Project Documentation: Ecosystem Simulation (Imperative vs. Functional)

1. Project Overview

This project simulates a simple 10x10 grid ecosystem containing **Animals (Rabbits)**, **Food**, and **Obstacles**. The simulation tracks the interactions between these entities over a set number of steps.

Simulation Rules:

- **Movement:** Animals move randomly to adjacent cells.
- **Energy:** Moving costs energy. Colliding with obstacles or other animals costs extra energy. Eating food restores energy.
- **Life Cycle:** Animals die if energy drops to 0. Animals reproduce if they have sufficient energy/age and find an empty spot.
- **Goal:** Observe how the ecosystem evolves (births, deaths, food consumption) using two fundamentally different coding approaches.

---

2. Implementation Approaches

A. Imperative Implementation (imperative_simulation.py)

This version represents the traditional "Computer Model" of programming, focusing on **how** to change the system state step-by-step.

- **Mutable State:** The grid and animals are mutable objects. When an animal moves, we directly update its coordinates (animal.r = new_r) and modify the grid cell (grid[r][c] = animal).
- **Object-Oriented Structure:** Uses a class Animal to encapsulate state (id, energy, location).
- **Control Flow:** Uses standard while and nested for loops to iterate through the simulation steps and grid cells.
- **Side Effects:** Functions like process_turn_imperative return nothing (void); they exist solely to modify the global grid and SIM_STATS variables.

B. Functional Implementation (functional_simulation.py)

This version represents the **Functional Paradigm**, focusing on stateless transformations and mathematical correctness.

- **Immutability:** Variables and data structures cannot be changed once created.
    - Instead of updating a grid cell, every step creates a *new* grid dictionary with the changes applied.
    - Instead of modifying an animal's energy, a *new* animal record is created with the updated energy value.

- **Recursion & Tail Recursion:** Since functional languages (and this implementation) avoid mutable loop counters, iteration is achieved via recursion.
    - The recursive_processor function is **Tail Recursive**, meaning the recursive call is the very last operation, allowing for efficient execution without stack growth.
- **Invariant Programming:** The simulation loop is designed using the **Principle of Communicating Vases**.
    - **Work to do (S):** The list of remaining_animals.
    - **Accumulator (A):** The accumulated_grid and accumulated_stats.
    - As the list of animals shrinks, the accumulator grows with the new state, maintaining the invariant.
- **Pure Functions:** Functions like pure_process_animal_turn calculate the next state without modifying any external variables (side-effect free).

4. Technical Comparison

Feature	**Imperative Approach**	**Functional Approach**	**Theoretical Concept**
State Management	**Mutable:** Updates memory in place (grid[x][y] = val).	**Immutable:** Returns new copies	Single Assignment
Iteration	**Loops:** while, for.	**Recursion:** Tail-recursive functions.	Invariant Programming
Data Structures	**Classes:** Objects with internal state.	**Records:** Dictionaries treated as data-only.	Record Structures
Execution Flow	**Sequence:** Step-by-step instructions.	**Composition:** combining functions (merge_stats, sim_step).	Function Composition