Hospital Drones
GreenCare Systems

UTEK 2T6
Group 6

Table of contents

Problem Statement

- **2 main challenges of modern healthcare:**
  1. Fast response logistics

  2. Design for Sustainability

(healthcare delivery systems account for 4.4% of GHG emissions) [1]

How can we optimize supply delivery in a sustainable manner?

[1] https://global.noharm.org/sites/default/files/documents-files/5961/HealthCaresClimateFootprint_092319.pdf

Opportunity

Medical Supply Chain Optimization using **DRONES** inside hospitals to deliver supplies using:

- Energy-efficient drones and routes
- Prioritize urgency
- Optimized sustainability

Introducing GreenCare Systems

Who Are We?

- Optimizing the path of **Matternet** drones to ensure medical personnel can get the fastest, most energy efficient delivery of healthcare equipment within the hospital.

- Use **Dijkstra** and **RRT** algorithms to determine priority deliveries among drones.

- Offering an interactive and easy-to-use UI for medical professionals

01.

Design Decisions

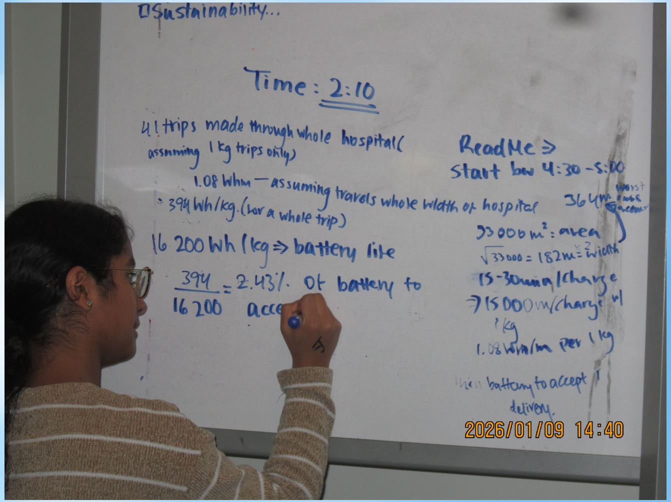
Matternet Drones [1]

- Quadcopter system (vs helicopter system)
    - Low cost
    - Smaller blades → safer
- *“speed up the transport of human specimen samples… by up to 70%” (CEO of Labour Berlin)*
- Good for short range delivery [2]
- First drone delivery system achieving standard Type C certification and Production Certification [3]
- lightweight capacity (85% of e-commerce shipments and healthcare products) [3]
- Widely implemented in healthcare for developing countries. [1]

[1] http://bulbuldelivery.com/how-ai-for-drone-technology-is-revolutionizing-delivery-routes/#:~:text=1.,conditions%20and%20wide-open%20areas.
[2]https://www.theverge.com/2024/10/3/24261066/matternet-m2-drone-delivery-service-silicon-valley-launch
[3]https://www.freightwaves.com/news/drone-disruptors-matternet-is-taking-cities-into-the-skies#:~:text=control%20the%20drones.-,We%20build%20not%20only%20the%20aircraft%20that%20flies%2C%20we%20also,and%20then%20how%20we%20land.&text=That%20sort%20of%20end%2Dto,our%20class%20in%20regulatory%20approvals.%E2%80%9D&text=Matternet%20has%20been%20operating%20beyond,drone%20network%20in%20Abu%20Dhabi
[4] https://www.businesswire.com/news/home/20221130005322/en/Matternet-Receives-FAA-Production-Certificate-for-its-M2-Drone-Delivery-System

Matternet Drones [1]

- Max payload: 2kg (4.4 lbs)
    - **P = b0 + w*b1**
        - b0 = power for drone frame
        - b1 = power for each additional kg
            - ~1.08 Wh/mkg payload
- Max speed: 57.6km/h = 16m/s
- Max range:
    - 20 km with a 1 kg payload
    - 15 km with a 2 kg payload



Used in items.py and energy.py for payload and energy (recharge, ability to do delivery) calculations

[1] https://www.sciencedirect.com/science/article/pii/S0925527319300106

Algorithm

Our system uses a **hybrid pathfinding approach** combining:

1. **Dijkstra's Algorithm:** used by Uber. Energy efficient. [1]

2. **Modified RRT Algorithm:** path-planning algorithm while avoiding obstacles. Good for emergency and high priority drones during intersections. [2]

[1] https://ioaglobal.org/blog/how-uber-utilises-data-science/#:~:text=Route%20optimisation%20is%20crucial%20for,and%20speed%20of%20the%20service.
[2] https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

Where do we fit?

	Dijkstra	RRT	Combined RRT + Dijkstra
Purpose	Closest path	Path planning w/ collision avoidance	Optimal routing with dynamic obstacle avoidance
Used in	Optimal drone assignment based on shortest path distance	Navigate around obstacles and other drones in real-time	1. Dijkstra **finds** the closest drone and initial optimal path 2. RRT **refines** the path dynamically, avoiding collisions and other drones
Innovation	Graph-based routing through hospital hallways, not just Euclidean distance	Extended with 3-lane traffic system and priority-based yielding	Combines graph-based optimality (Dijkstra) with free-space flexibility (RRT)

Location: `graph.py` - `find_closest_drone_location()`

```python
def find_closest_drone_location(self, requester_location_id: int,
                                drone_locations: List[int]) -> Optional[int]:
    """Find closest drone using Dijkstra's shortest path (not Euclidean
    distance)"""

    # Use Dijkstra to find shortest paths from requester location
    distances, _ = self.weighted_dijkstra(requester_location_id)

    # Find minimum distance among available drone locations
    closest_id = None
    min_distance = float('inf')

    for drone_loc_id in drone_locations:
        if drone_loc_id in distances and distances[drone_loc_id] <
        min_distance:
            min_distance = distances[drone_loc_id]
            closest_id = drone_loc_id

    return closest_id
```

Location: `rrt_pathfinding.py` - `_is_collision_free()`

```python
```python
def _is_collision_free(self, point, other_drones, current_drone_id,
                       is_emergency, current_lane, current_priority_level):
    """Check collision with 3-lane system and priority-based yielding"""

    for drone_id, trajectory in other_drones.items():
        other_lane = getattr(other_pos, 'lane', 1)
        other_priority = getattr(other_pos, 'priority_level', 3)

        # Same lane: stricter collision check
        if current_lane == other_lane:
            dist = self._distance(point, predicted_pos)
            if dist < self.lane_width * 1.5:
                # Lower priority must yield to higher priority
                if (not is_emergency and current_priority_level < 4) and \
                   (other_is_emergency or other_priority >= 4):
                    return False  # Lower priority must yield

        # Emergency vehicles get 3x safety margin
        if other_is_emergency and not is_emergency:
            emergency_safety_radius = self.obstacle_radius * 3.0
            if dist < emergency_safety_radius:
                return False

    return True
```
```

Dijkstra [1]

3-lane system!

RRT [2]

[1] https://www.codecademy.com/article/dijkstras-shortest-path-algorithm
[2] https://www.cs.cmu.edu/~motionplanning/lecture/lec20.pdf

Location: `service.py` - `_assign_drone_to_request()`

```python
def _assign_drone_to_request(self, request: Request) -> bool:
    """Assign closest available drone to request using RRT path planning"""
    is_emergency = request.emergency or request.priority.is_emergency
    available_locations = self._get_available_drone_locations(for_emergency=is_emergency)
    if not available_locations:
        return False
    # STEP 1: Use Dijkstra to find closest drone location
    closest_loc_id = self.graph.find_closest_drone_location(
        request.requester_location_id, available_locations
    )
    if closest_loc_id is None:
        return False
    # Find the drone at that location
    assigned_drone = None
    for drone in self.drones.values():
        if (drone.status == "available" and
            drone.current_location_id == closest_loc_id and
            drone.emergency_drone == is_emergency):
            assigned_drone = drone
            break
    # STEP 2: Use RRT to plan path with collision avoidance
    start_loc = self.graph.nodes[closest_loc_id]
    goal_loc = self.graph.nodes[request.requester_location_id]
    path = self.rrt_planner.plan_path_with_traffic_rules(
        start_loc=start_loc,
        goal_loc=goal_loc,
        current_drone_id=assigned_drone.id,
        is_emergency=is_emergency,
        active_drone_flights=self.active_flights,
        all_drones=self.drones,
        current_priority_level=request.priority.value
    )
    # STEP 3: Fallback to Dijkstra if RRT fails
    if len(path) < 2:
        path, _ = self.graph.find_shortest_path(closest_loc_id, request.requester_location_id)

    # Assign the route
    assigned_drone.delivery_route = path
    request.assigned_drone_id = assigned_drone.id
    return True
```

Combined

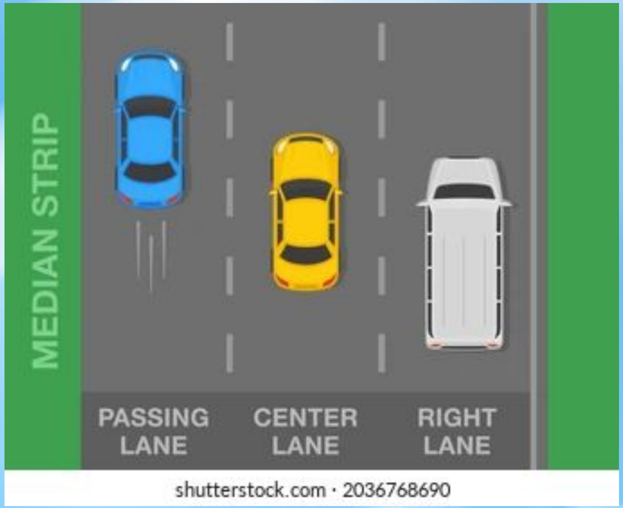
Location: `rrt_pathfinding.py` - `_is_collision_free_with_lanes()`

```python
def _is_collision_free_with_lanes(self, from_point, to_point,
                                  active_drone_flights, all_drones,
                                  current_drone_id, lane_offset):
    """
    Check if path segment is collision-free considering 3-lane system
    """
    # Calculate lane position (left: -1, middle: 0, right: +1)
    current_lane = lane_offset

    for drone_id, flight_info in active_drone_flights.items():
        if drone_id == current_drone_id:
            continue

        other_drone = all_drones.get(drone_id)
        if not other_drone:
            continue

        # Check priority - lower priority must yield
        other_priority = flight_info.get('priority_level', 3)
        current_priority = ...  # current drone's priority

        if current_priority < other_priority:
            # Lower priority: must yield if in same lane
            if self._same_lane_conflict(current_lane, other_drone, from_point, to_point):
                return False
        elif current_priority > other_priority:
            # Higher priority: can use any lane
            pass

    return True
```

3-lane

Innovation: 2-3 lane system

- Each hallway supports 3 drones side-by-side (3+m width total)
- Priority-based lane assignment:
- Emergency/high-priority drones → middle lane
- Normal/low-priority drones → left/right lanes
- Lower-priority drones yield to higher-priority ones



What Makes us Innovative?

03.

Strengths

- Medical professionals can focus on tasks
- Time efficient (average **45 seconds** to use program) compared to walking
- **Dijkstra and RRT** algorithm = accurate and energy efficiency
- **Multiple trips** enabled for sustainability
- Uses **Triage** system, used in hospitals [2]

Weakness

- Don't work **both inside and outside hospitals**
- See future steps!

Opportunities

- **UAV** for emergency drones = efficient
  - Drones go to closest charging station instead of back to storage
- Allow **patients** to use the app as well
- Assess **priority** of drone delivery
- Implementing **sanitation** area drones

Threats

- Making sure r**outes avoid areas** near MRI scans or high danger areas for certain time periods.
- Other algorithms
- Made of carbon fibre + composites [1] (tradeoff with being lightweight)
- Untested **noise level.** No parachute features

[1] https://www.designlife-cycle.com/drones#:~:text=The%20use%20of%20carbon%20fiber,internal%20components%20are%20working%20properly
[2] https://pub-haldimandcounty.escribemeetings.com/filestream.ashx?DocumentId=3293#:~:text=The%20Canadian%20Triage%20and%20Acuity,figure%201%20for%20further%20details

Competitor Analysis

	Zipline [1, 2]	GreenCare Systems	Delivery Drone Canada [3, 4]
Emergency Vehicle Response System + Customization	✗	✓	✗
Path Planning + Optimization Algorithms	✓	✓	✓
App controlling what is delivered	✓	✓	✗
Priority Queue + Speed Adjustments	✗	✓	✗

[1]https://dronedeliverycanada.com/technology/advanced-drone-technology-the-canary-rpa/#:~:text=In%20addition%2C%20the%20Canary%20is,commercial%20route%20for%20the%20Canary.

[2]https://dronedeliverycanada.com/technology/advanced-drone-technology-the-canary-rpa/#:~:text=In%20addition%2C%20the%20Canary%20is,commercial%20route%20for%20the%20Canary.

[3] https://www.zipline.com/about/zipline-fact-sheet

[4] https://www.zipline.com/technology

Key Features

Drone Charging

Drones will be charged with solar energy

- Emergency charging stations powered by battery

Item Assortment

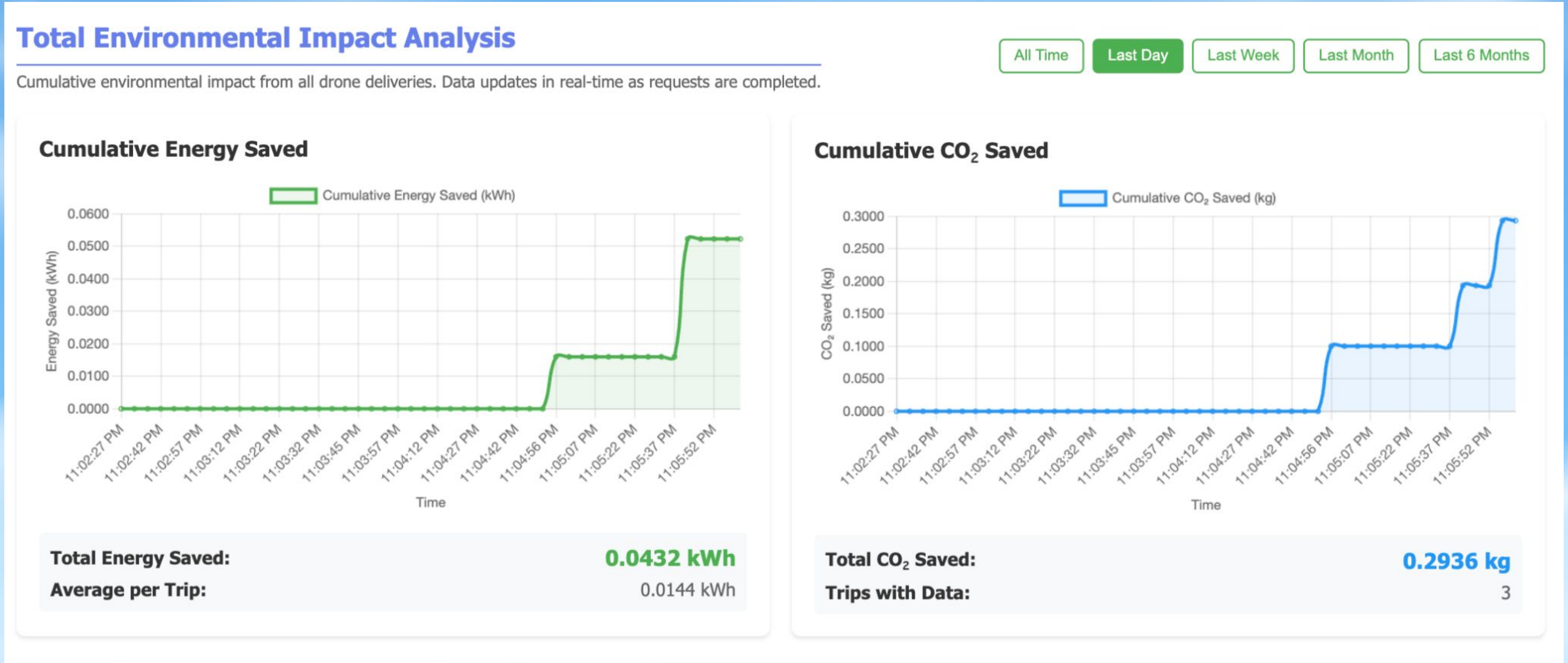
If one drone cannot get everything → automatically sends two drones

- Prioritizes important items

Manual Override

Requesters are able to manually mark a drone as “Force Emergency Flag” → allow drones to travel quicker

Some Sample Outputs



*Calculated from Matternet data shown previously, in UI, from 4 requests alone!

Sample Drone outputs

Request #3 - J W Davis CTAS III - Urgent

(No description provided)

Payload: 1x Gauze Pack
Weight: 0.060 kg / 2 kg max Location: 7 | Completed: 1/11/2026, 9:49:41 AM
Drone ID: 4

View Path

Energy Savings Report

Distance Traveled:	**0.124 km (124 m)**	Drone Energy:	**0.1413 kWh**
Traditional Energy:	**0.1372 kWh**	Energy Saved:	**-0.0041 kWh**
Savings %:	**-3.01%**		
CO_2 Emissions Saved:			**0.1072 kg**

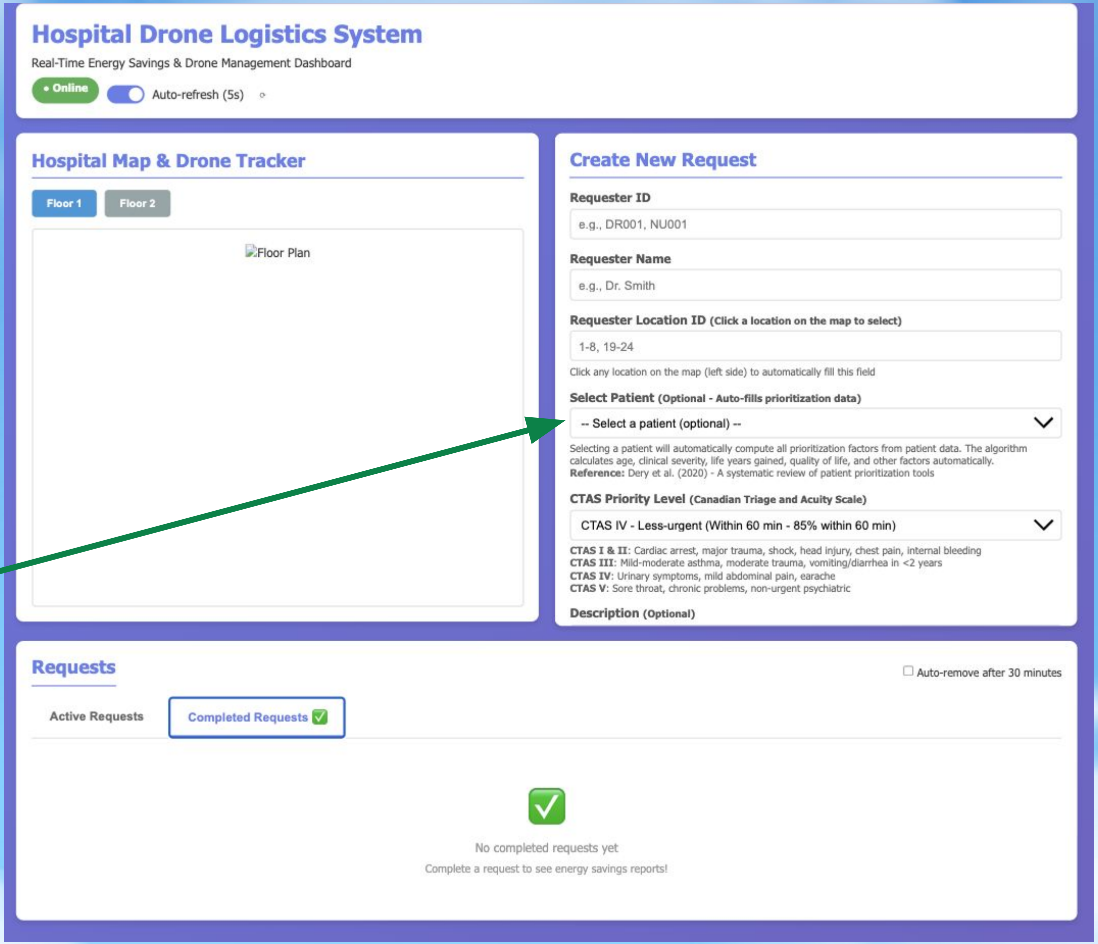
Time Comparison (vs Walking at 3 mph / 4.828 km/h)

Walking Time:	**1.54 min (92.5 sec)**	Drone Time:	**0.83 min (49.6 sec)**	Time Saved:	**0.71 min (42.9 sec)**	Speed Improvement:	**1.86x faster**	Time Savings:	**46.4%**

UI Design

Section Placement

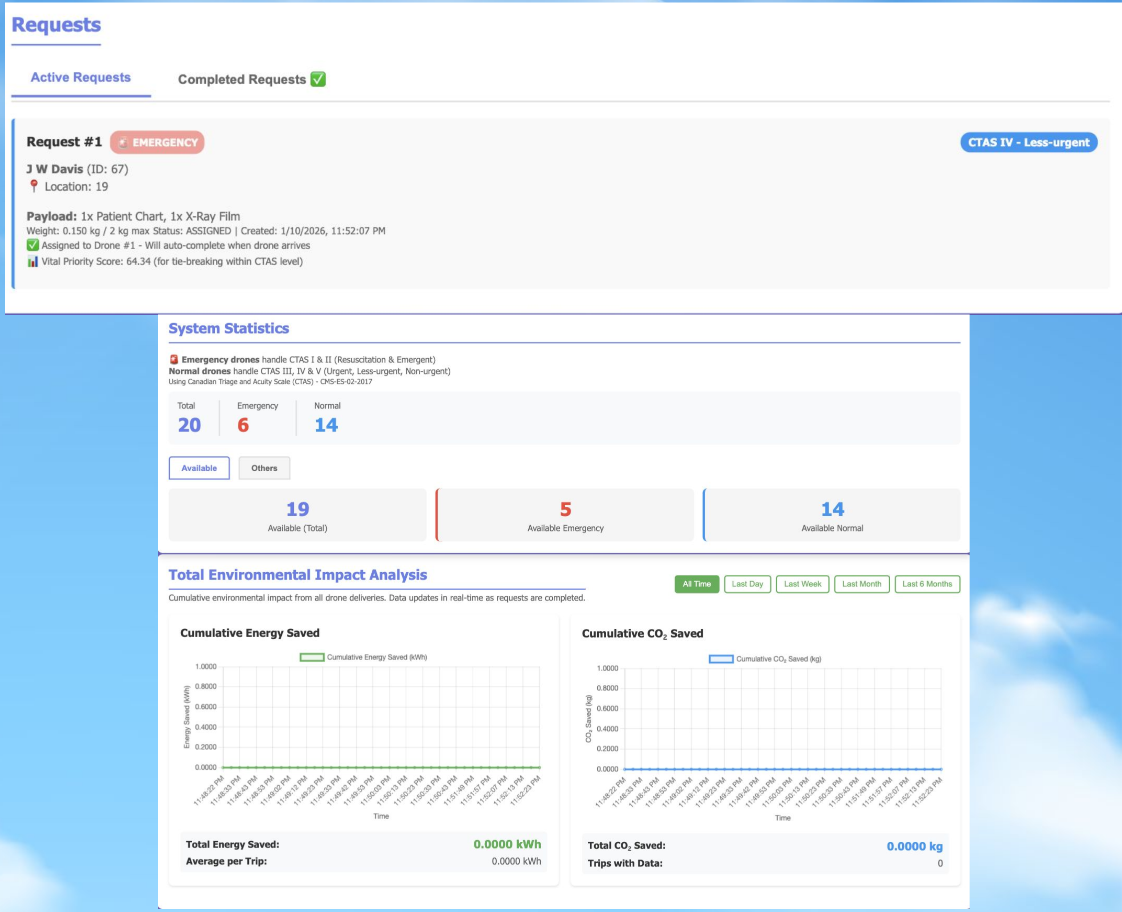
- Sections are organized in terms of relevance



UI Design

User Friendliness

- Shows status of Request, moves to completed, can choose to show completed for past 30 mins or not
- Shows drone statuses

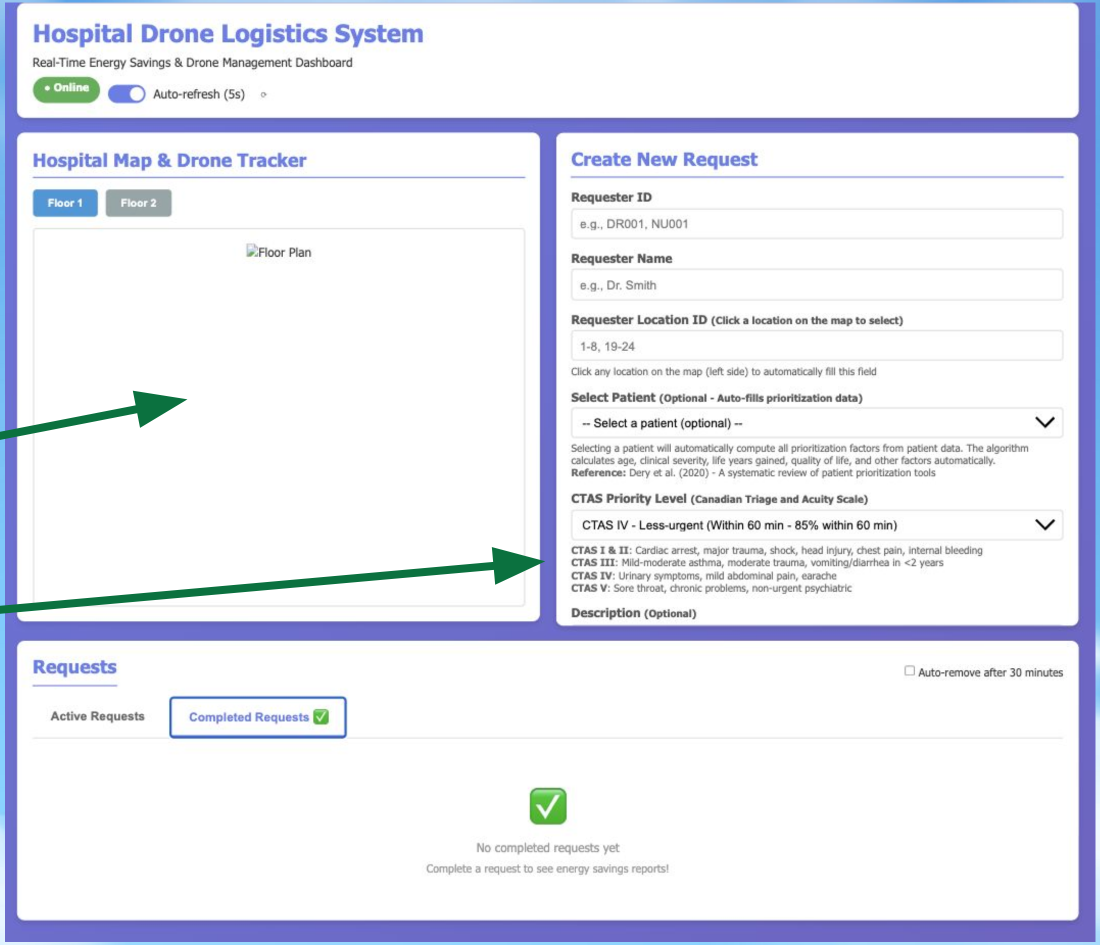


UI Design

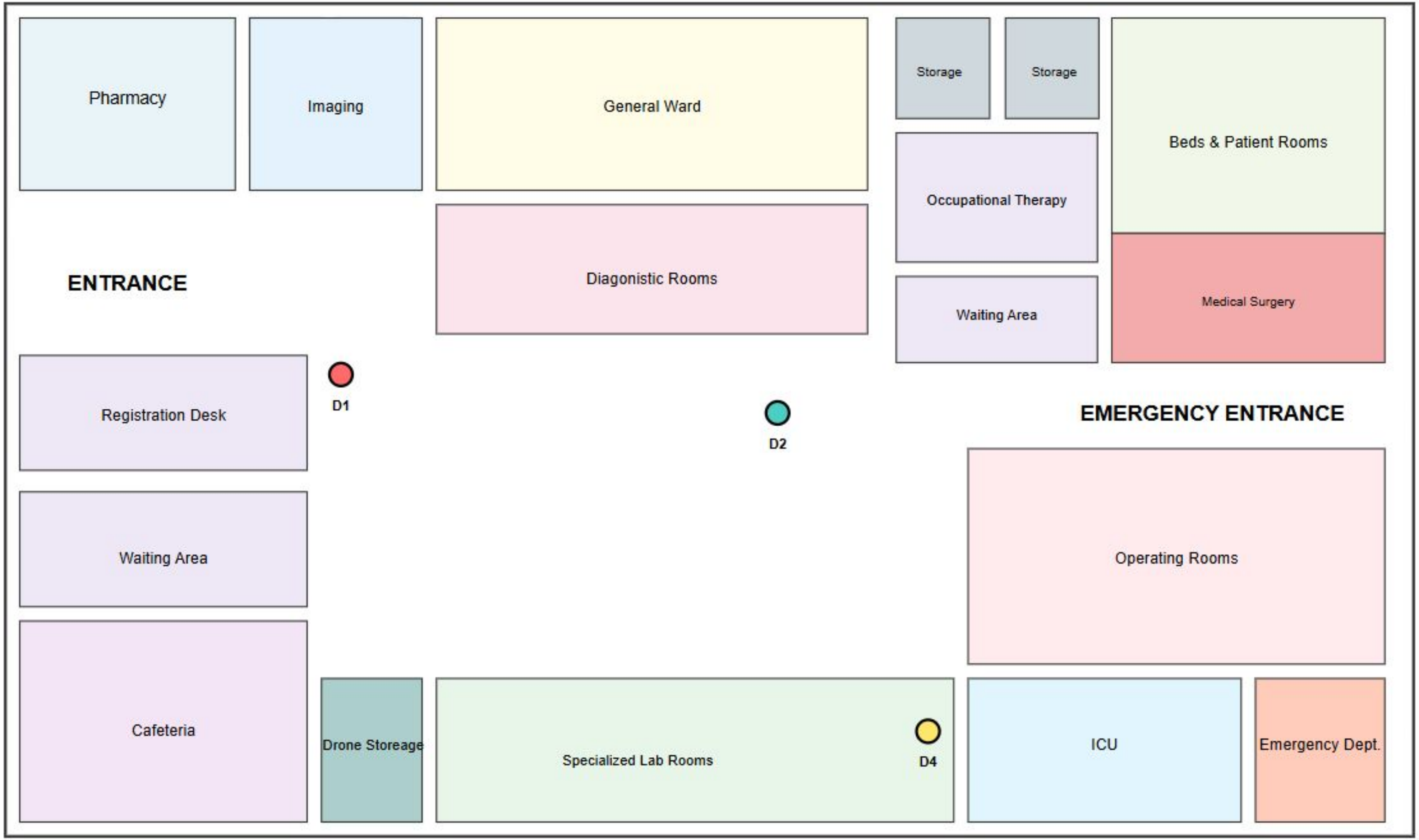
Interactive Map

- Click on sections on map to selection location for delivery
- Clickable drones to see where drone is delivering to

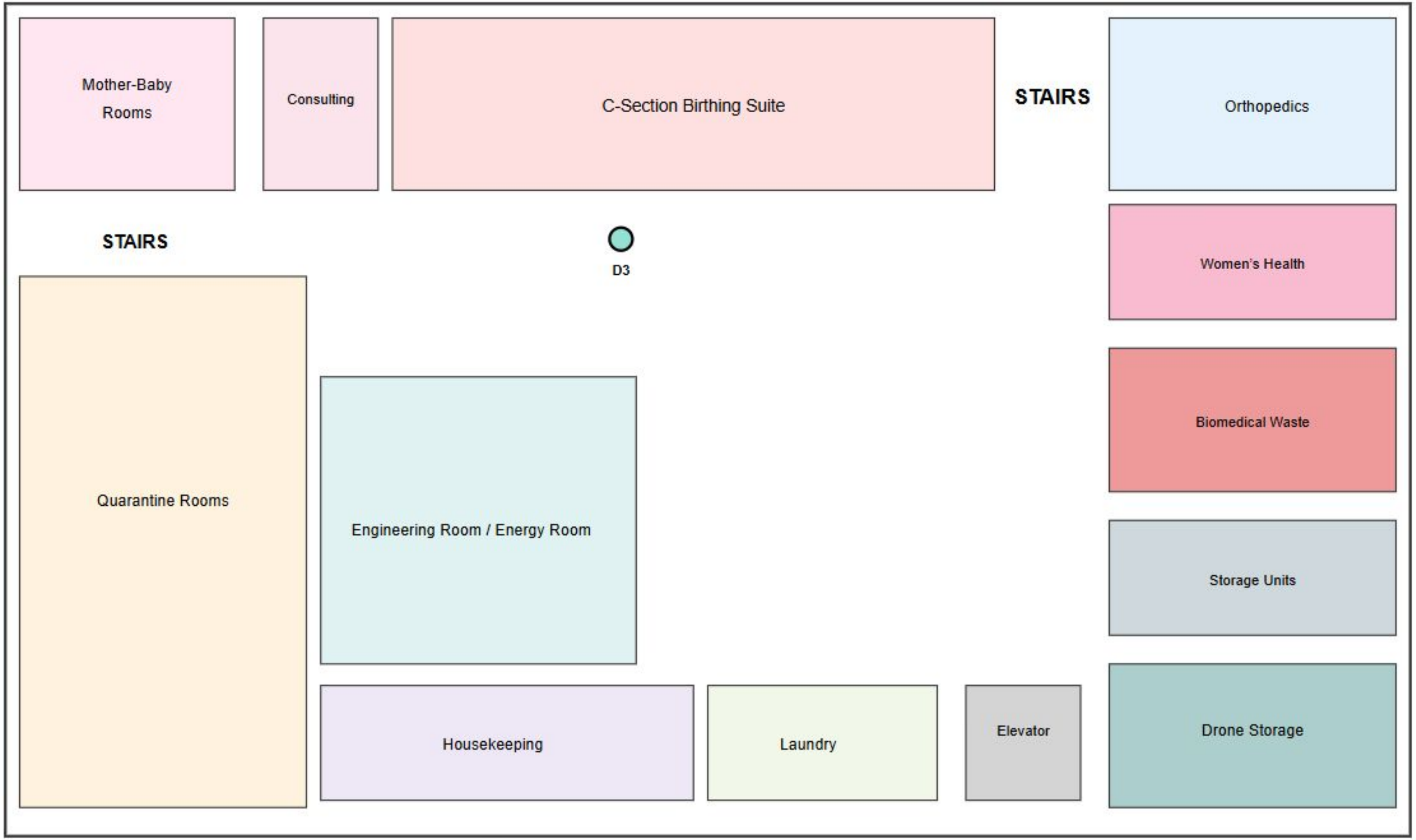
- Provide a priority rank for getting your request fulfilled



Floor 1 - Main Hospital Floor



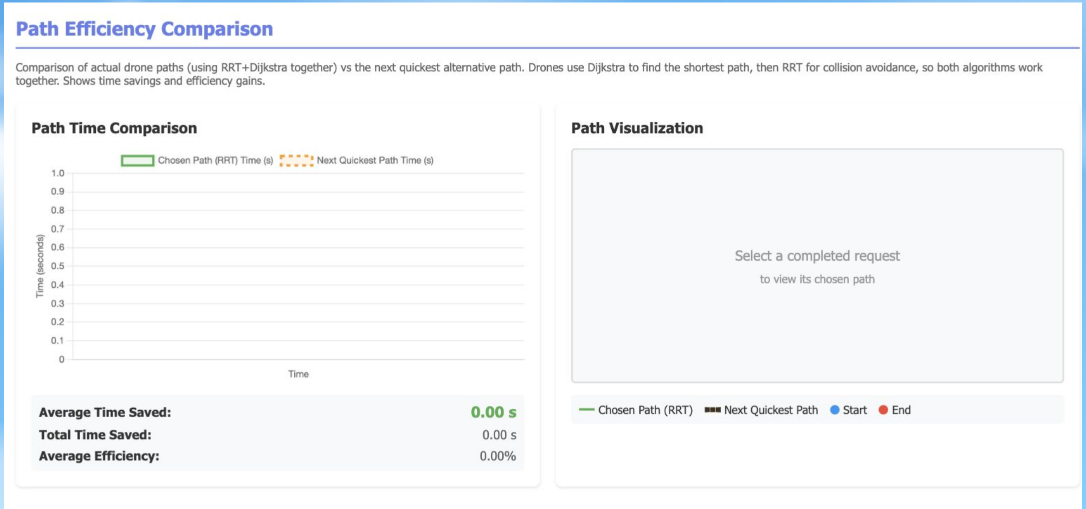
Floor 2 - Maternity & Specialized Care

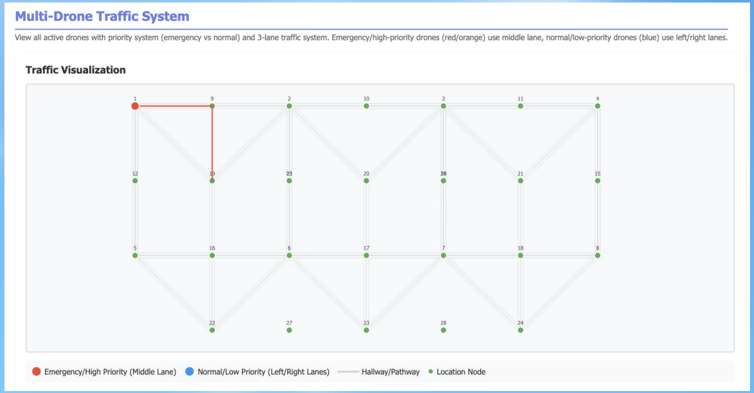


Next Steps

03.

Extra Features That Need Work





Testing BLE or RFID so drones relay task priority and implement the path optimization algorithms.

Implementing UAV so multiple optimally placed charging ports allow for drones to dock at closest charging station.

Include sanitation zones for drones/areas only certain drones are able to access via fobs (activated/deactivated based on location+routes they use)

Alerts section: indicate if any issues arose with a drone + automatically send out a replacement drone

Mobile App - Patients

Accessibility

- Sign in with personal ID and number
  - Require ID verification
- Once registered, app connects to database containing all key information

Key features

- Order food
- Request basic items (bandaids, water)

Integrate LLM

- Take description and items selected → rank priority within the CTAS levels

Thank You!

Do you have any questions?

Files

Frontend

- index.html
- map.css

Backend

- main.py - entry point, initialization, and example usage
- api.py - Flask REST API server for frontend integration
- rrt_pathfinding.py - RRT* (rapidly-exploring random tree) algorithm integration for collisions and drone paths
- graph.py - hospital graph with weighted Dijkstra implementation
- models.py - data structures (location, drone, request, priority, patient)
- energy.py - energy calculations, track energy + CO2 savings
- service.py - drone assignment with priority queue and RRT path planning implementation
- items.py - item catalog and payload management
- patients.py - patient database and vitals management
- map.js - three.js-based 3D/2D hospital map (SVG for 2D)

models.py

Defines all the core data structures (classes) used throughout the system.

Key feature: CTAS (Canadian Triage and Acuity Scale) - medical priority system

- **Request Class:** Represents a delivery request from hospital staff.

```
`id`, `requester_id`, `requester_name`
`requester_location_id` - Where the request is coming from
`priority` - CTAS priority level
`emergency` - Boolean flag
`status` - PENDING, ASSIGNED, IN_TRANSIT, COMPLETED, CANCELLED
`assigned_drone_id` - Which drone is handling this
`energy_saved_kwh`, `co2_saved_kg` - Energy metrics (after completion)
```

Key feature: Tracks full lifecycle from creation to completion with energy savings

- **Drone class:** Represents a physical drone in the system.
- **Two types of drones**
- **Location Class**
- **State machine for request lifecycle**
- **Patient Class:** Vital Priority System - automatically calculates priority based on patient condition

```
- `id`, `current_location_id` - Where the drone is
- `status` - available, assigned, in_transit, charging
- `emergency_drone` - Boolean (emergency vs normal drone)
- `battery_level_kwh` - Current battery
- `delivery_route` - List of location IDs for current route
- `current_speed_m_per_sec` - Speed based on priority
```

service.py

The heart of the system - manages requests, drone assignments, routing, and energy calculations.

Main feature: DroneAssignmentService

Feature: Priority Queue System

- Uses min-heap (Python's `heapq`)
- Lower priority value = higher priority (CTAS I = 1 is most urgent)
- Automatically processes highest priority requests first
- Ensures emergency requests are always handled first

```
def __init__(self, hospital_graph: HospitalGraph):
    self.graph = hospital_graph  # Hospital layout
    self.requests: Dict[int, Request] = {}  # All requests
    self.drones: Dict[int, Drone] = {}  # All drones
    self.priority_queue = []  # Min-heap for priority queue
    self.active_flights: Dict[int, dict] = {}  # Active drone flights
    self.rrt_planner = RRTPathPlanner(...)  # RRT path planner
```

service.py

The heart of the system - manages requests, drone assignments, routing, and energy calculations.

Main feature: Combines Dijkstra (optimal assignment) + RRT (dynamic avoidance)

```
def _assign_drone_to_request(self, request: Request) -> bool:
    # STEP 1: Use Dijkstra to find closest drone location
    closest_loc_id = self.graph.find_closest_drone_location(
        request.requester_location_id, available_locations
    )

    # STEP 2: Use RRT to plan path with collision avoidance
    path = self.rrt_planner.plan_path_with_traffic_rules(...)

    # STEP 3: Fallback to Dijkstra if RRT fails
    if len(path) < 2:
        path, _ = self.graph.find_shortest_path(...)
```

Feature: Full lifecycle tracking with automatic assignment

- Energy cycles calc
- Drones can intercept additional requests while in flight
- Tracks battery levels for each drone
- Compares actual path (RRT+Dijkstra) vs next quickest path
- Priority-based speed

```
create_request() → _assign_drone_to_request() →
update_drone_positions() → complete_request()
```

service.py

```
1. **`create_request()`** - Creates new request, automatically assigns
drone
2. **`_assign_drone_to_request()`** - Core assignment logic (Dijkstra +
RRT)
3. **`complete_request()`** - Marks request complete, calculates energy
savings
4. **`get_request_status()`** - Returns current request status
5. **`get_statistics()`** - System-wide stats (total energy saved, etc.)
6. **`update_drone_positions()`** - Updates drone locations during flight
7. **`_check_and_intercept_request()`** - Multi-stop optimization
8. **`get_energy_report()`** - Detailed energy savings report
```

api.py

Flask REST API that connects frontend to backend service layer.

```
app = Flask(__name__)
app.run(host='0.0.0.0', port=5001, debug=True)
```
```
- Serves web dashboard at `http://localhost:5001/`
- REST API endpoints at `/api/*`
```

```
POST /api/initialize
```

- Creates hospital graph with all locations
- Initializes 20 drones (6 emergency, 14 normal)
- Sets up charging stations

energy.py

Main feature: Energy required to take a trip for Data and To take a trip

Feature: Determining the amount of energy based on the payload provided by the drone.

- Compared the energy and the average time taken with walking, using electric carts and traditional vehicles for data.

- Determines the CO_2 saved (in kg)

```python
# Distance-based energy calculation
# Base consumption: 1.08 Wh/m = 0.00108 kWh/m with 1 kg payload
# Energy scales with payload: with 0 kg: ~0.9x, with 1 kg: 1.0x, wi
# (Inferred from range: 20 km with 1 kg -> 15 km with 2 kg = 1.33x
if payload_weight_kg <= 0:
    # No payload - slightly less energy
    payload_multiplier = 0.9
elif payload_weight_kg <= 1.0:
    # Linear scaling from 0.9x to 1.0x for 0-1 kg
    payload_multiplier = 0.9 + (payload_weight_kg / 1.0) * 0.1
else:
    # Non-linear scaling from 1.0x to 1.33x for 1-2 kg
    # Range decreases from 20 km to 15 km (ratio = 1.33)
    extra_weight = payload_weight_kg - 1.0
    payload_multiplier = 1.0 + (extra_weight / 1.0) * 0.33
```

items.py

Key Feature: Figuring out the amount of drones needed based on payload for the items that are requested.

Catalogue of most common items request by medical professional with weight for ordering

```
@classmethod
def split_payload(cls, item_quantities: Dict[str, int], patient_critical: bool = False) -> List[D
    """
    payload into multiple requests if it exceeds capacity
    priotizie items based on patient condition - most critical items go first
    Args:
        item_quantities: Dictionary mapping item_id to quantity
        patient_critical: True if patient is in critical condition
    Returns:list of item_quantities dictionaries, each representing one drone load, prioritized
    """
    if not item_quantities:
        return []
    # calc total weight
    total_weight = cls.calculate_total_weight(item_quantities)
    if total_weight <= cls.MAX_PAYLOAD_CAPACITY_KG:
```

rrt_pathfinding.py

- It picks a random path its starting point
- Finds the Euclidean distance to get the trajectory and determines if there are obstacles like drones in the way.
- Calculates the cost of the trajectory to get the

minimum cost

- If RRT takes too long, Dijkstra's algorithm will

be used instead.

```
#  nearest node
nearest_point, nearest_loc_id = self._nearest_node(tree, rand_point)
#  towards random point
new_point = self._steer(nearest_point, rand_point, step_size)
#  if collision-free (with enhanced emergency vehicle handling)
#  current speed from trajectory estimation if available
current_speed = 2.5  # def speed
if other_drones:
    # est average speed from other drones for relative speed calculation
    for traj in other_drones.values():
        if len(traj) > 1:
            avg_speed = traj[0].speed if traj else 2.5
            break
if not self._is_collision_free(
    new_point, other_drones, current_drone_id, is_emergency,
    timestamp=i * 0.1, current_speed=current_speed
):
    continue
#  nearby nodes for rewiring (RRT*)
near_nodes = self._near_nodes(tree, new_point, step_size * 2.0)
#  best parent (RRT* optimization)
```

```python
def plan_path_with_traffic_rules(
    self,start_loc: Location, goal_loc: Location,
    current_drone_id: int, is_emergency: bool,active_drone_flights: Dict[int, dict], all_drones: Dict[int, 'Drone']  # type: ignore
) -> List[int]:
```

```python
other_drone_positions: Dict[int, List[DronePosition]] = {}
for drone_id, flight_info in active_drone_flights.items():
    if drone_id == current_drone_id:
        continue
    drone = all_drones.get(drone_id)
    if not drone or drone.status not in ["assigned", "in_transit"]:
        continue
```

```python
# estimate positions along route
positions = []
current_time = 0.0
for i, loc_id in enumerate(route):
    if loc_id in self.graph.nodes:
        loc = self.graph.nodes[loc_id]
        speed = drone.current_speed_m_per_sec if drone else 2.5
        # est time to reach this location
        if i > 0:
            prev_loc = self.graph.nodes[route[i-1]]
            dist = self.graph.euclidean_distance(prev_loc, loc)
            current_time += dist / speed
        is_emerg = drone.emergency_drone if drone else False
        positions.append(DronePosition(
            drone_id=drone_id,
            location_id=loc_id,
            x=loc.x,
            y=loc.y,
            z=0.0,
            timestamp=current_time,
            is_emergency=is_emerg,
            speed=speed
        ))
```

```python
#  RRT to plan collision-free path
path = self.plan_path_with_avoidance(
    start_loc=start_loc,
    goal_loc=goal_loc,
    current_drone_id=current_drone_id,
    is_emergency=is_emergency,
    other_drones=other_drone_positions,
    max_iterations=300 if is_emergency else 500  # emergency drones get faster planning
)
if path is None or len(path) < 2:
    #  to simple shortest path
    path, _ = self.graph.find_shortest_path(start_loc.id, goal_loc.id)
return path
```

graph.py

Represents hospital layout as a data structure and models the hospital as a graph

- Nodes = locations (rooms, charging stations)
- Edges = pathways (hallways)
- Weights = travel time/distance between times

Implemented Dijkstra for shortest path finding (closest drone based on path distance)

```python
def weighted_dijkstra(self, start_id: int, target_id: Optional[int] = None) -> Tuple[Dict[int, float], Dict[int, int]]:
    if start_id not in self.nodes:
        raise ValueError(f"Start location {start_id} not in graph")

    # Initialize distances: all nodes initially unreachable (infinity)
    distances: Dict[int, float] = {node_id: float('inf') for node_id in self.nodes}
    distances[start_id] = 0.0

    # Track previous node for path reconstruction
    previous: Dict[int, int] = {}

    # Priority queue: (distance, node_id)
    pq = [(0.0, start_id)]
    visited = set()

    while pq:
        current_dist, current_id = heapq.heappop(pq)

        if current_id in visited:
            continue

        visited.add(current_id)

        # Early termination if target reached
        if target_id is not None and current_id == target_id:
            break

        # Explore neighbors
        if current_id in self.adjacency_list:
            for neighbor_id, edge_weight in self.adjacency_list[current_id]:
                if neighbor_id in visited:
                    continue
                # Calculate total distance: edge weight + heuristic adjustment
                new_dist = current_dist + edge_weight
                # If found a shorter path, update it
                if new_dist < distances[neighbor_id]:
                    distances[neighbor_id] = new_dist
                    previous[neighbor_id] = current_id
                    heapq.heappush(pq, (new_dist, neighbor_id))

    return distances, previous
```

main.py

Initialized locations of rooms, charging stations, sample paths

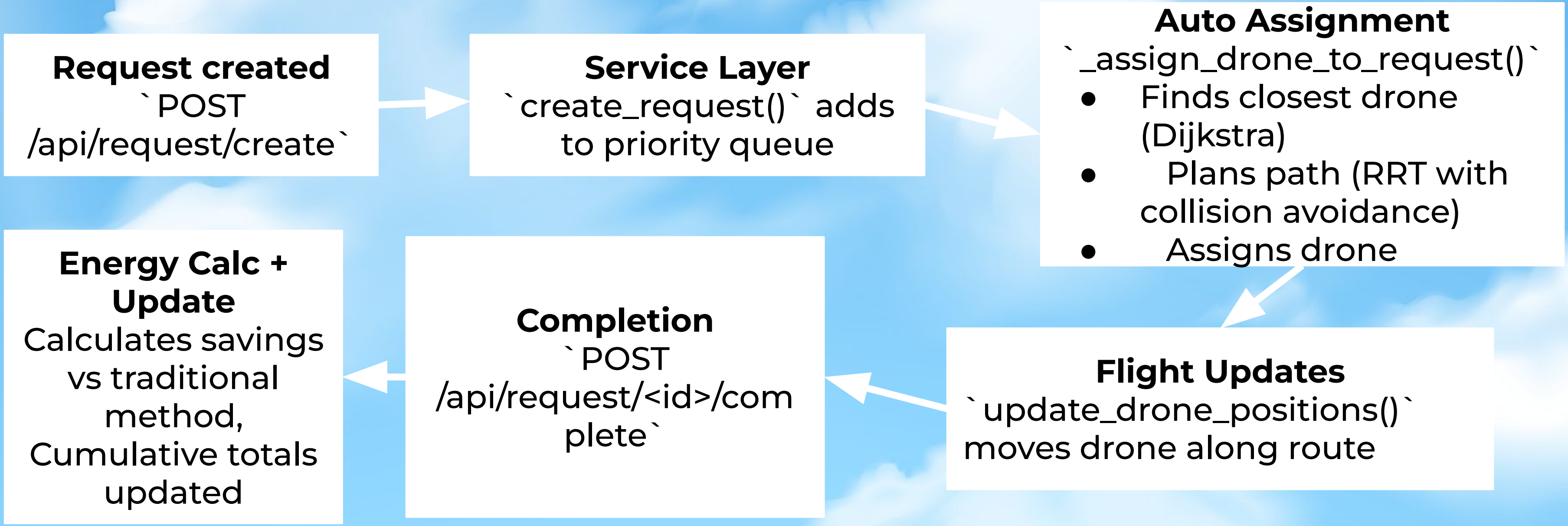
Generating the paths for drones (charging pathway, hospital pathway)

```python
locations = [
    Location(1, "Emergency Room", 0, 0, 1),
    Location(2, "ICU", 62, 0, 1),
    Location(3, "Pharmacy", 124, 0, 1),
    Location(4, "Lab", 186, 0, 1),
    Location(5, "Cafeteria", 0, 60, 1),
    Location(6, "Ward A", 62, 60, 1),
    Location(7, "Ward B", 124, 60, 1),
    Location(8, "Surgery", 186, 60, 1),
    # Additional locations for more complex routing
    Location(19, "Radiology", 31, 30, 1),
    Location(20, "Physical Therapy", 93, 30, 1),
    Location(21, "Cardiology", 155, 30, 1),
    Location(22, "Oncology", 31, 90, 1),
    Location(23, "Orthopedics", 93, 90, 1),
    Location(24, "Neurology", 155, 90, 1),
]
```

```python
# emergency drones: Half at leftmost node, half at rightmost node
emergency_drone_count = 6
for i in range(emergency_drone_count):
    # First half at leftmost, second half at rightmost
    if i < emergency_drone_count // 2:
        start_location_id = leftmost_location_id
    else:
        start_location_id = rightmost_location_id
    service.add_drone(start_location_id, emergency_drone=True)
    # mark drone as available
    drone = service.drones[service.next_drone_id - 1]
    drone.status = "available"
    drone.is_charging = False  # Not at charging station, just available
    drone.battery_level_kwh = drone.battery_capacity_kwh * 0.8  # Start at 80% charge
# normal drones: Half at leftmost node, half at rightmost node
normal_drone_count = 14
for i in range(normal_drone_count):
    # First half at leftmost, second half at rightmost
    if i < normal_drone_count // 2:
        start_location_id = leftmost_location_id
    else:
        start_location_id = rightmost_location_id
    service.add_drone(start_location_id, emergency_drone=False)
    # mark drone as available
    drone = service.drones[service.next_drone_id - 1]
    drone.status = "available"
    drone.is_charging = False  # Not at charging station, just available
    drone.battery_level_kwh = drone.battery_capacity_kwh * 0.8  # Start at 80% charge
# total: 6 emergency drones + 14 normal drones = 20 drones
return service
```

System flow chart

Models.py: Data structures (Request, Drone, Location, Priority)
Service.py: Business logic (assignment, routing, energy, battery)
API.py: REST endpoints (create request, get status, statistics)



Q&A

Q: How do you ensure emergency requests are handled first?
A: We use a **min-heap priority queue** where lower priority values (CTAS I = 1) are processed first. The queue is sorted by priority value, then by waiting time.

Q&A

Q: How do you find the closest drone?
A: We use **Dijkstra's** algorithm on the hospital graph to find the shortest path distance, not just **Euclidean** distance. This accounts for hallways and pathways.

Q&A

Q: What happens if RRT pathfinding fails?
A: We fall back to Dijkstra's shortest path algorithm. This ensures we always have a valid route.

Q&A

Q: How do you calculate energy savings?
A: We calculate drone energy consumption based on **distance** and **payload** weight, then compare against traditional methods (vehicle, electric cart, walking) using industry-standard formulas.

Q&A

Q: Can drones handle multiple requests?
A: Yes! We have multi-stop optimization. When a drone is in flight, we evaluate if accepting a second request is energy-efficient. If it saves energy or is within 2.53% of baseline, the drone intercepts the new request.

Q&A

Q: How do you prevent collisions?
A: RRT pathfinding with **3-lane traffic system.** Emergency drones get middle lane, normal drones use left/right lanes. Lower priority drones yield to higher priority ones.

Q&A

Q: What's the difference between emergency and normal drones?
A: Emergency drones are faster **(4 m/s vs 2.5 m/s),** can only handle emergency requests, and **get priority** in the 3-lane system (middle lane).

Q&A

Q: How do you track battery?
A: Each drone has a battery level. When it drops below threshold, we automatically send it to the nearest charging station before it can accept new requests.

Q&A

Q: What data does the frontend get?
A: Everything! Request status, drone positions, routes, energy savings, statistics, graph structure for visualization. The API provides full system state.

Q&A

Q: How do you handle concurrent requests?
A: Thread-safe implementation using Python's threading.Lock() to prevent race conditions when multiple API requests modify the same data.