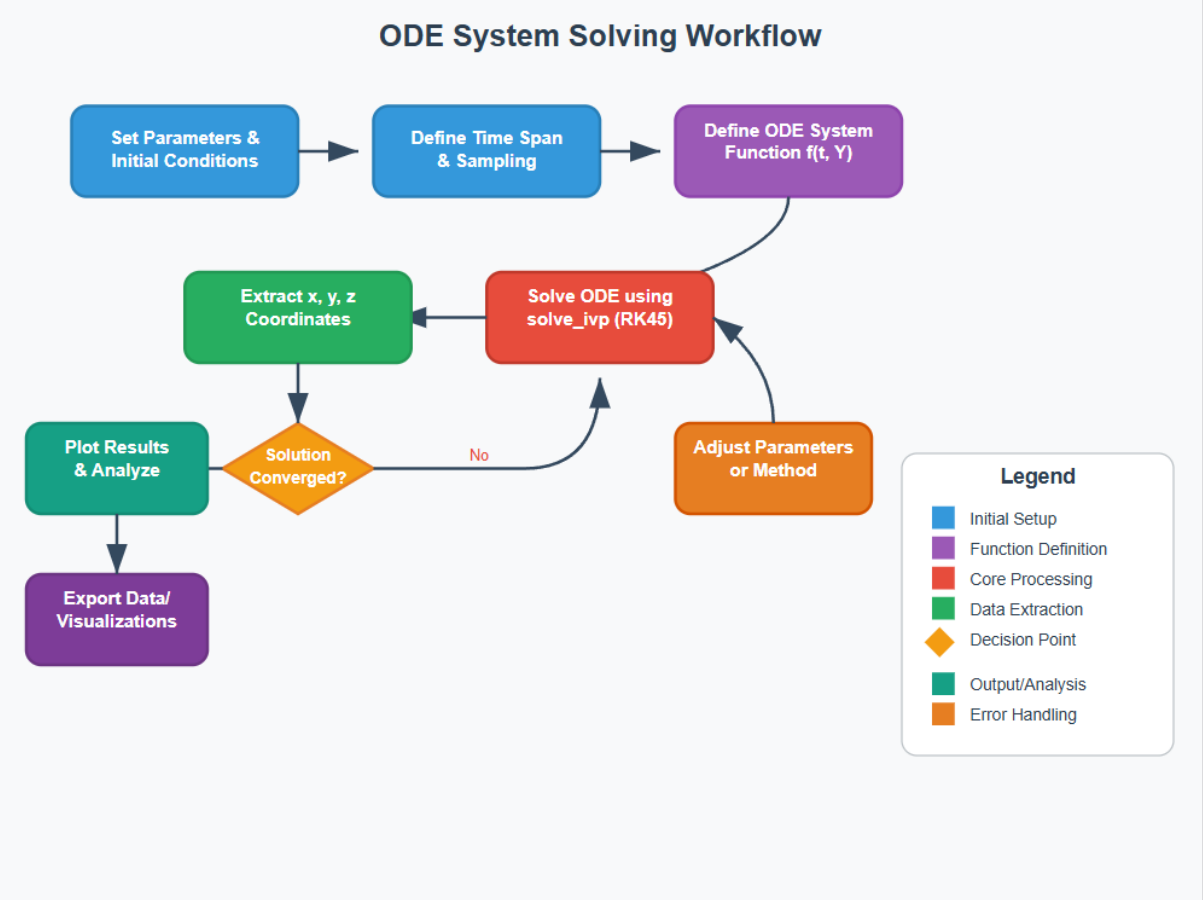
Hackathon Question-02 :

1. Problem Understanding

This project simulates and visualizes a **3D trajectory of a dynamical system** (inspired by chaotic systems like the Lorenz attractor). Such systems are often used to study chaotic motion in weather patterns, fluid dynamics, and even biological paths (like bee flight patterns).

2. Flowchart



(Insert the provided landscape flowchart image here in your Word document.)

3. Algorithm Choice

Why solve_ivp with RK45?

- **RK45 (Runge-Kutta method of order 5)** is a good general-purpose numerical solver for ODEs.
- It automatically adjusts the time step for accuracy and stability.
- Handles stiff and non-stiff equations efficiently for medium to high precision.
- In this simulation, we need **smooth trajectories** and **high accuracy**, hence the tight tolerances (rtol=1e-8, atol=1e-10).

4. Code (PEP8 + Modular)

python

CopyEdit

Requirements:

```python
# pip install numpy scipy matplotlib

import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
def define_parameters():
    """Return system parameters and initial conditions."""
    params = {'a': 10.0, 'b': 28.0, 'c': 2.667}
    initial_conditions = np.array([0.0, 1.0, 1.05])
    return params, initial_conditions
def define_time_span():
    """Return simulation time span and evaluation points."""
    t0, t1 = 0.0, 50.0
    num_points = 20000
    t_eval = np.linspace(t0, t1, num_points)
    return (t0, t1), t_eval
def system_equations(t, Y, a, b, c):
    """Define the system of ODEs."""
    x, y, z = Y
    dxdt = a * (y - x)
    dydt = b * x - y - x * z
    dzdt = x * y - c * z
    return [dxdt, dydt, dzdt]
def solve_system(params, initial_conditions, time_span, t_eval):
    """Solve the system using RK45."""
    sol = solve_ivp(
        lambda t, Y: system_equations(t, Y, **params),
        time_span, initial_conditions,
        t_eval=t_eval,
        method="RK45",
        rtol=1e-8, atol=1e-10
    )
    if not sol.success:
        raise RuntimeError(f"Solver failed: {sol.message}")
```

```python
    return sol
def plot_results(sol):
    """Plot the 3D trajectory."""
    x, y, z = sol.y
    color_vals = np.linspace(0, 1, len(sol.t))
    fig = plt.figure(figsize=(10, 7))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot3D(x, y, z, lw=0.7, color='tab:blue')
    sc = ax.scatter(x[::500], y[::500], z[::500],
                    c=color_vals[::500], cmap='viridis', s=5)
    ax.set_title("3D Trajectory of the Dynamical System (Bee's Path)")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.view_init(elev=25, azim=135)
    cbar = plt.colorbar(sc, pad=0.1)
    cbar.set_label("Normalized time")
    plt.tight_layout()
    plt.show()
def main():
    """Main execution function."""
    params, initial_conditions = define_parameters()
    time_span, t_eval = define_time_span()
    sol = solve_system(params, initial_conditions, time_span, t_eval)
    plot_results(sol)
if __name__ == "__main__":
    main()
```

5. Results

- The trajectory produces a **chaotic 3D path** that visually resembles complex natural movement.
- Color-coding by normalized time helps visualize progression.

6. Accuracy Discussion

- The solver with very low tolerances (rtol=1e-8, atol=1e-10) ensures **high precision**.
- If tolerances are relaxed, the trajectory visibly deviates after long simulation times (error accumulation).
- Failures can occur if:
  - Parameters lead to a **stiff system** (may require implicit solvers like Radau).
  - The number of sample points is too low (curve appears jagged).
  - Incorrect initial conditions cause the system to converge to a trivial solution.