **Dispatch Loop:** The main simulation loop (`run_simulation`) iterates over each delivery in the list [49]. For each delivery:
- It first filters the available drones to find **eligible drones** for that delivery [50]. An eligible drone is one that:
    - Is located at the same hub as the delivery's origin (we assume drones can only serve deliveries from their home hub for simplicity).
    - Has enough battery (>50% by default) and is currently idle (not in maintenance or already busy).
    - Has sufficient payload capacity for the package weight.
- If no eligible drone is available at that hub, the delivery is immediately marked as failed due to "ineligible drone" [51] (no drone could take it).
- If one or more drones are eligible, the simulation must choose one to dispatch. In the current implementation, **if the AI module is not used, a random eligible drone is selected** [52] (`random.choice(eligible)`). This represents a random dispatch or first-come-first-serve assignment. If the AI Decision Module is enabled (see next section), the selection could instead be made by the AI's recommendation.
- **Route Simulation:** Once a drone and delivery are paired, the simulation generates a route and determines if any complications occur. The function `simulate_route_metadata(drone, delivery)` returns a route info dictionary [53] including:
- `distance_m`: A random distance for the delivery route (e.g., between 500 and 8000 meters).
- `eta_min`: An estimated time of arrival in minutes (random 5 to 30 minutes).
- `energy_cost_percent`: How much battery this trip would consume (random 5% to 40% of battery).
- `detour`: A flag indicating if a route detour was needed due to a blocked segment (simulated with a 10% chance inside this function). If a detour is needed, the route's `path` may be empty or flagged as problematic.
- (In an enhanced version of the simulation using NetworkX, this function would compute an actual shortest path on a map graph and potentially remove a segment to simulate a blockage, forcing a detour. The prototype notebook demonstrates printing out blocked path segments [54]. However, the simplified version just uses a random flag.)
- **Outcome Determination:** Given the route info, the `evaluate_delivery(drone, delivery, route, fail_rate, block_fail_prob, battery_thresh)` function is called to decide success or failure [55]. The logic is:
- If the route path is empty (meaning a detour was required), then with probability = `Blocked Segment Fail Rate` the delivery fails (otherwise, if lucky, the drone still manages despite detour) [55].

- If the route's energy cost exceeds the drone's current battery * `Battery Failure Threshold`, then the delivery fails due to insufficient battery [56]. (E.g., if threshold is 90% and the drone has 80% battery, it cannot take a job that needs >72% battery).
- Otherwise, with probability = `Base Failure Rate` the delivery may fail due to a random issue [57] (and a random failure reason like "weather" or "GPS loss" is chosen).
- If none of those conditions trigger a failure, the delivery is marked **successful**.
- **Logging the Result:** The outcome is recorded via `log_delivery_result(drone_id, delivery, success, reason, log)` which appends an entry to an in-memory log list for this run [58] [59]. Each log entry has details like drone ID, delivery ID, status, failure reason (None if success), timestamp, delivery location, hub, payload, etc.
- **Post-Delivery Updates:** If the delivery was successful, the drone's record is updated: its `deliveries_completed` count increments, and its status might change to "maintenance" if it has completed a multiple of 10 deliveries [60] [61]. (Every 10th delivery triggers a maintenance event, simulating the need for periodic check-ups. In maintenance, the drone won't be eligible for new deliveries until maintenance is resolved.)
- The function `resolve_maintenance(drones)` is called to immediately clear any drones in maintenance back to idle [62] (for simplicity, we assume maintenance happens instantly at the end of the delivery cycle or that the maintenance downtime is short within the simulation run).
- Essentially, maintenance in this simulation is more of a counter for how many deliveries a drone has done, rather than introducing a time delay; a more advanced version could persist a "maintenance" status so the drone sits out some number of delivery cycles.
- **Completion:** The loop continues until all deliveries are attempted. The function then returns the log of all deliveries (`result_log`). If the simulation was a multi-run batch, this whole process is repeated N times; the multi-run stats function collects the success rates of each run into a list and produces summary statistics [63] [40].

The simulation engine is thus a mix of deterministic rules and random stochastic elements. Developers can adjust these functions to change how outcomes are determined (for instance, plugging in a more sophisticated battery drain model, or integrating actual route calculations). The code is structured to easily plug in new logic – for example, replacing the random drone selection with an AI-based selection, or using real map data for routes.

AI Decision Module (Intelligent Drone Assignment)

One of the standout features of NYDroneAI is its integration of an AI language model to assist with decision-making. This is implemented using Hugging Face's **Flan-T5 Small** model and orchestrated through **LangChain**. The AI module's goal is to analyze a delivery request and suggest the best drone for that delivery, providing a more informed assignment than a random pick.

How it works: In the development notebook (and planned for the app), the following LangChain prompt chain is defined:

1. **Delivery Summarization Prompt:** *"Summarize this delivery request for a drone dispatcher..."* – The raw data of a delivery (delivery ID, hub, payload, etc.) is fed into a prompt that asks the LLM to summarize the key details in a concise manner [64]. This could produce a summary like: *"Delivery D0004 from Brooklyn hub, payload 3.2 kg, destination ~1.5 km away. Drone needs ~15% battery for trip."* (The model infers or is told distance/battery info if available).

2. **Route Analysis Prompt:** Given the summary, a second prompt is: *"Given the delivery summary, estimate potential risks, optimal routes, and urgency level."* [65] The model might output something like: *"Route likely requires crossing water; risk of wind high. Optimal route: direct south then east; urgency: medium."* This step is meant to let the AI reason about any factors that might influence which drone to choose (e.g., maybe a longer route needs a drone with higher battery, or a riskier route needs a more reliable drone).
3. **Drone Selection Prompt:** Finally, the AI is given the analysis plus a list of candidate drone IDs (eligible drones), and prompted: *"Based on the analysis, select the best drone ID from this list: [D1, D5, A3, ...]. Only output the ID."* [66] The model then outputs one ID, presumably the drone it deems most suitable. For example, it might output **"D5"** if Drone D5 had the highest battery or some advantage. The chain uses a template that ensures the answer is just an ID with no extra text.

These three prompts are linked in a LangChain **SequentialChain**, so the output of one feeds into the next [67]. The Flan-T5 model (a text-to-text model) is loaded and used via the HuggingFace pipeline integrated with LangChain's `HuggingFacePipeline` wrapper [68] [64]. This allows treating the model as if it were an `LLM` in LangChain, enabling use of `LLMChain` for each prompt.

Integration with simulation: When this AI module is enabled, the simulation's drone selection step would be modified to call the AI chain instead of random selection. For example, instead of `random.choice(eligible)`, the code would do something like: `chosen_drone = orchestrator.run(delivery=delivery_details, candidates=eligible_ids)`, where `orchestrator` is the SequentialChain. The result `chosen_drone` is the ID the model picked [3] [69]. The simulation then uses that drone for the delivery. If the AI for some reason picks an invalid drone (one not in the list), the code can catch that and log it as a failure (as was done in testing, logging "FLAN_FAIL" for an invalid suggestion) [69].

Example: Suppose a delivery from Manhattan hub has 5 candidate drones [A1, A2, A3, A4, A5] available. Perhaps A4 has the highest battery and lowest prior workload. The AI might output "A4" [70] as the best choice after analyzing. The simulation then dispatches drone A4 for that job and proceeds.

The AI module adds a layer of decision-making that can consider more context (like payload weight, distance, etc.) rather than a simplistic rule. It's especially powerful as such language models can encode expert knowledge or learned strategies (for instance, always use drones from the same hub, or avoid using almost-depleted drones for long trips). In practice, Flan-T5 small is a relatively limited model, so its decisions might not always be optimal; however, this setup is a proof-of-concept for how larger or more specialized models could be integrated in dispatch decisions.

Developers can tweak the prompt templates or chain structure to improve the AI's performance. For instance, adding more details to the summary or analysis prompt, or providing the drones' battery levels and payload capacities in the prompt, would likely yield better choices. One could also swap in a larger model (like Flan-T5 Base or even GPT-3.5 via API) by adjusting the HuggingFace pipeline or LangChain LLM used – the rest of the integration would remain the same.

Data Logging and Persistence

For analysis beyond the live dashboard, NYDroneAI can log data to a SQLite database (`nydroneai.db`). Using SQLite provides persistence across runs and a way to query or join data for deeper analysis.

- **Logging Deliveries:** Each delivery attempt can be inserted into a database table (e.g., `deliveries_log`) with all its details. In the prototype, `log_delivery_result` was written to append to a list and also optionally insert into a database via `sqlite-utils` if a connection is open [69]. The fields would include drone_id, delivery_id, hub, payload, success/fail, reason, timestamp, etc. This essentially mirrors the CSV download but in a structured database form.
- **Logging Maintenance:** Similarly, events like a drone entering maintenance could be logged to a `maintenance_log` table (drone_id, time, reason). The code shows an example of logging a maintenance trigger: `log_maintenance(conn, drone_id, reason)` was used in development [71].
- **Data Schema:** The schema is simple. For instance, a **Deliveries** table might be: `(delivery_id TEXT, drone_id TEXT, hub TEXT, payload REAL, status TEXT, failure_reason TEXT, timestamp TEXT)`. Drone IDs and delivery IDs can serve as keys to join or index data. A **Maintenance** table might have: `(drone_id TEXT, event TEXT, timestamp TEXT)` for each maintenance occurrence.
- **Using the Data:** With the data in SQLite, developers or analysts can run SQL queries to answer questions like "Which hub has the highest failure rate overall?" or "How does average success rate change with number of drones deployed?". This is beyond the scope of the Streamlit app's interface, but the data is there for offline analysis. The `sqlite-utils` library even allows one to easily export to JSON or do analysis in Python.
- **Performance:** Logging to SQLite for each delivery in a simulation of hundreds of deliveries is reasonably fast, but it does add overhead. For extremely large simulations, one might skip logging or log only aggregate stats. In our use case, the scale is manageable, and having a persistent record is valuable for debugging and verification of results over multiple runs.

(Note: In the current Streamlit UI code, logging to the database may be turned off or simplified to keep the simulation quick. It's mentioned here because the infrastructure exists and is documented for completeness, and it could be enabled for a production or expanded version.)

Project Structure and Code Organization

As shown earlier in the project tree, the main code resides in `Dashboard.py`. For clarity:

- **Dashboard.py:** Contains everything – from UI setup to simulation logic and (optionally) AI integration hooks. It is a single-file Streamlit app. Within this file, logic is organized into functions (for generating drones, deliveries, simulation run, plotting each chart, etc.), which keeps the code modular. This also makes it easier to test individual pieces (e.g. one could call `simulate_route_metadata()` or `evaluate_delivery()` independently).
- **NYDrone delivery SIM.ipynb:** (Optional, for development) A Jupyter Notebook was used during development to prototype features such as the LangChain integration and advanced route planning. It contains experimental code (for example, the use of NetworkX to simulate actual paths and blockages, and testing the Flan-T5 chain with sample data). While not needed for running the app,

this notebook provides insight into how certain parameters or logic were tuned. It's included as a reference for developers.
- **data/** folder: Intended for storing any output data or logs. For instance, if one wanted to save each run's log as a CSV or the SQLite DB, they could appear here. In a future extension, this folder could also include static datasets (e.g., a precomputed graph of NYC routes for NetworkX, or GIS data for hubs and no-fly zones, etc.).
- **requirements.txt:** Lists all the dependencies (detailed in the Tech Stack section). If deploying to another environment, this ensures all required libraries are installed.

The code is documented with comments to explain non-obvious parts (e.g., what each plot is showing). For developers extending this project, key points of interest in the code are: - The **top section** where parameters are defined and the Streamlit UI is constructed. - The **simulation functions** (`run_simulation`, `evaluate_delivery`, etc.) which could be modified for more complex logic. - The **visualization functions** (`plot_delivery_outcome_pie`, `plot_failure_reasons`, etc.) where one could change styling or add new charts. - The **AI integration** points – currently in development, but conceptually where the random drone selection could be replaced by an AI call.

By understanding these parts, one can add features like a new chart (e.g., a line chart of cumulative deliveries over time), integrate a different AI model, or connect this simulation as a backend to an API or a larger application.

Deployment Options

NYDroneAI can be deployed in various ways to share the dashboard with others or host it as an online service. Here are the supported or recommended deployment methods:

- **Streamlit Community Cloud:** The simplest option – Streamlit's own cloud platform allows you to deploy directly from a GitHub repo for free. You would push the code to a GitHub repository, then in Streamlit Cloud, link your GitHub account and select the repository and `Dashboard.py` as the app entry point. The Streamlit Cloud will handle setting up the environment (using `requirements.txt`) and hosting the app. This method is quick and ideal for demos and small-scale use. (Note: The free tier might have limited resources; running the AI model on Streamlit Cloud is possible for the small model, but performance may be slower.)
- **Docker Container:** For more control or to deploy on other cloud providers, you can create a Docker image. A simple Dockerfile would use a Python base image, copy the project files, install requirements, and then use the command `streamlit run Dashboard.py`. Once containerized, the app can run on any platform that supports Docker (AWS, Azure, GCP, Heroku, etc.). You might need to ensure the container has enough memory for the model. Example Docker deployment:
- Build the image: `docker build -t nydroneai-dashboard .`
- Run the container: `docker run -p 8501:8501 nydroneai-dashboard`
- Then access via the server's IP on port 8501.
- **Virtual Machine or On-Prem Server:** Simply install the required packages on a VM or server (Linux or Windows), just as you would locally, and run `streamlit run Dashboard.py`. Ensure the firewall allows the chosen port (8501 by default) and access the server's URL. This approach may be suitable for an internal network or an on-premises demonstration.
- **Enterprise Deployment Considerations:** If integrating into a larger system, one could also refactor the code to separate the simulation logic and host it as an API (for example, a FastAPI app where a

POST request triggers a simulation and returns results). The Streamlit front-end could then query that API. This, however, would be a significant architectural change – by default, Streamlit apps are meant to be self-contained.

- **Scaling and Performance:** For most uses (a few hundred deliveries, small model inference), the app runs comfortably on a single machine. If you plan to scale up (say thousands of drones or using a large AI model), ensure the deployment environment has sufficient CPU/RAM, or consider switching the AI model to an endpoint (e.g., OpenAI API for GPT, which offloads computation).
- **Maintenance Mode:** The simulation is not stateful between runs unless logging to the database. If deployed long-term, the SQLite DB could grow; it might be wise to archive or clear logs periodically or incorporate a UI control to reset data. Streamlit Cloud resets on each restart, so persistent logging there might not accumulate unless paired with an external database.

In summary, deploying NYDroneAI can be as easy as a few clicks on Streamlit Cloud or as customizable as building a Docker container for your infrastructure. Choose the method that best fits your needs and audience. The key is ensuring the environment has the required libraries (as listed) and access to the model downloads.

Conclusion

NYDroneAI provides a rich environment for experimenting with drone logistics scenarios, blending simulation with AI and interactive analytics. **For users**, it offers a user-friendly tool to tweak scenarios and visually understand outcomes. **For developers**, it serves as an example of integrating simulation code with modern web apps and AI libraries, and is highly extensible – one can plug in new models or data sources to enhance realism.

Whether used for educational purposes (teaching how last-mile drone deliveries might work) or for prototyping logistics strategies in a company, this project demonstrates the power of combining data simulation with AI-driven insights. We encourage users to explore different “what-if” scenarios (e.g., what if drone failure rates drop with better hardware? what if we double the fleet size? etc.) and see the impact instantly.

The project is open-source (MIT Licensed) and contributions or customizations are welcome. With potential future enhancements like real map-based routing, larger language models for even smarter dispatch decisions, and integration with real-time data (weather, traffic), NYDroneAI could be developed into a realistic digital twin of urban drone delivery networks.

---

1 2 4 5 8 9 10 11 14 15 readme.md
file://file-DohBzKrG3eAd6goFgpm79v

3 13 54 64 65 66 67 68 69 70 71 NYDrone delivery SIM.pdf
file://file-DbGG4PuauRNPKjLFqRHFDG

6 7 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 55 56 57 58 59 60 61 62 63 Dashboard.py
file://file-QURfHvm4geJqeFjqLP2H9R

12 requirements.txt
file://file-1nn942jtyWcF6SttshHT7G