TED UNIVERSITY

Department of Computer Engineering

MoonAI

Low-Level Design Report

Team:

Caner Aras Emir Irkılata Oğuzhan Özkaya

Supervisor: Ayşenur Birtürk

CMPE 492 — Senior Project II

Spring 2026

Table of Contents

1 Introduction

This Low-Level Design (LLD) document describes the complete class-level architecture of **MoonAI**: a predator-prey simulation for studying NeuroEvolution of Augmenting Topologies (NEAT) and evolutionary algorithms. It serves as the authoritative reference linking the High-Level Design (HLD, December 2025) to the actual C++17 implementation.

The HLD defined six subsystems at a conceptual level. This document closes the gap to the source code: every class, struct, enum, algorithm, and design decision is documented with enough precision that a developer could re-implement any component independently.

1.1 Object Design Trade-offs

Five key trade-offs shaped the implementation.

1.1.1 Value Semantics vs. Pointer Semantics for `Genome`

`EvolutionManager::population_` stores `Genome` objects by value (`std::vector<Genome>`). `Species::members_` holds raw, non-owning `Genome*` pointers into that contiguous vector.

Alternatives considered:

- `std::shared_ptr<Genome>` — would eliminate the raw pointer but incurs reference-counting overhead and heap fragmentation for every genome in the hot evaluate/reproduce loop.
- `std::unique_ptr<Genome>` — moves ownership but breaks the contiguous layout needed for GPU packing.

Decision: The species list is rebuilt at the start of every generation *before* the population vector is reallocated, so the raw pointers are valid for the entire lifetime of one generation. The value-semantics vector gives cache-friendly iteration and trivial move/copy semantics for checkpointing.

1.1.2 GPU CSR Layout vs. Padded Layout

The GPU inference path packs all agent networks into flat device arrays using Compressed Sparse Row (CSR) encoding. Each agent is described by a `GpuNetDesc` containing eight integer offsets into shared flat arrays.

Alternatives considered:

- **Padded layout** — each network occupies a fixed-size slot (`MAX_NODES` × `MAX_NODES`). Simple indexing, but wastes up to 10× VRAM for the small early-generation topologies that NEAT produces.

- **Per-agent device allocation** — flexible but requires thousands of `cudaMalloc` calls and non-coalesced memory access.

Decision: CSR eliminates VRAM waste and enables coalesced reads along the `eval_order` and `conn_w` arrays. The host-side packing step (`upload_network_data`) runs once per generation, so its O(N) complexity is acceptable.

1.1.3 Precomputed Incoming Adjacency in `NeuralNetwork`

At construction, `NeuralNetwork` builds `incoming_[node_idx]`, a list of `(source_idx, weight)` pairs for each node. The `activate()` method then iterates this precomputed structure.

Alternative: Build the adjacency on each forward pass from the raw connection list. Profiling showed this as the dominant cost in the CPU inference path (map allocation in the hot loop).

Decision: One-time O(E) build cost at network construction; zero allocation per `activate()` call. Activation state is stored in a pre-allocated `float` vector `values_` of fixed size.

1.1.4 Static Utility Classes (`Mutation`, `Crossover`, `Physics`)

These classes expose only `static` methods and carry no instance state.

Alternative: Instantiable objects or singletons. Singletons would require locking for potential future multi-threading; instantiable objects would add unnecessary storage.

Decision: Pure functions are simpler to reason about, easier to test (no construction overhead), and completely thread-safe given the same `Random` instance. All operators are deterministic given the same `Random` state, so unit tests verify correctness via outcome assertions rather than mock objects.

1.1.5 `TickCallback` Decoupling

`EvolutionManager` exposes a `std::function<void(int, const SimulationManager&)>` type alias called `TickCallback` rather than holding a raw `Logger*`.

Alternative: Pass a `Logger*` directly — simpler but couples the evolution engine to the data layer.

Decision: The callback pattern lets `main.cpp` wire logging, live visualization updates, and benchmarking probes independently. The slight `std::function` overhead (one indirect call per tick) is negligible compared with the simulation physics cost.

1.2 Interface Documentation Guidelines

- All public interfaces are defined exclusively in `.hpp` files under `src/`.

- Include paths are written relative to the `src/` root, e.g. `#include "evolution/genome.hpp"`.
- **`const` correctness:** all read-only accessors are `const`; mutation methods are clearly non-const.
- **RAII:** all resources are managed by constructors/destructors — GPU memory (`GpuBatch`), file handles (`Logger`), and the SFML window (`VisualizationManager`).
- All symbols reside in namespace `moonai`; CUDA internals reside in `moonai::gpu`.
- This document uses exact C++17 signatures as the canonical interface specification.

1.3 Engineering Standards

- **UML 2.5** — Class diagrams use standard notation: + public, − private, # protected; `<<abstract>>`, `<<static>>` stereotypes; standard relationship arrows (inheritance, composition, dependency).
- **IEEE 1016-2009** — *Software Design Descriptions*: the structure of this document follows this standard.
- **C++17 ISO/IEC 14882:2017** — implementation language.
- **NEAT specification** — Stanley & Miikkulainen (2002) [1] defines correctness requirements for the compatibility-distance formula, speciation, and reproduction.
- **ACM Code of Ethics** — reproducibility and transparency are maintained via seeded RNG, checkpointing, and open source tooling.

1.4 Definitions, Acronyms, and Abbreviations

See Section 9 for the full glossary. Key terms used throughout this document:

NEAT	NeuroEvolution of Augmenting Topologies — the evolutionary algorithm implemented in `moonai::evolution`.
CSR	Compressed Sparse Row — the GPU memory layout for variable-topology networks.
RAII	Resource Acquisition Is Initialization — C++ idiom for deterministic resource lifetime.
RNG	Random Number Generator; the project uses MT19937-64 via `std::mt19937_64`.
NN	Neural Network — a `NeuralNetwork` object built from a `Genome`.
HLD	High-Level Design document (December 2025).

LLD This document — Low-Level Design.

SensorInput The 15-element observation vector fed to each agent's NN per tick.

TickCallback `std::function<void(int, const SimulationManager&)>` — hook invoked after each simulation tick.

GpuBatch RAII wrapper for all CUDA device memory needed for one generation.

InnovationTracker Per-generation map from connection pairs to innovation numbers, ensuring history alignment for NEAT crossover.

2 Package Overview

MoonAI is structured as six CMake static libraries under `src/`. Each library corresponds to one architectural subsystem with a single, well-defined responsibility. Figure 1 shows the package dependency graph.

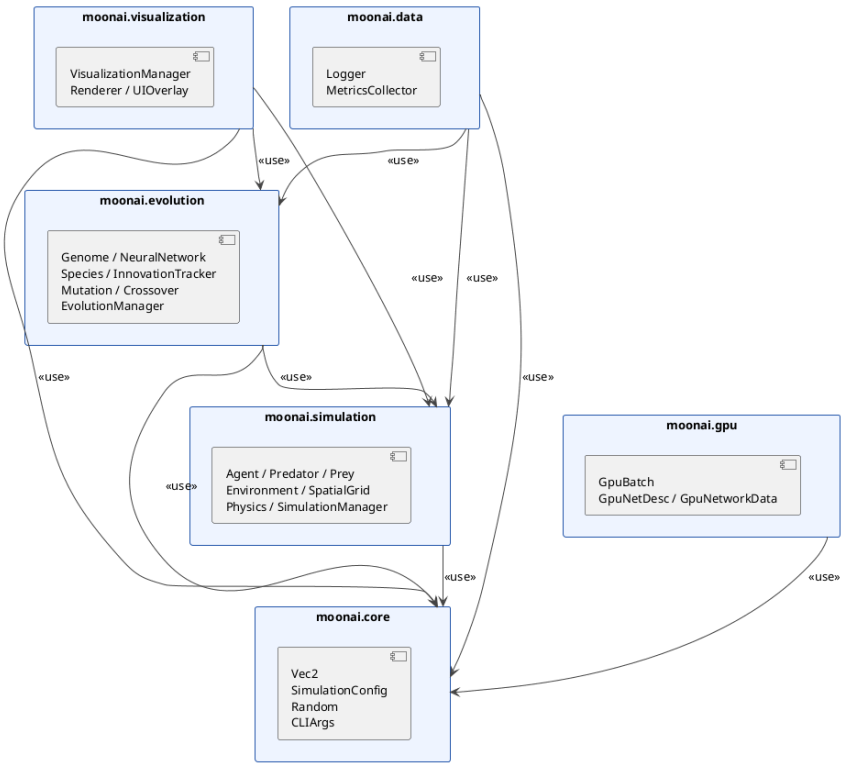

Figure 1: Package dependency diagram. All packages consume `moonai.core`; arrows denote `<<use>>` relationships.

Table 1: Package summary

Package	CMake library	Source path
moonai.core	moonai_core	src/core/
moonai.simulation	moonai_simulation	src/simulation/
moonai.evolution	moonai_evolution	src/evolution/
moonai.visualization	moonai_visualization	src/visualization/
moonai.data	moonai_data	src/data/
moonai.gpu	moonai_gpu	src/gpu/

Key classes by package:

- moonai.core: Vec2, SimulationConfig, Random, CLIArgs, BoundaryMode
- moonai.simulation: Agent, Predator, Prey, Environment, SpatialGrid, Physics, SensorInput, SimulationManager
- moonai.evolution: Genome, NeuralNetwork, Species, InnovationTracker, Mutation, Crossover, EvolutionManager
- moonai.visualization: VisualizationManager, Renderer, UIOverlay, OverlayStats
- moonai.data: Logger, MetricsCollector, GenerationMetrics
- moonai.gpu: GpuBatch, GpuNetDesc, GpuAgentStats, GpuNetworkData

2.1 Dependency Rules

- **moonai.core** has no internal dependencies. It provides primitive types (Vec2, AgentId), configuration (SimulationConfig), and the seeded RNG (Random) consumed by every other package.
- **moonai.gpu** depends only on gpu_types.hpp from moonai.core. It has no link-time dependency on moonai.evolution, which prevents circular dependencies; the host-side packing of GPU network data lives in EvolutionManager.
- **moonai.simulation** depends on moonai.core only. The simulation engine is deliberately unaware of NEAT to allow independent testing.
- **moonai.evolution** depends on moonai.core and moonai.simulation (for SimulationManager in evaluate_generation).
- **moonai.visualization** and **moonai.data** are leaf consumers: they depend on all domain packages but nothing depends on them.

3 Core Package (src/core/)

The core package provides shared primitive types, configuration loading, CLI argument parsing, and the seeded random-number generator used throughout the simulation.

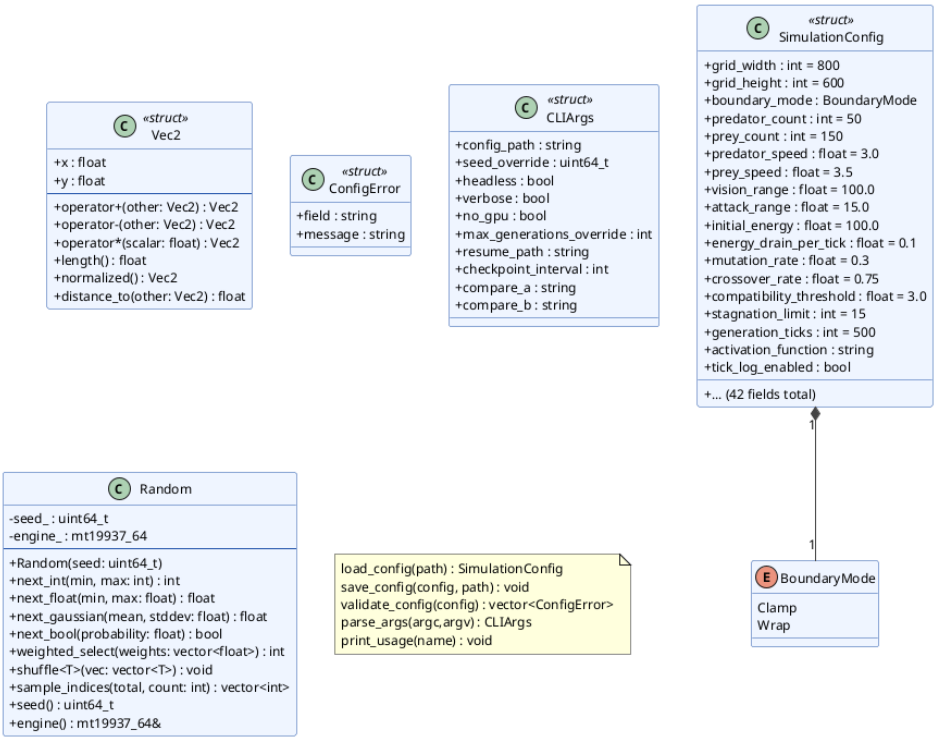

Figure 2: Core package class diagram.

3.1 `Vec2` (struct)

Plain aggregate for 2-D floating-point coordinates. Chosen as a struct (no encapsulation overhead) because it appears in tight physics loops.

Table 2: `Vec2` interface

Member	Type	Description
`x`	`float`	Horizontal coordinate (default 0.0)
`y`	`float`	Vertical coordinate (default 0.0)
`operator+(other)`	`Vec2`	Component-wise addition
`operator-(other)`	`Vec2`	Component-wise subtraction
`operator*(scalar)`	`Vec2`	Scalar multiplication
`length()`	`float`	Euclidean length $\sqrt{x^2 + y^2}$
`normalized()`	`Vec2`	Unit vector (zero-safe)
`distance_to(other)`	`float`	Euclidean distance to another `Vec2`

`AgentId` is a type alias: `using AgentId = std::uint32_t`. It uniquely identifies each agent within a generation.

3.2 `BoundaryMode` (enum class)

```
1 enum class BoundaryMode { Clamp, Wrap };
```

`Clamp` constrains positions to $[0, w] \times [0, h]$. `Wrap` applies toroidal topology via `std::fmod`. Controlled by the `"boundary_mode"` field in `default_config.json`.

3.3 `SimulationConfig` (struct)

Central parameter object. Loaded from JSON; passed by value throughout the system.

Table 3: `SimulationConfig` fields — Environment and Simulation (28 fields)

Field	Type	Def.	Range	Description
Environment				
`grid_width`	`int`	800	> 0	World width in pixels
`grid_height`	`int`	600	> 0	World height in pixels
`boundary_mode`	`BoundaryMode`	Wrap	—	Clamp or Wrap
Population				
`predator_count`	`int`	50	≥ 1	Number of predator agents
`prey_count`	`int`	150	≥ 1	Number of prey agents
Agent				
`predator_speed`	`float`	3.0	> 0	Max predator speed (px/tick)
`prey_speed`	`float`	3.5	> 0	Max prey speed (px/tick)
`vision_range`	`float`	100.0	> 0	Sensor vision radius (px)
`attack_range`	`float`	15.0	> 0	Predator kill radius (px)
`initial_energy`	`float`	100.0	> 0	Starting energy per agent
`energy_drain_per_tick`	`float`	0.1	≥ 0	Energy lost each tick
`energy_gain_from_kill`	`float`	50.0	≥ 0	Energy gained on kill
`energy_gain_from_food`	`float`	30.0	≥ 0	Energy gained from food
`food_pickup_range`	`float`	10.0	> 0	Food collection radius (px)
Food				
`food_count`	`int`	200	≥ 0	Maximum food items on map
`food_respawn_rate`	`float`	0.05	$[0, 1]$	Probability each depleted food respawns per tick
Simulation loop				
`generation_ticks`	`int`	500	> 0	Ticks per generation
`max_generations`	`int`	0	≥ 0	Stop after N gens (0 = unlimited)
Data				
`output_dir`	`string`	output	—	Root directory for run output
`log_interval`	`int`	1	≥ 1	Generations between CSV rows
`checkpoint_interval`	`int`	0	≥ 0	Checkpoint every N gens (0=off)
Misc				
`target_fps`	`int`	60	> 0	SFML render target FPS
`seed`	`uint64_t`	0	—	RNG seed (0 = random)

Table 4: `SimulationConfig` fields — NEAT Evolution (14 fields)

Field	Type	Def.	Range	Description
Mutation rates				
`mutation_rate`	`float`	0.3	$[0, 1]$	Probability to apply weight mutation
`crossover_rate`	`float`	0.75	$[0, 1]$	Fraction of offspring produced by crossover
`weight_mutation_power`	`float`	0.5	> 0	Std-dev of Gaussian weight perturbation
`add_node_rate`	`float`	0.03	$[0, 1]$	Probability of add-node structural mutation
`add_connection_rate`	`float`	0.05	$[0, 1]$	Probability of add-connection mutation
`max_hidden_nodes`	`int`	50	≥ 0	Cap on hidden node additions
Speciation				
`compatibility_threshold`	`float`	3.0	> 0	δ threshold for species assignment
`c1_excess`	`float`	1.0	≥ 0	Excess gene coefficient (c_1)
`c2_disjoint`	`float`	1.0	≥ 0	Disjoint gene coefficient (c_2)
`c3_weight`	`float`	0.4	≥ 0	Weight difference coefficient (c_3)
`stagnation_limit`	`int`	15	> 0	Generations without improvement before species removal
Fitness weights				
`fitness_survival_weight`	`float`	1.0	≥ 0	Survival time coefficient
`fitness_kill_weight`	`float`	5.0	≥ 0	Kill count coefficient
`fitness_energy_weight`	`float`	0.5	≥ 0	Energy efficiency coefficient
`fitness_distance_weight`	`float`	0.1	≥ 0	Distance traveled coefficient
`complexity_penalty_weight`	`float`	0.01	≥ 0	Genome complexity penalty
Neural network				
`activation_function`	`string`	sigmoid	—	NN activation: sigmoid/tanh/relu
Tick logging				
`tick_log_enabled`	`bool`	false	—	Write per-tick agent state CSV
`tick_log_interval`	`int`	10	≥ 1	Log every N ticks (when enabled)

3.4 `ConfigError` (struct)

```
struct ConfigError {
    std::string field;    // config key that failed validation
    std::string message;  // human-readable error description
};
```

Returned by `validate_config()` to report all validation failures in one pass.

3.5 `CLIArgs` (struct)

Table 5: `CLIArgs` fields

Field	CLI flag / meaning
`config_path`	`--config <path>`
`seed_override`	`--seed <n>` (overrides config)
`headless`	`--headless` — suppress SFML window
`verbose`	`--verbose` — debug-level spdlog output
`help`	`--help`
`no_gpu`	`--no-gpu` — force CPU inference
`max_generations_override`	`--generations <n>`
`resume_path`	`--resume <dir>` — load checkpoint
`checkpoint_interval`	`--checkpoint <n>`
`compare_a`	`--compare-a <dir>` (future use)
`compare_b`	`--compare-b <dir>` (future use)

3.6 Free Functions

Table 6: Core free functions (`config.hpp`)

Signature	Description
`load_config(path: string) → SimulationConfig`	Parse JSON file; throw on missing required fields.
`save_config(config, path: string) → void`	Serialise config to JSON. Used by `Logger` at run start.
`validate_config(config) → vector<ConfigError>`	Check all fields against valid ranges; return all errors found.
`parse_args(argc, argv) → CLIArgs`	Parse POSIX-style command-line arguments.
`print_usage(program_name) → void`	Print help text to stdout.

3.7 `Random` (class)

Single 64-bit Mersenne Twister instance shared by simulation, evolution, and data layers. A single instance per simulation ensures reproducible runs from any given seed.

Table 7: `Random` interface

Method	Description
`Random(seed: uint64_t)`	Construct; seed the MT19937-64 engine
`next_int(min, max)`	Uniform integer in $[min, max]$
`next_float(min, max)`	Uniform float in $[min, max]$
`next_gaussian(mean, stddev)`	Normal distribution sample
`next_bool(probability)`	True with given probability $p \in [0, 1]$
`weighted_select(weights)`	Index proportional to weight values
`shuffle<T>(vec)`	Fisher-Yates in-place shuffle
`sample_indices(total, count)`	Sample *count* distinct indices from $[0, total)$
`seed() const`	Return the construction seed
`engine()`	Direct access for seeding from checkpoints

Private members:

- `seed_ : uint64_t` — stored to support checkpoint save/restore.
- `engine_ : std::mt19937_64` — the underlying generator.

4 Simulation Package (`src/simulation/`)

The simulation package models the physical world: agents, environment, food, spatial acceleration, sensor building, and attack processing. It is intentionally unaware of NEAT or neural networks.

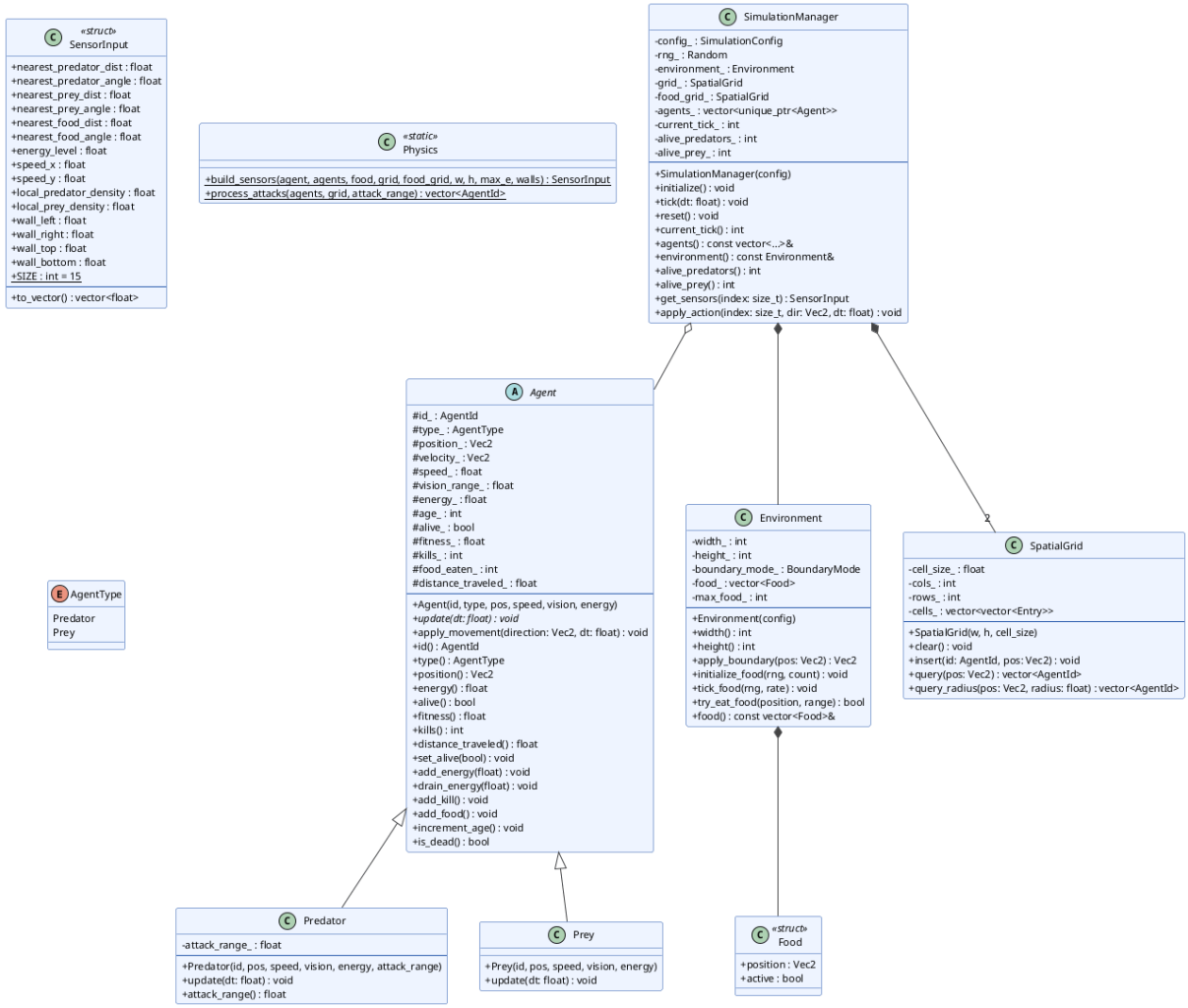

Figure 3: Simulation package class diagram.

4.1 `AgentType` (enum class)

```
enum class AgentType { Predator, Prey };
```

4.2 `Agent` (abstract class)

Base class for all simulated organisms. Stores kinematic state, energy, and fitness accounting.

Table 8: `Agent` protected members

Member	Type	Description
`id_`	`AgentId`	Unique identifier within generation
`type_`	`AgentType`	`Predator` or `Prey`
`position_`	`Vec2`	Current world position
`velocity_`	`Vec2`	Current velocity vector (default {0,0})
`speed_`	`float`	Maximum speed (px/tick)
`vision_range_`	`float`	Sensor detection radius
`energy_`	`float`	Current energy level
`age_`	`int`	Ticks survived (incremented by `increment_age()`)
`alive_`	`bool`	False when energy reaches 0 or killed
`fitness_`	`float`	Accumulated raw fitness value
`kills_`	`int`	Prey eliminated (predators only)
`food_eaten_`	`int`	Food items consumed (prey only)
`distance_traveled_`	`float`	Total distance moved; accumulated in `apply_movement()`

Table 9: `Agent` public interface

Method	Description
`Agent(id,type,position,speed,vision,energy)`	Construct with initial state
`virtual update(dt: float)`	Per-tick update hook (implemented by subclass)
`apply_movement(direction: Vec2, dt: float)`	Move by `direction * speed * dt`; accumulate `distance_traveled_`
`id(), type(), position(), velocity()`	State accessors (const)
`speed(), vision_range(), energy()`	Parameter accessors (const)
`age(), alive(), fitness()`	Lifecycle accessors (const)
`kills(), food_eaten(), distance_traveled()`	Fitness-input accessors (const)
`is_dead()`	`true` when not alive (const)
`set_position(pos: Vec2)`	Teleport (used by boundary enforcement)
`set_alive(alive: bool)`	Mark dead on kill or starvation
`add_fitness(amount: float)`	Accumulate fitness
`add_energy(amount: float)`	Add energy (food, kill)
`drain_energy(amount: float)`	Subtract energy (per-tick drain)
`increment_age()`	Called once per tick by `SimulationManager`
`add_kill()`	Increment kill counter
`add_food()`	Increment food counter

4.3 `Predator : Agent`

```
class Predator : public Agent {
public:
    Predator(AgentId id, Vec2 position, float speed,
             float vision_range, float energy, float attack_range);
    void update(float dt) override;
    float attack_range() const;
private:
    float attack_range_;
};
```

`update()` is a no-op stub; direction is set externally via `SimulationManager::apply_action()`. `attack_range_` is the kill radius checked by `Physics::process_attacks()`.

4.4 `Prey : Agent`

```
class Prey : public Agent {
public:
    Prey(AgentId id, Vec2 position, float speed, float vision_range, float
        energy);
    void update(float dt) override;
};
```

No additional members. Food eating is handled by `Environment::try_eat_food()`.

4.5 `Food` (struct)

```
struct Food {
    Vec2 position;
    bool active = true;  // false = consumed, awaiting respawn
};
```

4.6 `Environment` (class)

Owns the world boundaries and the food array. All geometry is expressed in pixels.

Table 10: `Environment` interface

Method	Description
`Environment(config: const SimulationConfig&)`	Store dimensions and mode
`width() → int`	World size in pixels
`height() → int`	World size in pixels
`boundary_mode() → BoundaryMode`	Clamp or Wrap
`apply_boundary(pos: Vec2) → Vec2`	Clamp: $x \leftarrow \text{clamp}(x, 0, w)$; Wrap: $x \leftarrow \text{fmod}(x + w, w)$
`initialize_food(rng: Random&, count: int)`	Scatter *count* food items uniformly
`tick_food(rng: Random&, rate: float)`	Each inactive food respawns with probability *rate*
`try_eat_food(position: Vec2, range: float) → bool`	Find nearest active food within *range*; deactivate and return `true`
`food() → const vector<Food>&`	Immutable food list for rendering/sensors

Private members: `width_, height_: int`, `boundary_mode_: BoundaryMode`, `food_: vector<Food>`, `max_food_: int`.

4.7 `SensorInput` (struct)

The 15-element observation vector produced by `Physics::build_sensors()` and fed as input to each agent's neural network.

Table 11: `SensorInput` fields

Field	Description
`nearest_predator_dist`	$[-1, 1]$; -1 = not visible; else $1 - d/range$
`nearest_predator_angle`	$[-1, 1]$; $atan2(\Delta y, \Delta x)/\pi$
`nearest_prey_dist`	$[-1, 1]$; same encoding as predator dist
`nearest_prey_angle`	$[-1, 1]$; same encoding as predator angle
`nearest_food_dist`	$[-1, 1]$; for prey agents; -1 if none visible
`nearest_food_angle`	$[-1, 1]$; direction to nearest food
`energy_level`	$[0, 1]$; `energy / initial_energy`
`speed_x`	$[-1, 1]$; current velocity x / max speed
`speed_y`	$[-1, 1]$; current velocity y / max speed
`local_predator_density`	$[0, 1]$; fraction of cells in vision range containing predators
`local_prey_density`	$[0, 1]$; fraction of cells in vision range containing prey
`wall_left`	$[0, 1]$; distance to left boundary / world width (Clamp mode)
`wall_right`	$[0, 1]$; distance to right boundary / world width
`wall_top`	$[0, 1]$; distance to top boundary / world height
`wall_bottom`	$[0, 1]$; distance to bottom boundary / world height

`static constexpr int SIZE = 15;` — must match the number of NN input nodes.
`to_vector() → vector<float>` — returns all fields in declaration order.

4.8 `Physics` (static class)

All methods are `static`. No instance state.

4.8.1 `build_sensors()`

Queries the `SpatialGrid` and `food_grid` for neighbors within `vision_range`. Algorithm:

1. Query agent's cell and 8 neighboring cells in `SpatialGrid`.
2. For each visible agent of opposite type, compute $d =$ distance_to, $\theta =$ atan2$(\Delta y, \Delta x)/\pi$.
3. Encode: *dist_enc* $= 1 - d/$*vision_range*, clamped to $[-1, 1]$.
4. Repeat for food using `food_grid`.
5. Fill wall sensors from `BoundaryMode` and agent position.

4.8.2 `process_attacks()`

For each alive `Predator`, queries `SpatialGrid` with radius `attack_range`. Marks any overlapping alive `Prey` as dead; increments predator kill counter; adds `energy_gain_from_kill`. Returns vector of killed `AgentId` values.

4.9 `SpatialGrid` (class)

Hash-grid for O(1) amortized spatial queries.

```
SpatialGrid(int world_width, int world_height, float cell_size);
```

`cell_size` is set to `vision_range / 2` in `SimulationManager`, ensuring a query of radius `vision_range` covers at most a 5×5 patch of cells.

Table 12: `SpatialGrid` public interface

Method	Description
`clear()`	Erase all entries (called each tick before rebuild)
`insert(id, position)`	Place agent in appropriate cell
`query(position) → vector<AgentId>`	Return all agents in the cell containing *position* plus 8 neighbors
`query_radius(position, radius) → vector<AgentId>`	Filter `query()` results by Euclidean distance

Private: `cells_: vector<vector<Entry>>` where `Entry = {AgentId id; Vec2 pos;}`. Index computed as $cell\_y(y) \times cols + cell\_x(x)$.

4.10 `SimulationManager` (class)

Orchestrates the per-tick simulation loop.

Table 13: `SimulationManager` public interface

Method	Description
`SimulationManager(config)`	Construct; create `Environment` and grids
`initialize()`	Spawn agents at random positions; scatter food
`tick(dt: float)`	One simulation step (see below)
`reset()`	Re-initialize agents and food for new generation
`current_tick() → int`	Ticks elapsed since last reset (const)
`agents() → const vector<unique_ptr<Agent>>&`	All agents (alive and dead)
`environment() → const Environment&`	World environment
`alive_predators() → int`	Cached count of live predators (const)
`alive_prey() → int`	Cached count of live prey (const)
`get_sensors(agent_index) → SensorInput`	Build sensor vector for one agent
`apply_action(agent_index, direction, dt)`	Set velocity and call `apply_movement()`

4.10.1 Per-Tick Sequence (internal)

1. `rebuild_spatial_grid()` — clear and re-insert all alive agents.
2. `rebuild_food_grid()` — same for active food items.
3. Per agent (in `evaluate_generation`): caller calls `get_sensors`, runs NN, calls `apply_action`.

4. `process_energy(dt)` — drain energy; mark zero-energy agents dead.
5. `process_food()` — for each alive prey, call `try_eat_food`.
6. `process_attacks()` — delegate to `Physics::process_attacks()`.
7. `count_alive()` — update `alive_predators_` and `alive_prey_`.
8. Tick food respawn: `environment_.tick_food(rng_, respawn_rate)`.

Private members: `config_: SimulationConfig`, `rng_: Random`, `environment_: Environment`, `grid_: SpatialGrid`, `food_grid_: SpatialGrid`, `agents_: vector<unique_ptr<Agent>>`, `current_tick_: int`, `alive_predators_: int`, `alive_prey_: int`.

5 Evolution Package (`src/evolution/`)

The evolution package implements the NEAT algorithm: genome representation, neural network construction and inference, speciation, mutation, crossover, and the generation evaluation loop.

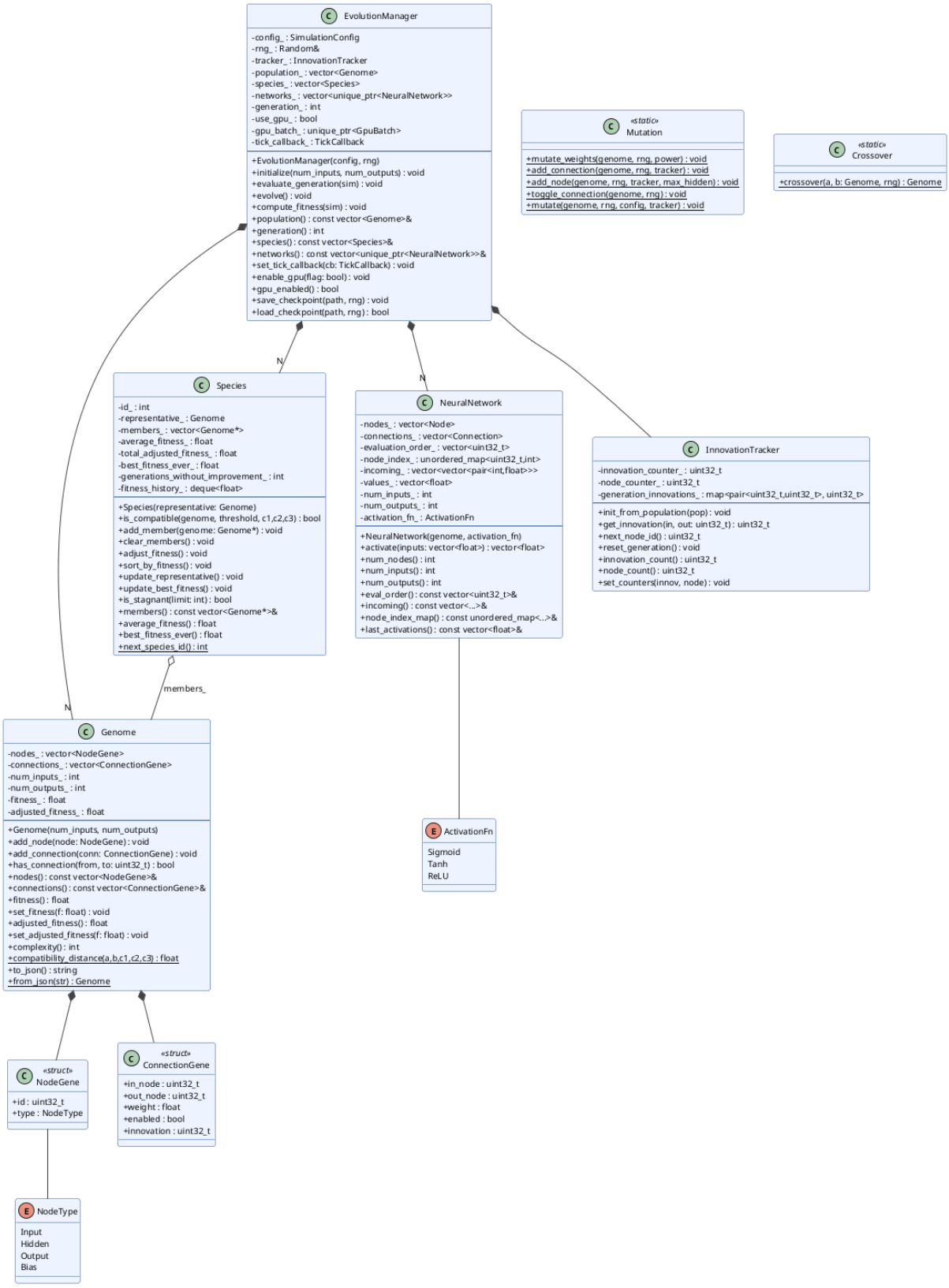

Figure 4: Evolution package class diagram.

5.1 `NodeType` (enum class)

```
enum class NodeType { Input, Hidden, Output, Bias };
```

`Bias` nodes have a fixed activation of 1.0 and appear in every initial genome.

5.2 `NodeGene` and `ConnectionGene` (structs)

Table 14: Gene struct fields

Struct	Field	Type	Description
`NodeGene`	`id`	`uint32_t`	Global node identifier
	`type`	`NodeType`	Input/Hidden/Output/Bias
`ConnectionGene`	`in_node`	`uint32_t`	Source node id
	`out_node`	`uint32_t`	Target node id
	`weight`	`float`	Synaptic weight
	`enabled`	`bool`	False = structurally present but not expressed
	`innovation`	`uint32_t`	Historical marking for alignment during crossover

5.3 `Genome` (class)

Encodes a variable-topology neural network as gene lists.

Table 15: `Genome` interface

Method	Description
`Genome(num_inputs, num_outputs)`	Create minimal genome (inputs + bias + outputs; no hidden)
`add_node(node)`	Append `NodeGene`
`add_connection(conn)`	Append `ConnectionGene`
`has_connection(from, to)`	Check structural connectivity
`has_node(id)`	Membership test
`max_node_id()`	Largest node id in genome
`nodes()`	Gene list accessors
`connections()`	Gene list accessors
`fitness() / set_fitness(f)`	Raw fitness from evaluation
`adjusted_fitness() / set_adjusted_fitness(f)`	Shared fitness (divided by species size)
`num_inputs(), num_outputs()`	Topology constants
`complexity()`	`nodes_.size() + enabled_connection_count`
`compatibility_distance(a,b,c1,c2,c3)`	Static; compatibility metric (see below)
`to_json()`	Serialize to JSON string
`from_json(str)`	Static factory from JSON string

5.3.1 Compatibility Distance Algorithm

Defined by Stanley & Miikkulainen [1]:

$$\delta = \frac{c_1 \cdot E + c_2 \cdot D}{\max(N_1, N_2)} + c_3 \cdot \bar{W}$$

where E = excess gene count, D = disjoint gene count, $\bar{W}$ = mean weight difference of matching genes, and N_i = genome size. Genes are aligned by innovation number.

5.3.2 JSON Serialization Format

```
{
  "nodes": [{"id": 0, "type": "input"}, ...],
  "connections": [
    {"in": 0, "out": 16, "weight": 0.42, "enabled": true, "innovation": 3},
        ...
  ]
}
```

Listing 1: Genome JSON schema

Private members: `nodes_`, `connections_`, `num_inputs_`, `num_outputs_`, `fitness_`, `adjusted_fitness_`.

5.4 `NeuralNetwork` (class)

Builds a runnable feedforward network from a `Genome`.

5.4.1 Construction Algorithm

1. Build `nodes_` list from enabled nodes in genome.
2. Topological sort: Kahn's algorithm on the enabled connection DAG. DFS cycle detection is applied to any new hidden connections before insertion. Result stored in `evaluation_order_` (list of `uint32_t` node ids in activation order).
3. Precompute `incoming_[i]`: for each node at eval-order index i, collect all `(source_idx, weight)` pairs from enabled connections.
4. Allocate `values_` float vector of length `nodes_.size()`.

5.4.2 `activate(inputs)` Algorithm

1. Copy `inputs` into `values_` at input node indices.
2. Set bias node value to 1.0.
3. Iterate `evaluation_order_`:

$$v_i = f\left(\sum_{(j,w)\in incoming[i]} w \cdot v_j\right)$$

   where f is the configured activation function.
4. Return output node values in order.

5.4.3 Activation Functions

```
enum class ActivationFn { Sigmoid, Tanh, ReLU };
```

Table 16: Activation function formulas

Name	Formula	Output range
`Sigmoid`	$1/(1+e^{-x})$	$(0,1)$
`Tanh`	$\tanh(x)$	$(-1,1)$
`ReLU`	$\max(0,x)$	$[0,\infty)$

Table 17: `NeuralNetwork` interface

Method	Description
`NeuralNetwork(genome, activation_fn)`	Construct; run build algorithm
`activate(inputs)`	Forward pass; returns output activations
`num_nodes(), num_connections()`	Topology counts
`num_inputs(), num_outputs()`	I/O sizes
`raw_nodes()`	All node descriptors
`eval_order()`	Activation sequence (for GPU packing)
`incoming()`	Precomputed adjacency
`node_index_map()`	node_id → value index
`last_activations()`	Activations from most recent `activate()` call

5.5 `Species` (class)

Groups genomes by structural similarity. Manages stagnation detection and adjusted fitness for fair reproduction allocation.

Table 18: `Species` private members

Member	Description
`id_`	Unique species id
`representative_`	Reference genome for compatibility tests
`members_`	Non-owning pointers into `EvolutionManager::population_`
`average_fitness_`	Mean raw fitness of members
`total_adjusted_fitness_`	Sum of adjusted fitnesses
`best_fitness_ever_`	Highest fitness seen since species creation
`generations_without_improvement_`	Incremented each generation max fitness does not improve
`fitness_history_`	Rolling history of average fitness

5.5.1 Key Algorithms

`adjust_fitness()`: Divides each member's raw fitness by species size (explicit fitness sharing):

$$f_i' = f_i / |members|$$

`is_stagnant(limit)`: Returns `generations_without_improvement_ >= limit`.

`update_best_fitness()`: If the current maximum member fitness exceeds

`best_fitness_ever_`, update it and reset the stagnation counter; otherwise increment it.

`update_representative()`: Set `representative_` to the highest-fitness member (used for next-generation compatibility tests).

5.6 `InnovationTracker` (class)

Ensures historical markings are consistent within a generation, enabling correct NEAT crossover alignment.

Table 19: `InnovationTracker` interface

Method	Description
`init_from_population(population)`	Scan all genomes; set counters to max innovation / node ids found
`get_innovation(in, out)`	Look up or create innovation for edge (in→out) within this generation
`next_node_id()`	Return and increment the global node counter
`reset_generation()`	Clear `generation_innovations_` (called at start of `evolve()`)
`innovation_count(), node_count()`	Current counter values
`set_counters(innov, node)`	Restore from checkpoint

Private: `generation_innovations_: map<pair<uint32_t,uint32_t>, uint32_t>` — the within-generation map that guarantees the same structural mutation receives the same innovation number across all genomes in one generation.

5.7 `Mutation` (static class)

All methods are `static`.

Table 20: `Mutation` operator algorithms

Method	Algorithm
`mutate_weights(genome, rng, power)`	For each connection: with probability 0.9 apply Gaussian perturbation $w \leftarrow w + \mathcal{N}(0, power)$; with probability 0.1 replace with uniform sample from $[-2, 2]$.
`add_connection(genome, rng, tracker)`	Sample random unconnected node pair; check no cycle (DFS); assign innovation via `tracker.get_innovation()`; initialize weight $\sim \mathcal{U}(-1, 1)$.
`add_node(genome, rng, tracker, max_hidden)`	Select random enabled connection; disable it; insert new hidden node (`tracker.next_node_id()`); add connection `in` → `new` with weight 1.0; add connection `new` → `out` with old weight.
`toggle_connection(genome, rng)`	Pick random connection; flip `enabled` flag.
`mutate(genome, rng, config, tracker)`	Apply each operator independently: `mutate_weights` with `mutation_rate`, `add_node` with `add_node_rate`, `add_connection` with `add_connection_rate`, `toggle_connection` at a fixed low rate.

5.8 `Crossover` (static class)

```
static Genome crossover(const Genome& parent_a,
                        const Genome& parent_b, Random& rng);
```

Algorithm: Align genes by innovation number.

1. **Matching genes:** Pick from either parent with probability 0.5.
2. **Disjoint / excess genes:** Always inherit from the fitter parent.
3. **Disabled genes:** If disabled in either contributing parent, disable in child with probability 0.75.
4. Inherit `num_inputs` and `num_outputs` from the fitter parent.

5.9 `EvolutionManager` (class)

Top-level controller for NEAT: holds the population, networks, species, and orchestrates the evaluate → compute_fitness → evolve cycle.

Table 21: `EvolutionManager` private members

Member	Description
`config_`	Run parameters
`rng_`	Shared RNG (owned by main)
`tracker_`	Innovation numbering
`population_`	Full population by value
`species_`	Current species list
`networks_`	One NN per genome
`generation_`	Current generation index
`num_inputs_, num_outputs_`	Determined at `initialize()`
`use_gpu_`	GPU inference enabled
`gpu_batch_`	Device memory (CUDA build only)
`tick_callback_`	Optional per-tick hook

Table 22: `EvolutionManager` public interface

Method	Description
`EvolutionManager(config, rng)`	Store config and RNG reference
`initialize(num_inputs, num_outputs)`	Create minimal genomes; build networks
`evaluate_generation(sim)`	Run one generation (see below)
`evolve()`	Speciate → remove stagnant → reproduce
`compute_fitness(sim)`	Write fitness into each genome (see below)
`population()`	Access genome vector
`generation()`	Current generation number
`species()`	Current species list
`networks()`	Built network list
`set_tick_callback(cb)`	Install `TickCallback`
`clear_tick_callback()`	Remove callback
`enable_gpu(flag)`	Toggle GPU inference path
`gpu_enabled()`	Query current GPU state
`save_checkpoint(path, rng)`	Serialize to JSON (see below)
`load_checkpoint(path, rng)`	Restore from checkpoint; false if not found

5.9.1 `evaluate_generation()` Algorithm

1. Call `sim.reset()` and `sim.initialize()`.
2. For each tick $t \in [0, \textit{generation\_ticks})$:
   a. For each alive agent i: call `sim.get_sensors(i)`; run `networks_[i]->activate()`; decode output to direction vector; call `sim.apply_action(i, direction, dt)`.
   b. Call `sim.tick(dt)`.
   c. If `tick_callback_` is set, invoke it with $(t, \textit{sim})$.
3. On GPU path: `gpu_batch_->pack_inputs()` before agent loop; `batch_neural_inference()` launches CUDA kernel; `gpu_batch_->unpack_outputs()` retrieves results.

5.9.2 `compute_fitness()` Algorithm

For each genome i paired with `agents_[i]`:

$$f_i = w_s \cdot \frac{age}{T} + w_k \cdot kills + w_e \cdot \frac{energy}{E_0} + w_d \cdot \frac{dist}{d_{\max}} - w_c \cdot complexity$$

where T = `generation_ticks`, E_0 = `initial_energy`, $d_{\max} = \sqrt{w^2 + h^2}$, and the w_* constants come from `SimulationConfig`.

5.9.3 `evolve()` Algorithm

1. `speciate()`: For each genome, find the first compatible species (representative comparison). No match → new species. Update `update_best_fitness()` on each species.
2. `remove_stagnant_species()`: Remove species where `is_stagnant(stagnation_limit)` is true (except the top-fitness species).
3. Compute offspring allocation via largest-remainder method proportional to total adjusted fitness.
4. `reproduce()`: For each species, elitism (top genome survives); remainder via `Crossover::crossover` + `Mutation::mutate` using tournament selection within species.
5. `tracker_.reset_generation()`.
6. `build_networks()`.
7. Increment `generation_`.

5.9.4 Checkpoint Format

JSON file with fields: `generation`, `population` (array of genome JSON), `rng_state` (hex string of engine state), `species_stagnation` (array of `{id, best_ever, gens_no_improvement}`), `innovation_counter`, `node_counter`.

5.9.5 `TickCallback` Type Alias

```
using TickCallback =
    std::function<void(int tick, const SimulationManager& sim)>;
```

`main.cpp` uses this to wire `Logger::log_tick`, live visualization updates, and per-generation benchmarking probes into the evaluation loop without creating any dependency from `evolution/` on `data/` or `visualization/`.

6 Visualization Package (`src/visualization/`)

The visualization package renders the live simulation using SFML. It is always compiled in; when the binary is run with `--headless` the window is simply not created.

Figure 5: Visualization package class diagram.

6.1 `Renderer` (class)

Stateless drawing primitives for the world view. Reuses pre-allocated SFML shapes (`circle_`, `triangle_`, `rect_`) to avoid per-frame allocation.

Table 23: `Renderer` public interface

Method	Description
`draw_background(target, width, height)`	Fill background with dark color
`draw_grid(target, width, height, cell_size)`	Draw spatial-grid overlay (when `show_grid`)
`draw_boundaries(target, width, height)`	Draw world-edge rectangle
`draw_food(target, food)`	Small green circles for active food
`draw_agent(target, agent, selected)`	Triangle (predator) or circle (prey)
`draw_vision_range(target, agent)`	Dashed circle at `vision_range`
`draw_sensor_lines(target, agent, agents, food)`	Lines to nearest predator/prey/food
`species_color(species_id)`	Deterministic hue (see below); static

Public flags: `show_grid`, `show_vision`, `show_sensors` (`bool`); `dead_fade_alpha` (`float`).

6.1.1 Species Color Algorithm

Hue for species n:

$$H = (n \times 137.508^\circ) \bmod 360^\circ$$

The golden angle (137.508°) ensures adjacent species IDs produce maximally dissimilar hues. Saturation = 1.0, Value = 0.9 (HSV). Converted to RGB via standard HSV-to-RGB formula.

6.2 `OverlayStats` (struct)

All data needed by `UIOverlay::draw()`.

Table 24: `OverlayStats` fields

Field	Description
`generation`	Current generation index
`tick`	Tick within current generation
`alive_predators`	Live predator count
`alive_prey`	Live prey count
`best_fitness`	Best fitness in current generation
`avg_fitness`	Mean fitness in current generation
`num_species`	Number of active species
`fps`	Measured frames per second
`speed_multiplier`	Simulation speed multiplier (1–16)
`paused`	Simulation is paused
`fast_forward`	`[FF]` HUD label when true
`selected_agent`	Index of selected agent (-1 = none)
`selected_energy`	Energy of selected agent
`selected_age`	Age of selected agent
`selected_kills`	Kill count of selected agent
`selected_food_eaten`	Food eaten by selected agent
`selected_fitness`	Fitness of selected agent
`selected_genome_complexity`	Node + connection count of selected genome

6.3 `UIOverlay` (class)

Renders HUD panels, fitness chart, and neural network visualization.

Table 25: `UIOverlay` public interface

Method	Description
`initialize(font_path)`	Load font from path (or fallback); return success
`draw(target, stats, genome*)`	Render all overlay elements
`has_font()`	Whether font loaded successfully
`push_fitness(best, avg)`	Append to rolling history deques
`set_activations(vals)`	Store {node_id → activation} for NN panel

Private members: `best_history_` and `avg_history_`: `deque<float>` capped at `CHART_MAX_POINTS = 150`. `node_activations_`:

`unordered_map<uint32_t, float>`. `font_`: `sf::Font`; `font_loaded_`: `bool`.

6.3.1 `draw_nn_panel()` Algorithm

Layout:

1. Arrange input nodes on the left column, output nodes on the right, hidden nodes in intermediate columns ordered by their position in `eval_order`.
2. For each node, draw a circle coloured by activation value:

$$v \leq -1 \Rightarrow \text{blue}(30, 30, 200);$$
$$v = 0 \Rightarrow \text{gray}(180, 180, 180);$$
$$v \geq 1 \Rightarrow \text{orange}(220, 120, 20)$$

   Intermediate values interpolated via `std::clamp`.
3. Draw enabled connections as lines; weight modulates line thickness.

6.3.2 `draw_fitness_chart()` Algorithm

Renders two polylines over the rightmost panel area: `best_history_` in orange and `avg_history_` in yellow. Y-axis auto-scales to $[\min(\textit{history}), \max(\textit{history})]$.

6.4 `VisualizationManager` (class)

Owns the SFML window, camera, input handling, and coordinates rendering.

Table 26: `VisualizationManager` private members

Member	Description
`config_`	Run parameters
`window_`	SFML window (null in headless)
`camera_view_`	Zoomable/pannable world camera
`renderer_`	World drawing primitives
`overlay_`	HUD overlay
`overlay_stats_`	Updated each `render()` call
`paused_`	Simulation pause state
`fast_forward_`	FF mode (skip rendering)
`speed_multiplier_`	Ticks per rendered frame (1–16)
`selected_agent_`	Index of clicked agent (-1)
`zoom_level_`	Camera zoom $\in [0.1, 10.0]$
`dragging_`	Right-click pan active

Table 27: `VisualizationManager` public interface

Method	Description
`initialize()`	Create SFML window; load font
`render(sim, evolution)`	Full frame: clear → draw world → draw HUD → display
`should_close()`	Window closed or Escape pressed
`handle_events()`	Process SFML event queue
`is_paused(), speed_multiplier()`	Input state queries
`should_reset() / clear_reset()`	Reset-requested flag
`should_step() / clear_step()`	Single-step flag
`is_fast_forward() / clear_fast_forward()`	FF flag
`selected_agent()`	Selected agent index
`set_selected_activations(vals, idx_map)`	Invert `node_index_map` → {id→activation}; forward to overlay
`push_fitness(best, avg)`	Delegate to `overlay_.push_fitness()`

6.4.1 Key Bindings

Table 28: Keyboard/mouse event map

Input	Action	Flag set
`Space`	Toggle pause	`paused_`
`R`	Request simulation reset	`reset_requested_`
`N` (step)	Advance one tick while paused	`step_requested_`
`H`	Toggle fast-forward mode	`fast_forward_`
`+` / `-`	Increase/decrease speed multiplier	`speed_multiplier_`
`G`	Toggle spatial-grid overlay	`renderer_.show_grid`
`V`	Toggle vision-range circles	`renderer_.show_vision`
`S`	Toggle sensor lines	`renderer_.show_sensors`
Mouse wheel	Zoom camera	`zoom_level_`
Right-click drag	Pan camera	`dragging_`
Left-click	Select nearest agent	`selected_agent_`
`Escape`	Close window	`running_`

6.4.2 Camera Model

`zoom_level_` is clamped to $[0.1, 10.0]$. The `sf::View` is scaled reciprocally: a zoom of 2.0 halves the viewport area. Right-click drag stores the cursor position at button-press (`drag_start_`) and updates the view center by the delta each `MouseMoved` event.

7 Data Package (`src/data/`)

The data package collects, aggregates, and persists simulation metrics.

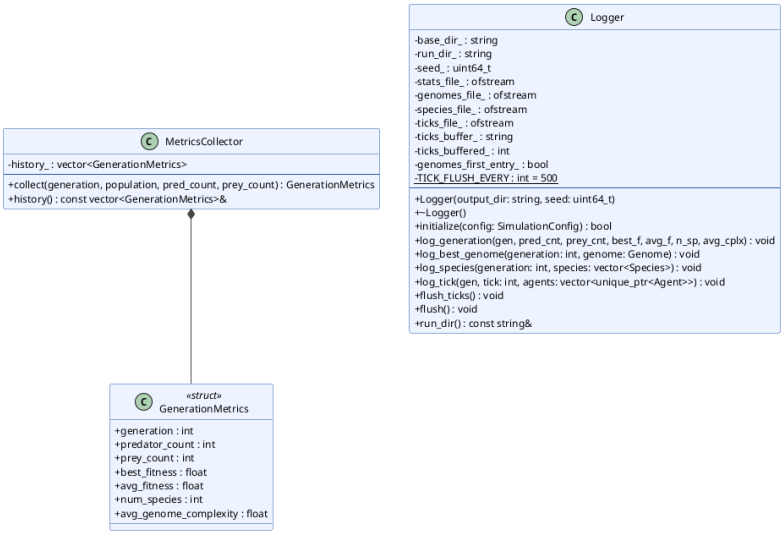

Figure 6: Data package class diagram.

7.1 `GenerationMetrics` (struct)

Plain value type returned by `MetricsCollector::collect()`.

Table 29: `GenerationMetrics` fields

Field	Description
`generation`	Generation index
`predator_count`	Alive predators at end of generation
`prey_count`	Alive prey at end of generation
`best_fitness`	Maximum genome fitness in population
`avg_fitness`	Mean genome fitness in population
`num_species`	Active species count
`avg_genome_complexity`	Mean `Genome::complexity()` across population

7.2 `MetricsCollector` (class)

Table 30: `MetricsCollector` interface

Method	Description
`collect(generation, population, pred_count, prey_count)`	Iterate `population` to find max/avg fitness and avg complexity; append to `history_`; return metrics struct.
`history()`	All metrics collected since construction.

Private: `history_: vector<GenerationMetrics>`.

7.3 `Logger` (class)

Writes all run output to `output/[YYYYMMDD_HHMMSS_seedN]/`.

Table 31: `Logger` interface

Method	Description
`Logger(output_dir, seed)`	Store base dir and seed; do not open files yet
`initialize(config)`	Create run directory; open all files; write headers; save `config.json`
`log_generation(...)`	Append one row to `stats.csv`
`log_best_genome(generation, genome)`	Append JSON entry to `genomes.json`
`log_species(generation, species)`	Append rows to `species.csv`
`log_tick(generation, tick, agents)`	Append rows to `ticks_buffer_`; auto-flush every 500 rows
`flush_ticks()`	Write `ticks_buffer_` to `ticks.csv` and reset buffer
`flush()`	Flush all open file streams
`run_dir()`	Path to the created run directory

Private members: `base_dir_`, `run_dir_`: `string`; `seed_`: `uint64_t`; `stats_file_`, `genomes_file_`, `species_file_`, `ticks_file_`: `std::ofstream`; `ticks_buffer_`: `string` (accumulated CSV rows); `ticks_buffered_`: `int`; `genomes_first_entry_`: `bool` (for JSON comma handling); `TICK_FLUSH_EVERY = 500`: `constexpr int`.

7.3.1 Output File Formats

`config.json` Full `SimulationConfig` snapshot, written by `Logger::initialize()`.

`stats.csv`

```
generation,predator_count,prey_count,best_fitness,avg_fitness,num_species,
   avg_complexity
0,50,150,12.34,5.67,8,16.2
...
```

`species.csv`

```
generation,species_id,size,avg_fitness,stagnation_count
0,1,42,8.1,0
...
```

`genomes.json` JSON array of best-genome snapshots:

```
[
  {"generation": 0, "genome": {
    "nodes": [{"id": 0, "type": "input"}, ...],
    "connections": [{"in": 0, "out": 16, "weight": 0.42,
                     "enabled": true, "innovation": 3}, ...]
```

```
  }},
  ...
]
```

`ticks.csv` (optional) Written when `tick_log_enabled: true`:

```
generation,tick,agent_id,type,alive,x,y,energy,kills,food_eaten
0,0,0,predator,1,312.5,201.3,100.0,0,0
...
```

Rows are buffered in `ticks_buffer_` (string) and flushed every `TICK_FLUSH_EVERY = 500` rows to amortize `write()` syscall cost.

8 GPU Package (`src/gpu/`)

The GPU package accelerates neural network inference and fitness evaluation on NVIDIA hardware. It is compiled only when `nvcc` is detected at CMake configuration time; if unavailable (or if `--no-gpu` is passed), `EvolutionManager` falls back to the CPU path transparently.

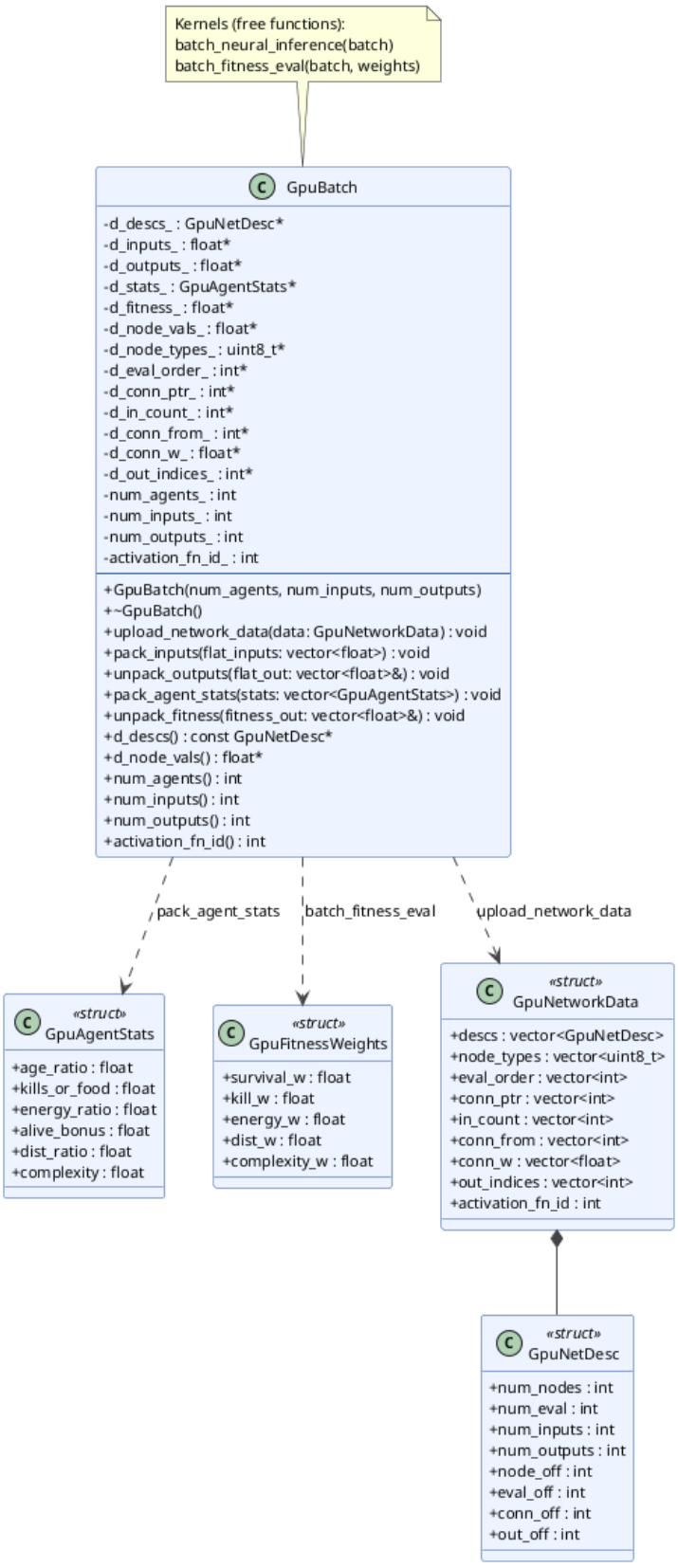

Figure 7: GPU package class diagram.

8.1 `GpuNetDesc` (struct)

Per-agent descriptor into the flat CSR arrays. Eight integer offsets into globally packed device arrays.

Table 32: `GpuNetDesc` fields

Field	Description
`num_nodes`	Total node count for this agent
`num_eval`	Length of the evaluation-order slice
`num_inputs`	Number of input nodes
`num_outputs`	Number of output nodes
`node_off`	Start index in `d_node_types[]`
`eval_off`	Start index in `d_eval_order[]`
`conn_off`	Start index in `d_conn_ptr[]` / `d_conn_w[]`
`out_off`	Start index in `d_out_indices[]`

8.2 CSR Layout Diagram

Each agent i is packed contiguously. Figure 8 illustrates the layout.

Agent 0			Agent 1			
$n_0, n_1, \ldots$	$e_0, e_1, \ldots$	$w_0, w_1, \ldots$	$n'_0, n'_1, \ldots$	$e'_0, e'_1, \ldots$	$w'_0, w'_1, \ldots$	$\ldots$
`node_types`	`eval_order`	`conn_w`	`node_types`	`eval_order`	`conn_w`	

Figure 8: CSR flat layout: agents packed consecutively, offsets stored in `GpuNetDesc`.

The advantage over a padded layout: with the typical NEAT topology of 15 inputs, 2 outputs, and 0–5 hidden nodes in early generations, a padded `MAX_NODES`^{2} weight matrix would waste up to 10× VRAM compared with CSR.

8.3 `GpuAgentStats` and `GpuFitnessWeights` (structs)

Table 33: `GpuAgentStats` fields (per-agent stats for fitness evaluation on device)

Field	Description
`age_ratio`	`age / generation_ticks`
`kills_or_food`	Kill count (predator) or food eaten (prey)
`energy_ratio`	`energy / initial_energy`
`alive_bonus`	1.0 if still alive at generation end, else 0.0
`dist_ratio`	`distance_traveled / max_possible_dist`
`complexity`	`Genome::complexity()` as float

Table 34: `GpuFitnessWeights` fields

Field	Type	Source in config
`survival_w`	`float`	`fitness_survival_weight`
`kill_w`	`float`	`fitness_kill_weight`
`energy_w`	`float`	`fitness_energy_weight`
`dist_w`	`float`	`fitness_distance_weight`
`complexity_w`	`float`	`complexity_penalty_weight`

8.4 `GpuNetworkData` (struct)

Host-side staging buffer assembled by `EvolutionManager` before `GpuBatch::upload_network_data()`.

Table 35: `GpuNetworkData` fields

Field	Description
`descs`	One descriptor per agent
`node_types`	Packed node type bytes (all agents)
`eval_order`	Topological order indices (all agents)
`conn_ptr`	CSR row pointer (start of each node's connections)
`in_count`	Number of incoming connections per node
`conn_from`	Source local-index for each connection
`conn_w`	Weight for each connection
`out_indices`	Local indices of output nodes
`activation_fn_id`	0=Sigmoid, 1=Tanh, 2=ReLU

8.5 `GpuBatch` (class, RAII)

RAII wrapper for all CUDA device memory required for one generation. Copy constructor and copy assignment are = `delete`; move semantics could be added if needed.

```
GpuBatch(int num_agents, int num_inputs, int num_outputs);
~GpuBatch();  // cudaFree all device pointers
```

Table 36: `GpuBatch` public interface

Method	Description
`upload_network_data(data)`	`cudaMemcpy` all fields of `GpuNetworkData` to device arrays. Called once per generation (topology changes with `evolve()`).
`pack_inputs(flat_inputs)`	Copy flat $N_{agents} \times N_{inputs}$ input matrix to `d_inputs_`.
`unpack_outputs(flat_out)`	Copy $N_{agents} \times N_{outputs}$ result from `d_outputs_` to host.
`pack_agent_stats(stats)`	Upload `vector<GpuAgentStats>` to `d_stats_`.
`unpack_fitness(fitness_out)`	Download fitness results from `d_fitness_`.
`d_descs(), d_node_vals(), …`	Raw device pointer accessors (passed to kernel launch)
`num_agents(), num_inputs(), num_outputs()`	Dimension queries
`activation_fn_id()`	Activation function id for kernel

Private device pointers: `d_descs_: GpuNetDesc*`, `d_inputs_: float*`, `d_outputs_: float*`, `d_stats_: GpuAgentStats*`, `d_fitness_: float*`, `d_node_vals_: float*`, `d_node_types_: uint8_t*`, `d_eval_order_: int*`, `d_conn_ptr_: int*`, `d_in_count_: int*`, `d_conn_from_: int*`, `d_conn_w_: float*`, `d_out_indices_: int*`.

8.6 CUDA Kernel Algorithms

8.6.1 `neural_forward_kernel`

Launch configuration: `(num_agents, 1)` blocks × `(1, 1)` threads — one CUDA thread per agent.

1. Read `GpuNetDesc d = d_descs[agent_id]`.
2. Initialize `d_node_vals[d.node_off..]` from `d_inputs[agent_id * num_inputs..]`.
3. Set bias node value to 1.0.
4. Iterate `eval_order[d.eval_off..d.eval_off+d.num_eval]`: for each node index n, compute

$$v[n] = f\left(\sum_{k=0}^{in\_count[n]-1} d\_conn\_w[\textit{conn\_ptr}[n]+k] \cdot v[d\_conn\_from[\textit{conn\_ptr}[n]+k]]\right)$$

5. Write output values from `d_out_indices` to `d_outputs[agent_id * num_outputs..]`.

Activation function dispatched via `activation_fn_id`: 0=sigmoid, 1=tanh, 2=ReLU — evaluated with a device-side switch to avoid function-pointer overhead.

8.6.2 `fitness_eval_kernel`

Launch configuration: one thread per agent.

1. Read `GpuAgentStats s = d_agent_stats[agent_id]`.
2. Compute:

$$f = w_s \cdot s.\textit{age\_ratio} + w_k \cdot s.\textit{kills\_or\_food} + w_e \cdot s.\textit{energy\_ratio} + w_d \cdot s.\textit{dist\_ratio} - w_c \cdot s.\textit{complexity}$$

3. Write to `d_fitness_out[agent_id]`.

8.6.3 Host Launch Functions

```
namespace moonai::gpu {
    void batch_neural_inference(GpuBatch& batch);
    void batch_fitness_eval(GpuBatch& batch, GpuFitnessWeights weights);
}
```

These functions select grid/block dimensions, launch the corresponding kernels, and call `cudaDeviceSynchronize()` before returning.

9 Glossary

`Activation function` A non-linear function applied to the weighted sum of inputs at each neural network node. MoonAI supports *Sigmoid*, *Tanh*, and *ReLU*.

`Adjusted fitness` A genome's raw fitness divided by its species size (explicit fitness sharing). Prevents any single large species from dominating reproduction.

`AgentId` `using AgentId = std::uint32_t`. Unique identifier assigned to each agent at generation start; stable within one generation.

`CSR` Compressed Sparse Row — a memory layout that packs variable-length per-agent network data into flat arrays with offset descriptors (`GpuNetDesc`), eliminating VRAM waste from padding.

`CUDA` NVIDIA's parallel computing platform. MoonAI uses CUDA to accelerate NN inference (`neural_forward_kernel`) and fitness evaluation (`fitness_eval_kernel`).

`eval_order` The topological sort of a network's nodes produced by Kahn's algorithm at `NeuralNetwork` construction time. Stored as a `vector<uint32_t>` of node ids in activation order.

`Fast-forward mode` An optional rendering mode (**H** key) where the visualization

	skips drawing frames, allowing the simulation to run at maximum CPU/GPU throughput. The HUD displays `[FF]` when active.
FPS	Frames Per Second — the rendering rate measured by `VisualizationManager`. Target set by `target_fps` in config.
GpuBatch	RAII class that owns all CUDA device memory for one generation's population. Allocates on construction; frees via `cudaFree` in destructor.
GpuNetDesc	Eight-integer struct describing one agent's network within the CSR flat arrays (offsets: `node_off`, `eval_off`, `conn_off`, `out_off`; counts: `num_nodes`, `num_eval`, `num_inputs`, `num_outputs`).
Golden angle	$137.508° = 360°(1 - 1/\phi)$ where ϕ is the golden ratio. Used by `Renderer::species_color()` to generate maximally distinct hues for consecutive species IDs.
HLD	High-Level Design document (December 2025) — conceptual architecture preceding this LLD.
IEEE 1016-2009	IEEE Standard for Software Design Descriptions. This document follows the structure defined therein.
InnovationTracker	Per-generation map from connection pairs `(in_node, out_node)` to innovation numbers. Guarantees that the same structural mutation receives the same historical marking within a generation, enabling correct NEAT crossover.
LLD	Low-Level Design — this document.
MT19937	Mersenne Twister 19937 — the pseudorandom number generator algorithm. MoonAI uses the 64-bit variant `std::mt19937_64` via the `Random` class.
NEAT	NeuroEvolution of Augmenting Topologies — the evolutionary algorithm that simultaneously evolves neural network weights and topology. Reference: Stanley & Miikkulainen (2002) [1].
NN	Neural Network — a `NeuralNetwork` object built from a `Genome`.
RAII	Resource Acquisition Is Initialization — C++ design idiom ensuring deterministic resource lifetime via constructors and destructors.

RNG Random Number Generator. In MoonAI, one `Random` instance is shared per simulation for full reproducibility.

SensorInput 15-element observation vector produced by `Physics::build_sensors()`. Encodes distances and angles to nearest predator, prey, and food; energy level; velocity components; density counts; and wall proximity.

SFML Simple and Fast Multimedia Library — used for window creation, 2-D rendering, and event handling in the visualization package.

SpatialGrid Hash-grid partitioning the world into cells of size `vision_range/2`. Enables O(1) amortized neighbor queries for sensor building and attack detection.

Species stagnation A species that has shown no improvement in best-ever fitness for `stagnation_limit` consecutive generations is removed by `remove_stagnant_species()`, except the current top-fitness species.

TickCallback `std::function<void(int, const SimulationManager&)>` — optional hook installed on `EvolutionManager` to receive per-tick notifications (for logging, visualization updates, and benchmarking).

Topological sort Total ordering of a DAG's nodes such that every edge (u, v) has u before v. Computed by Kahn's algorithm in `NeuralNetwork` constructor to determine activation evaluation order.

UML Unified Modeling Language. Version 2.5 class and package diagrams are used throughout this document.

Vec2 `struct Vec2 { float x, y; }` — the 2-D coordinate type used everywhere in simulation physics.

VRAM Video RAM — device memory on the GPU. CSR layout minimizes VRAM usage relative to padded alternatives.

References

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

[2] SFML Development Team, *SFML — Simple and Fast Multimedia Library*, version 2.6, 2024. [Online]. Available: https://www.sfml-dev.org

[3] N. Lohmann, *JSON for Modern C++*, version 3.11, 2024. [Online]. Available: https://github.com/nlohmann/json

[4] G. Ronen, *spdlog — Fast C++ logging library*, version 1.12, 2024. [Online]. Available: https://github.com/gabime/spdlog

[5] Google LLC, *GoogleTest — Google Testing and Mocking Framework*, 2024. [Online]. Available: https://github.com/google/googletest

[6] Microsoft Corporation, *vcpkg — C/C++ Dependency Manager*, 2024. [Online]. Available: https://github.com/microsoft/vcpkg

[7] Kitware Inc., *CMake Build System*, version 3.21+, 2024. [Online]. Available: https://cmake.org

[8] NVIDIA Corporation, *CUDA C++ Programming Guide*, version 12.x, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] A. Paradis, *PlantUML — Open-source UML diagram tool*, 2024. [Online]. Available: https://plantuml.com

[10] ISO/IEC, *ISO/IEC 14882:2017 — Programming Languages — C++17 Standard*, International Organization for Standardization, Geneva, 2017.

[11] IEEE, *IEEE Std 1016-2009 — IEEE Standard for Information Technology — Systems Design — Software Design Descriptions*, IEEE, New York, 2009.