TASK – 3

To solve this problem, I will use the A* algorithm which is a widely used pathfinding algorithm. It is an efficient algorithm and is used in games and robotics to find the shortest path between two points.

Here's my approach to solve the problem:

Parse the input list to create a graph. Each drone's starting and ending positions will be nodes in the graph. We will create edges between nodes that are adjacent and can be traversed by the drone.

Implement the A* algorithm to find the shortest path between the starting and ending positions of each drone.

To handle dynamic inputs, we can create an API endpoint that accepts the input list in the JSON format. The algorithm will be executed on the server, and the output can be returned in JSON format.

For the simulation, we can use a graphical representation of the 2D grid, where each drone is represented by a different color. As the drones move along their path, their positions can be updated on the grid to show their progress.

Libraries/frameworks that can be used are Flask or Django for creating the API endpoint and matplotlib for the graphical representation of the grid.

Here is the implementation in Python -

```
import heapq
import math

# Define a class for nodes in the graph
class Node:
    def _init_(self, x, y):
        self.x = x
        self.y = y
        self.g = math.inf  # cost of the path from the start node to this node
        self.h = math.inf  # estimated cost of the path from this node to the target node
        self.f = math.inf  # sum of g and h
        self.parent = None  # parent node in the path

    def _lt_(self, other):
        return self.f < other.f

# Define the A* algorithm
def astar(graph, start, end):
    open_list = []  # list of nodes to be considered
    closed_set = set()  # set of nodes already evaluated

    start_node = Node(start[0], start[1])
    end_node = Node(end[0], end[1])
    start_node.g = 0
```

```
start_node.h = heuristic(start_node, end_node)
start_node.f = start_node.g + start_node.h
heapq.heappush(open_list, start_node)

while open_list:
    current_node = heapq.heappop(open_list)

    if current_node.x == end_node.x and current_node.y == end_node.y:
        # found the target node
        path = []
        while current_node:
            path.append((current_node.x, current_node.y))
            current_node = current_node.parent
        return path[::-1]  # return reversed path

    closed_set.add((current_node.x, current_node.y))

    for neighbor in get_neighbors(graph, current_node):
        if (neighbor.x, neighbor.y) in closed_set:
            continue

        tentative_g = current_node.g + distance(current_node, neighbor)
        if tentative_g < neighbor.g:
            neighbor.g = tentative_g
            neighbor.h = heuristic(neighbor, end_node)
```

```
            neighbor.f = neighbor.g + neighbor.h
            neighbor.parent = current_node

        if neighbor not in open_list:
            heapq.heappush(open_list, neighbor)

    return None  # path not found

# Define a function to get the neighboring nodes
def get_neighbors(graph, node):
    neighbors = []
    for x in range(node.x - 1, node.x + 2):
        for y in range(node.y - 1, node.y + 2):
            if (x, y) == (node.x, node.y):
                continue  # skip the current node
            if (x, y) in graph:
                neighbors.append(graph[(x, y)])
    return neighbors

# Define a function to calculate the Euclidean distance between two nodes
def distance(node1, node2):
    return math.sqrt((node1.x - node2.x) * 2 + (node1.y - node2.y) * 2)

# Define a function to calculate the heuristic cost of a node
def heuristic(node, end_node):
```

```
    return distance(node, end_node)



def solve(input_list):
    graph = {}  # dictionary to store nodes in the graph
    for drone in input_list:
        start = (drone[0], drone[1])
        end = (drone[2], drone[3])
        graph[start] = Node(start[0], start[1])
        graph[end] = Node(end[0], end[1])

    paths = []
    for drone in input_list:
        start = (drone[0], drone[1])
        end = (drone[2], drone[3])
        t = drone[4]
        while True:
            # Check if the drone has reached its destination
            if graph[start].x == end[0] and graph[start].y == end[1]:
                break

            # Move the drone towards its destination
            path = astar(graph, start, end)
            if not path:
                print("No path found for drone at", start)
```

```
            break
        for node in path:
            graph[start].x = node[0]
            graph[start].y = node[1]
            time.sleep(0.1)  # wait for some time before moving to the next node
        t += len(path) * 0.1  # update the time for the next iteration
        start = (graph[start].x, graph[start].y)

    paths.append(path)

# Display the paths of the drones
fig, ax = plt.subplots()
for path in paths:
    x, y = zip(*path)
    ax.plot(x, y, marker='o')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Paths of Drones')
plt.show()
```

This function takes the input list of drones as an argument, creates a graph with nodes for the start and end positions of each drone, and then moves each drone towards its destination using the A* algorithm until it reaches its target. The function also displays the paths of the drones on a 2D plane using matplotlib.

Note that this function uses the astar function defined earlier to find the shortest path between the current position of the drone and its destination. The time.sleep(0.1) statement is used to wait for some time before moving the drone to the next node in the path, in order to simulate the movement of the drone. The time delay can be adjusted as needed.