TED UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

CMPE 491 – Senior Project I

MoonAI
High-Level Design Report

Team Members

Caner Aras

Emir Irkılata

Oğuzhan Özkaya

Supervisor

Ozan Zorlu

Project Web Page

December 30, 2025

Table of Contents

1. Introduction 1
1.1 Purpose of the System 1
1.2 Design Goals 1
1.3 Definitions, Acronyms, and Abbreviations 2
1.4 Overview 2
2. Current software architecture 2
3. Proposed software architecture 3
3.1 Overview 3
3.2 Subsystem Decomposition 4
3.2.1 Simulation Engine Subsystem 4
3.2.2 Evolution Core Subsystem 5
3.2.3 Visualization Manager Subsystem 5
3.2.4 Data Management Subsystem 5
3.3 Hardware/software mapping 6
3.3.1 Hardware Architecture 6
3.3.2 Software Stack 7
3.4 Persistent Data Management 7
3.5 Global Software Control 8
3.5.1 Initialization Phase 9
3.5.2 Main Execution Loop 9
3.5.3 Termination Phase 10
3.6 Boundary Conditions 10
4. Subsystem Services 11
4.1 Simulation Engine Subsystem Services 11
4.2 Evolution Core Subsystem Services 12
4.3 Visualization Manager Subsystem Services 14
4.4 Data Management Subsystem Services 15
5. Conclusion 16
6. Glossary ..... a
7. References ..... b

1. Introduction

This document presents the High-Level Design for the MoonAI project, translating the requirements and conceptual models defined in the Analysis Report into a concrete software architecture. It serves as the primary blueprint for the implementation phase, defining the subsystem decomposition, hardware/software mapping, and global control strategies required to realize the project's research goals. The design strategies detailed herein ensure the system meets the functional and non-functional requirements established in the previous analysis phase.

1.1 Purpose of the System

MoonAI is a modular and extensible experimental platform designed to analyse evolutionary algorithms and neural network evolution. Unlike existing ecological simulations, MoonAI uses a simplified predator-prey environment solely as a synthetic benchmark to evaluate algorithmic behaviours quantitatively. The system focuses on the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [1], [2], enabling researchers to study structural innovation, complexity growth, and emergent behavioural strategies under controlled, dynamic conditions.

1.2 Design Goals

The system design is driven by the following key goals derived from the non-functional requirements:

- **Performance**: Achieve real-time execution (> 30 FPS) with large populations by utilizing GPU acceleration (CUDA) for fitness evaluation and neural inference.
- **Modularity**: Decouple simulation logic, evolutionary computation, logging, and visualization to ensure the system is extensible.

- **Reproducibility**: Ensure deterministic experimental results through strict seed management and configuration versioning.
- **Observability**: Provide real-time visualization for debugging and "research-grade" data logging for post-hoc analysis.
- **Portability**: Support execution on personal workstations with different conditions and features.

1.3 Definitions, Acronyms, and Abbreviations

- **NEAT**: NeuroEvolution of Augmenting Topologies [1].
- **SFML**: Simple and Fast Multimedia Library [7].
- **CUDA**: Compute Unified Device Architecture (NVIDIA) [8].
- **Agent**: An autonomous entity (predator or prey) controlled by an evolutionary algorithm.
- **Genome**: The genetic representation of a neural network topology.

1.4 Overview

MoonAI is a desktop application primarily implemented in C++. It functions as a simulation loop that integrates a physics-lite environment, a genetic evolution engine, and a rendering pipeline. The user interacts with the system via configuration files to set up experiments and a GUI for real-time observation.

2. Current software architecture

There is no existing system that fulfils MoonAI’s research goals. While agent-based simulations [5] and predator–prey models exist, their objectives differ fundamentally from MoonAI’s purpose.

Limitations in Available Systems:

- They focus on ecological accuracy, not the analysis of evolutionary computation techniques.
- They use predefined behaviours instead of neuroevolutionary driven decision-making.
- They lack comprehensive control over genetic encodings, mutation operators, and evolutionary parameters.
- They rarely support topology-evolving neural networks such as NEAT.
- They do not provide GPU-accelerated computation for large experimental populations.
- They are not designed to produce structured datasets for machine learning research.
- They do not meet academic requirements for transparency, ethical compliance, or reproducibility.

Because no existing software combines:

- a controllable artificial environment
- an extensible evolutionary algorithm framework
- neural topology evolution
- research-grade data generation
- real-time visualization
- GPU-accelerated processing

The MoonAI system must be developed as a new, custom research platform.

3. Proposed software architecture

3.1 Overview

The proposed architecture follows an Object-Oriented, Modular design pattern [3]. The system is divided into four primary distinct subsystems: Simulation Engine, Evolution Core, Visualization, and Data Management. This separation of concerns ensures that the evolutionary algorithm can be swapped or modified without rewriting the physical simulation rules.

3.2 Subsystem Decomposition

The MoonAI system follows a Modular Architecture pattern, dividing the application into four distinct subsystems. This decomposition strategy ensures a strict separation of concerns: the simulation logic is decoupled from the visualization, and the evolutionary algorithm (NEAT) functions independently of the physical environment rules.

This modularity supports the project's extensibility requirements, allowing researchers to modify genetic operators or agent behaviours without necessitating major refactoring of the rendering or data logging components.

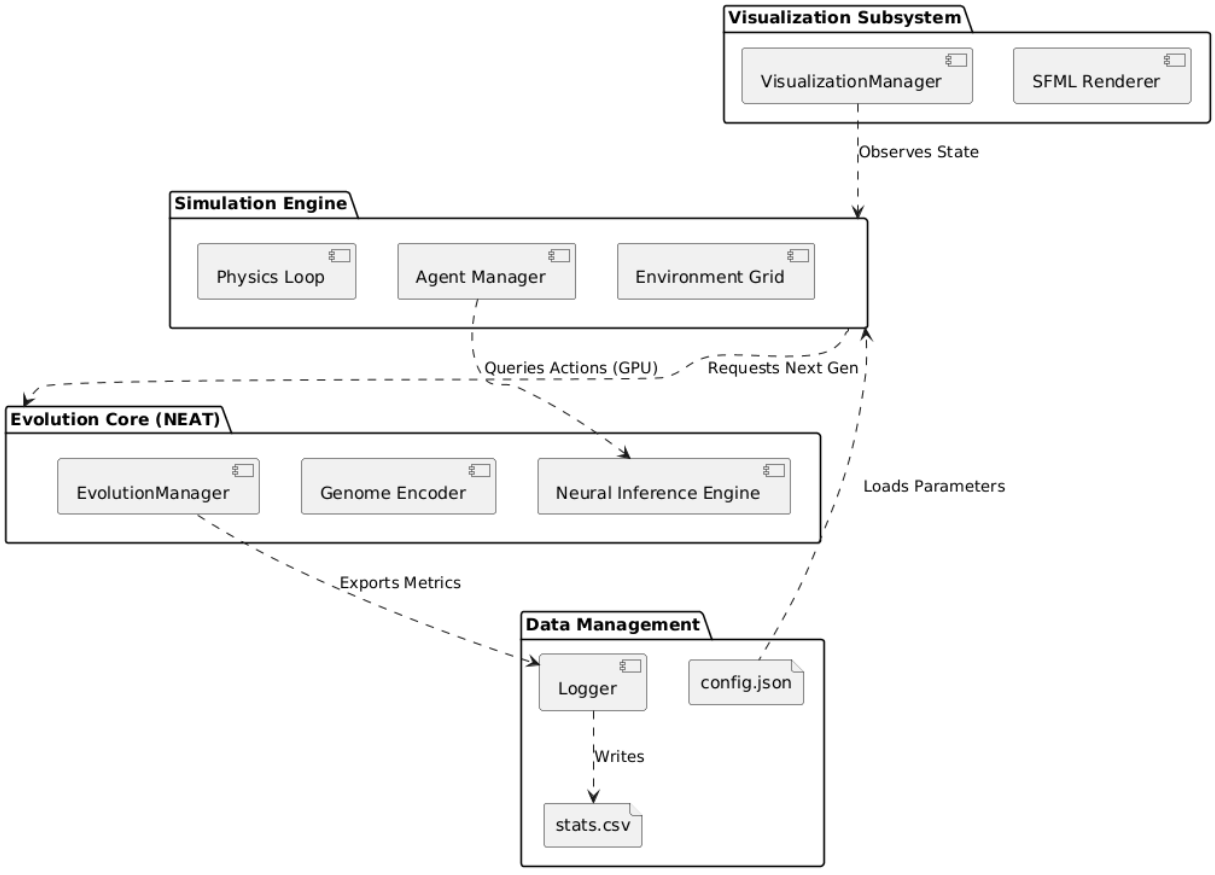

Figure 1: Subsystem Decomposition and Module Dependencies

3.2.1 Simulation Engine Subsystem

The Simulation Engine is the core component responsible for maintaining the state of the synthetic physical world. It operates on a discrete time-step basis, ensuring deterministic updates for all entities.

- **Responsibilities:** Manages the global simulation loop, collision detection, resource spawning, and agent sensing logic.
- **Key Classes:** SimulationManager, Environment, Grid, Agent, Predator, Prey.

3.2.2 Evolution Core Subsystem

This subsystem encapsulates the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [1]. It acts as the "brain" of the operation, determining how agents change and improve over generations.

- **Responsibilities:** Manages population pools, executes genetic operations (selection, crossover, mutation, speciation), and handles neural network inference (forward pass).
- **Key Classes:** EvolutionManager, Genome, NeuralNetwork.

3.2.3 Visualization Manager Subsystem

This subsystem handles all user-facing output and interaction. It bridges the gap between the internal simulation state and the researcher's screen.

- **Responsibilities:** Renders the 2D grid, agents, and UI overlays in real-time (> 30 FPS). It also captures and processes user inputs for simulation control (pause, speed, restart). Graphical representation of the experiment using the SFML library [7].
- **Key Classes:** VisualizationManager, SFML Wrappers.

3.2.4 Data Management Subsystem

This subsystem manages the persistence of experimental data, ensuring that results are preserved for offline analysis.

- **Responsibilities:** Collects runtime metrics (fitness scores, topology statistics, species diversity) and exports them to structured formats (CSV/JSON) compatible with Python analysis tools.
- **Key Classes:** Logger

3.3 Hardware/software mapping

MoonAI is designed as a standalone executable running on standard workstation hardware. This section defines the deployment strategy for the MoonAI system, detailing how software artifacts map to physical hardware resources to achieve the performance goal of real-time execution (< 30 FPS).

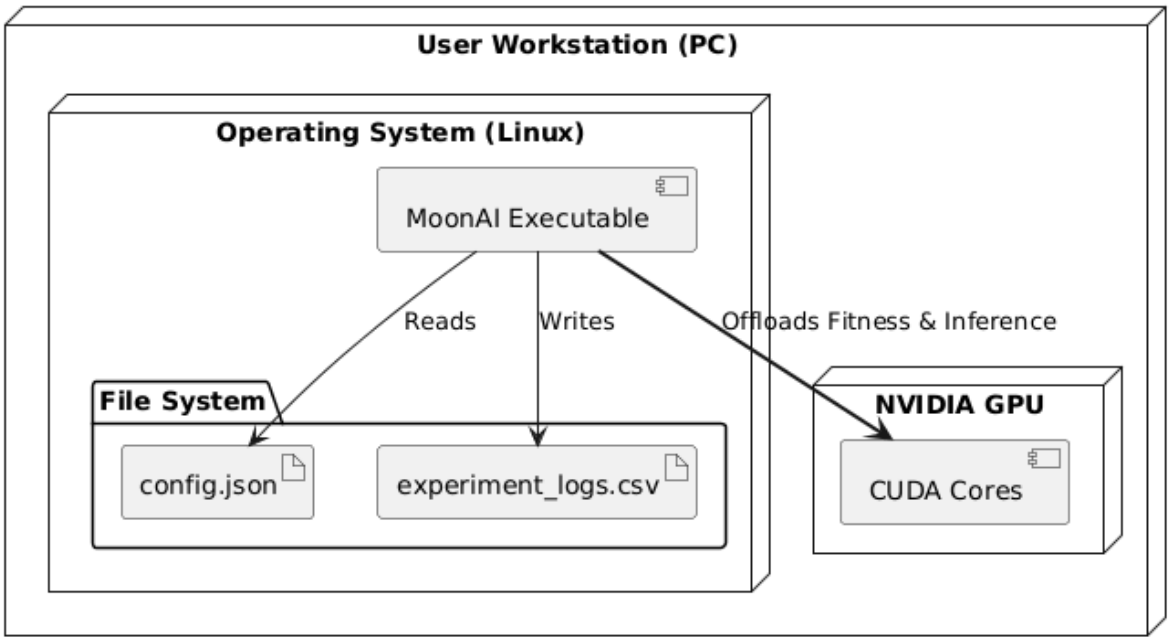

Figure 2: Hardware/Software Mapping

3.3.1 Hardware Architecture

The system is designed for a standard workstation environment with a heterogeneous computing architecture, leveraging both the Central Processing Unit (CPU) and a dedicated Graphics Processing Unit (GPU).

- **Central Processing Unit (CPU):** The CPU acts as the host controller. It executes the sequential application logic, including the main simulation loop, physics updates (collision detection), user interface event polling, and file I/O operations.
- **Graphics Processing Unit (NVIDIA GPU):** The system requires a CUDA-capable GPU to handle massive parallelism. The GPU executes custom CUDA kernels [8] for two specific tasks:
    - **Neural Inference:** Calculating the output of thousands of agent neural networks simultaneously per time step.
    - **Fitness Evaluation:** Aggregating performance metrics for large populations in parallel.

3.3.2 Software Stack

The system is built upon a high-performance stack compatible with Linux environment.

- **Linux** as operating system.
- **C++ Compiled Binary** for runtime.
- **SFML** [7] for simulation visualization and rendering.
- **CUDA Toolkit** [8] for heavy machine learning computations [6].
- **Python** for external analysis and processing output logs.

3.4 Persistent Data Management

MoonAI utilizes a File System-based Persistence Strategy rather than a relational database. This approach reduces overhead of large scale timeseries data, increases the performance, and reduces the complexity of the project, also simplifies the transport of experiment data between researchers and different environments like data analysis at python.

- **Configuration Management (Input):**
    - Experiment parameters (e.g., mutation_rate, grid_width, predator_count) are decoupled from the code and stored in human-readable JSON files (e.g., config.json).

    - This allows researchers to modify simulation variables and restart experiments immediately without recompiling the source code.
- **Data Logging (Output):**
    - **Time-Series Data:** Per-generation statistics (average fitness, species count) are streamed to CSV files (stats.csv) for efficient sequential writing and easy import into Pandas.
    - **Structural Data:** Complex data structures, such as the full neural topology of the "Best Agent" in a generation, are serialized to JSON (genomes.json) to preserve the graph structure of nodes and connections.
- **Reproducibility:**
    - To satisfy scientific rigor, every experiment initializes its Random Number Generator with a specific seed.
    - This **Master Seed** is logged at the start of every output file. If a researcher inputs this seed back into the configuration, the system guarantees an identical replay of the evolutionary trajectory.

3.5 Global Software Control

The global control flow follows a **hybrid Event-Driven and Time-Stepped architecture**. The application runs in a continuous loop that advances simulation time in discrete "ticks" while asynchronously listening for user interventions.

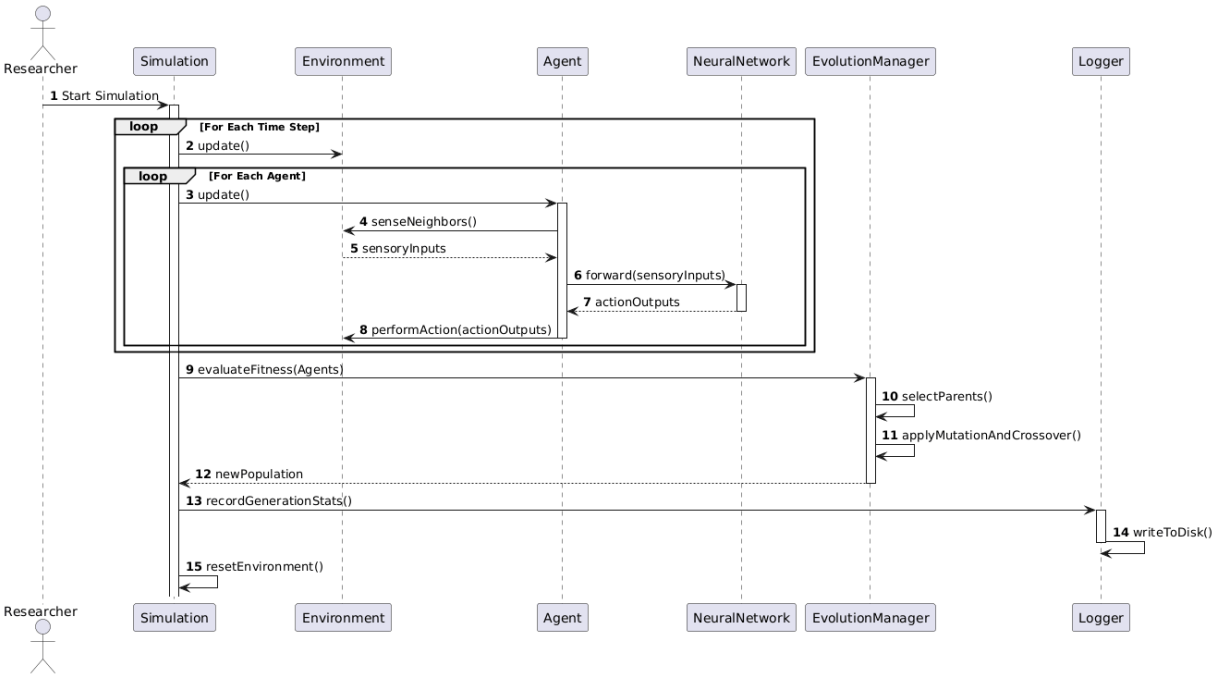

Figure 3: Global Control Sequence (Single Generation Loop)

3.5.1 Initialization Phase

Before the main loop begins, the system performs a sequential setup:

1. **Load Config:** The ConfigLoader reads and validates the JSON configuration file.
2. **Init GPU:** The EvolutionManager allocates necessary VRAM buffers for the population.
3. **Spawn Population:** The Environment instantiates the initial generation of Predator and Prey agents with random neural topologies.

3.5.2 Main Execution Loop

The loop runs continuously until a termination condition is met.

1. **Poll Events:** The VisualizationManager captures inputs (keyboard/mouse) to handle Pause, Resume, or Speed Up commands immediately.
2. **Update Phase (Physics & Logic):**
    a. The Simulation advances the logical time step.
    b. **Neural Inference:** All agents send sensory data to the GPU; the GPU computes action outputs and returns them to the CPU.

c. **Physics Update:** Agents move and interact (collide/eat) based on these actions.

3. **Evolution Phase (End of Generation):**
    a. If the time steps per generation are exhausted, the EvolutionManager pauses the physical simulation.
    b. It calculates fitness, performs selection, crossover, and mutation, and replaces the old population with new agents.
4. **Render Phase:**
    a. The VisualizationManager draws the current state of the grid and agents using SFML.

3.5.3 Termination Phase

Triggered when the maximum number of generations is reached or the user manually exits.

- **Cleanup:** The system flushes any remaining data in the write buffers to disk to prevent data loss.
- **Deallocation:** GPU memory is freed, and the application closes gracefully.

3.6 Boundary Conditions

This section defines how the system handles edge cases, hardware limitations, and failure states to ensure robustness.

- **Startup Validation:**
    - The system verifies not just the existence of config.json but also the validity of its schema. Illogical values (e.g., negative population size, probability > 1.0) trigger an immediate error report, preventing undefined behaviour during runtime.
    - **Hardware Fallback:** During initialization, the system checks for a CUDA-compatible device. If unavailable, it falls back to a CPU-based inference engine, logging a warning that performance will be degraded.
- **Resource Exhaustion & Scalability:**

    - **VRAM Management:** When scaling to large populations (thousands of agents), the system monitors GPU memory usage. If the population size exceeds available VRAM, the system will cap reproduction or terminate safely with an "Out of Memory" log rather than crashing the driver.
    - **Memory Leaks:** The C++ implementation utilizes RAII (Resource Acquisition Is Initialization) principles (e.g., smart pointers) to automatically manage memory lifecycles, ensuring stability during multi-day evolutionary runs.
- **Failure Recovery:**
    - In the event of an unexpected crash (e.g., power loss or driver failure), the Logger is designed to flush streams after every generation. This ensures that even if the run is aborted, the data from all previous completed generations is preserved for analysis.

4. Subsystem Services

This section details the specific services, responsibilities, and internal logic provided by each of the four primary subsystems defined in the system decomposition.

4.1 Simulation Engine Subsystem Services

The Simulation Engine is responsible for maintaining the main event loop of the simulation. It manages the global time-step loop, enforces environmental constraints, and orchestrates the agent decision cycle.

- **initializeSimulation(config)**: Parses the configuration object to construct the grid, place static obstacles, and spawn the initial population of predators and prey at valid locations.
- **updateState(timeStep)**: Advances the simulation by one discrete time step. This service triggers the decide() method for every agent and updates their physical positions based on velocity and collision checks.
- **manageGrid()**: Provides a spatial hashing or grid-based lookup service (getNeighbors) that allows agents to efficiently query their surroundings for nearby entities without checking every object in the world.

- **enforceConstraints()**: Ensures agents stay within bounds, consume resources correctly, and die if their energy reaches zero.

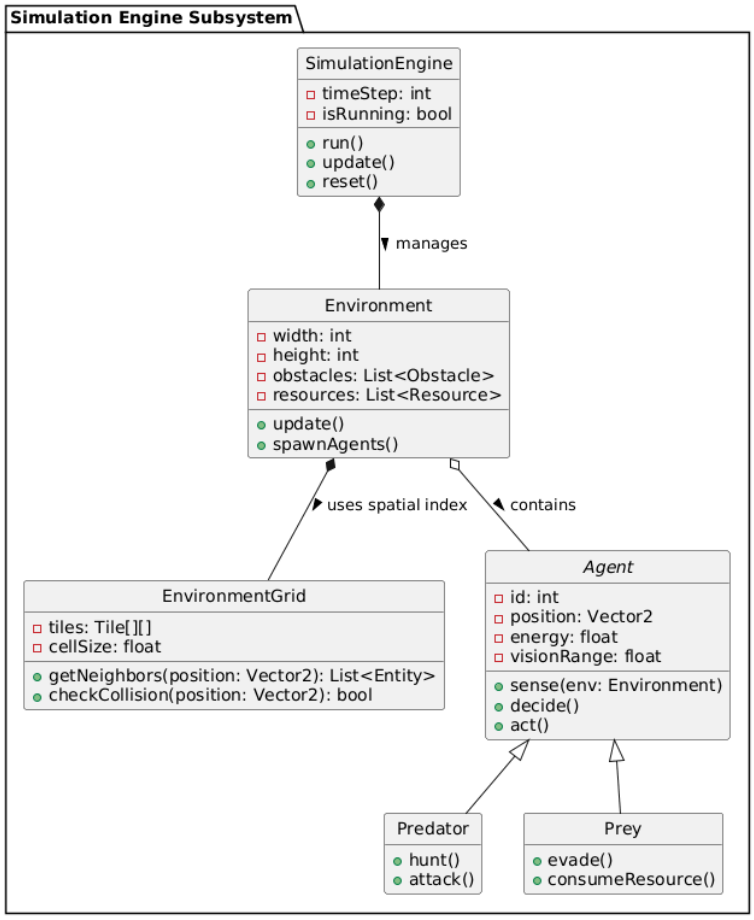

Figure 4: Simulation Engine Subsystem Internal Design

4.2 Evolution Core Subsystem Services

This subsystem encapsulates the NEAT (NeuroEvolution of Augmenting Topologies) algorithm. It is strictly separated from the physics engine to ensure that evolutionary logic does not interfere with simulation rules.

- **evaluateFitness(population)**: Calculates a performance score for every agent at the end of a generation based on metrics like survival time, energy gained, or prey captured.
- **reproduce(parents)**: Generates a new population for the next generation. This service handles complex genetic operations including crossover (combining parent genomes) and mutation (adding nodes/connections).

- **speciate()**: Groups genomes into species based on topological similarity. This service protects novel structural innovations from being eliminated prematurely by competition.
- **constructNetwork(genome)**: Translates a genetic code (genotype) into a functional neural network (phenotype) capable of inference.

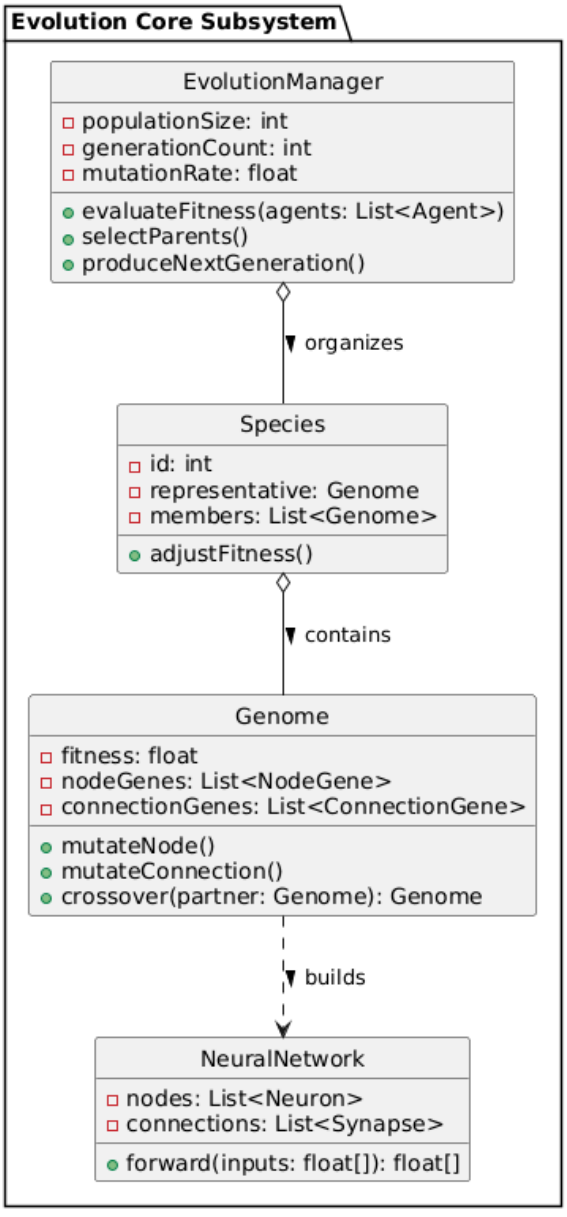

Figure 5: Evolution Core Subsystem Internal Design

4.3 Visualization Manager Subsystem Services

This subsystem manages the real-time graphical representation of the experiment using the SFML library. It serves as the primary interface for researcher observation and debugging.

- **renderFrame(simulationState)**: Draws the current state of the environment, agents, and resources to the window. It differentiates predators and prey using distinct visual markers (e.g., colors or shapes).
- **handleInput()**: Listens for and processes user commands, such as pausing the simulation, adjusting the speed multiplier, or toggling debug overlays.
- **updateStatsOverlay()**: Renders real-time text or mini graphs on top of the simulation view, displaying current generation, live population counts, and FPS.

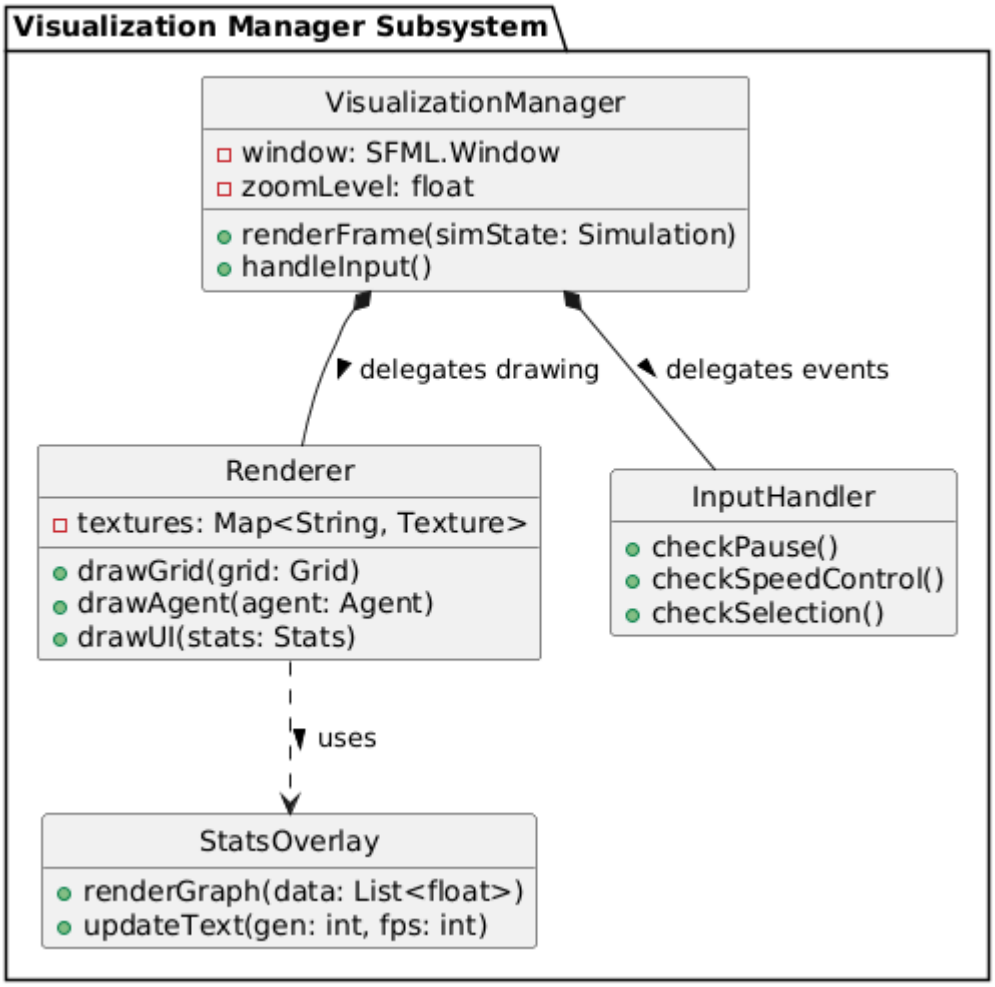

Figure 6: Visualization Manager Subsystem Internal Design

4.4 Data Management Subsystem Services

The Data Management subsystem ensures the persistence and portability of experimental results. It handles the "input/output" boundary of the application.

- **loadConfiguration(filepath)**: Reads experiment parameters (e.g., mutation rates, grid size) from JSON/YAML files and validates them before the simulation starts.
- **logGeneration(metrics)**: buffers critical data at the end of every generation, including max fitness, average complexity, and species count.
- **exportData()**: Writes the buffered logs to disk in standardized CSV or JSON formats compatible with Python analysis tools (Pandas/Matplotlib), ensuring data is available for post-hoc research.

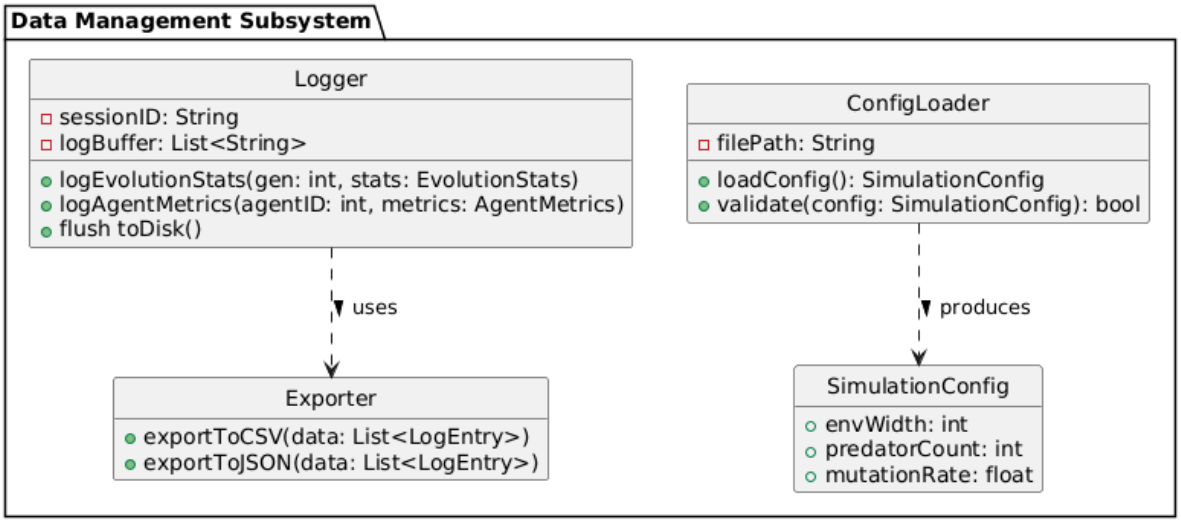

Figure 7: Data Management Subsystem Internal Design

5. Conclusion

The High-Level Design presented herein establishes a robust architectural foundation for MoonAI, positioning it as a capable research platform for evolutionary computation. By decomposing the system into distinct subsystems—Simulation, Evolution, Visualization, and Data Management—the design successfully addresses the critical need for a modular environment where evolutionary algorithms can be tested independently of physical simulation rules.

This architecture explicitly resolves the performance limitations found in existing ecological models by integrating NVIDIA CUDA for GPU acceleration. This hardware-software mapping ensures that the computationally intensive tasks of fitness evaluation and neural inference can scale to large populations without compromising the target real-time framerate of > 30 FPS. Furthermore, the strict separation of the NEAT algorithm from the agent simulation logic allows researchers to isolate and analyse specific variables in structural evolution.

Ultimately, this design prioritizes academic rigor and reproducibility. By enforcing deterministic seed management and structured data logging compatible with Python analysis tools, MoonAI fills the gap identified in current software offerings. The proposed system provides a stable, verifiable, and extensible framework, paving the way for the subsequent Detailed Design and Implementation phases.

6. Glossary

A	
Agent	An autonomous entity in the simulation (predator or prey) that senses the environment, evaluates inputs using a neural network, and performs actions such as moving, chasing, escaping, or consuming resources.
C	
Crossover	A genetic operation in evolutionary algorithms where two parent genomes combine to form an offspring genome, inheriting characteristics from both.
F	
Fitness	A quantitative measure of an agent's performance (e.g., survival duration, prey hunted, energy maintained) used by the evolution engine to select parents for reproduction.
Frame Rate (FPS)	The frequency at which the visualization subsystem renders images, used as a performance metric.
G	
Generation	A complete cycle of evaluating a population, selecting parents, and producing a new set of genomes.
Genome	A structured representation of an individual's neural architecture in the NEAT algorithm, including node genes, connection genes, and mutation history.
L	
Logger	Component responsible for recording simulation data, fitness values, neural architectures, and exporting structured logs for post-analysis.
M	
Mutation	A stochastic alteration to a genome used to introduce diversity in evolution.
P	
Population	The total set of genomes or agents in one evolutionary generation.
Predator / Prey	Two predefined agent categories with distinct behavioural roles; Predators chase and attack prey, while Prey attempt to evade and survive.
S	
Seed	An initial integer value used to initialize the pseudo-random number generator, ensuring that the sequence of "random" events can be identically reproduced in future runs.
Simulation Step	A single iteration of the simulation loop where all agents sense, decide, and act.
T	
Time Step	A discrete unit of simulated time.

7. References

[1] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, Jun. 2002.

[2] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Evolving Adaptive Neural Networks with and without Adaptive Synapses," in *Proc. 2003 IEEE Congr. on Evolutionary Computation*, Canberra, ACT, Australia, Dec. 2003, pp. 2557–2564.

[3] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2004.

[4] D. Floreano and C. Mattiussi, *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies*. Cambridge, MA, USA: MIT Press, 2008.

[5] U. Wilensky. (1999). *NetLogo* [Software]. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Available: http://ccl.northwestern.edu/netlogo/

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[7] SFML Development Team. (2023). *Simple and Fast Multimedia Library - Official Documentation* [Online]. Available: https://www.sfml-dev.org/

[8] NVIDIA Corporation. (2023). *CUDA Toolkit Documentation* [Online]. Available: https://docs.nvidia.com/cuda/

[9] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, USA: University of Michigan Press, 1975.