The Ecosystem Safari Simulator: A Technical Manual for Use as a Conceptual Agent-Based Model (ABM)

Section 1: Model Overview & Typology

1.1 Model Definition: Conceptual Agent-Based Model (ABM)

The Ecosystem Safari Simulator is a Conceptual Agent-Based Model (ABM) designed to simulate the population dynamics and interactions within a simplified savanna ecosystem. The model's architecture, as defined in the source code, adheres to the classic components of an ABM:

- **Agents:** The model's primary actors are autonomous, discrete "agents" representing individual animals. These agents are instantiated as JavaScript objects and stored within the global animals = [] array. Each agent possesses a heterogeneous set of internal state variables (e.g., species, state, hunger, matingCooldown, speed) that govern its behaviour.
- **Agent Factory:** New agents are instantiated and parameterised via the createBlockyAnimal(species) function. This function acts as an "agent factory," assigning a unique, hard-coded "genetic" profile (e.g., speed, radius, baseMatingCooldown) based on the species argument provided during creation or reproduction.
- **Scheduler:** The model operates on a discrete-time-step scheduler. This is managed by the main animate() loop, which calculates a deltaTime (time elapsed per frame) and subsequently calls the updateAnimals(deltaTime) function. This "update" function iterates through the entire animals array, executing the behavioural logic for each agent sequentially in every simulation tick.
- **Environment:** Agents interact with a spatially explicit, continuous (non-grid-based) 3D environment. This "world" is defined by a hard-coded area (const tankSize = 50) and is populated with static obstacles (obstacles = [], instantiated by createMountainsAndWater) and dynamic resources (grazingPatches = [], trees = []) that agents can interact with and deplete.

1.2 Core Objective: Systemic "Thought Experiment"

The simulation's core purpose is explicitly stated in its introduction modal (#introductionModal) within the : "🎯 **Your Objective:** Maintain a stable and balanced savanna ecosystem for as long as possible. Prevent extinctions and avoid overpopulation that depletes resources!".

This objective frames the model not as a "game" with a win-condition, but as a "thought experiment" in systemic management. The user's primary goal is to achieve **homeostasis** (a stable, balanced system) and avoid **collapse**. The model's value as a research tool lies in its demonstration of systemic behaviour, non-linear dynamics, and feedback loops.

The inclusion of user-controlled "Policy" sliders (e.g., predationRate, herbivoreMatingRate)

transforms the user from a "player" into a "policy-maker." The simulation thus becomes a conceptual laboratory for testing hypotheses about the systemic impact of different management strategies (e.g., "What is the systemic effect of a policy that encourages high predator reproduction?").

Furthermore, the model explicitly simulates the ethical and practical trade-offs in conservation through the "Hunter" intervention logic. This subsystem models the critical distinction between sustainable "culling" (governed by quotas and management cycles) and destabilising "poaching" (which triggers social and legal consequences), providing a direct model of a coupled Social-Ecological System (SES).

1.3 Key Components

The model's architecture, as defined in the source code, is comprised of four distinct component classes:

1. **Environment (The "World" & Resources):** This class includes all static and dynamic environmental components and their "physics."
    - **Instantiation Functions:** createEnvironmentBase, createMountainsAndWater, createGrazingPatches, createBlockyTree, createClouds.
    - **Update Loops:** updateGrazingPatches(deltaTime), updateTrees(deltaTime).
2. **Agents (Autonomous Animals):** This class includes the autonomous entities that exhibit behaviour within the environment.
    - **Instantiation Functions:** createBlockyAnimal, spawnOffspring.
    - **Update Loop:** updateAnimals(deltaTime).
3. **External Interventions (User Controls):** This class includes the mechanisms by which the user, as an external force, perturbs the system's parameters and state.
    - **Policy (Passive) Controls:** setupUIEventListeners() translates slider inputs into global physics modifiers (e.g., predationRateFactor, herbivoreMatingCooldownFactor).
    - **Direct (Active) Intervention:** spawnHunter(), fireWeapon(), onHunterMove(), updateHunter(deltaTime).
    - **Intervention Consequences:** checkHuntingConsequences(), showFailureMessage("ARRESTED").
4. **Data Logger (Quantitative Output):** This class runs parallel to the simulation loop, acting as a silent observer that records the system state at fixed intervals.
    - **Sampling Rate:** 1.0 Simulated Day (approx. every 20 real-time seconds).
    - **Output Format:** CSV (Comma Separated Values), generated client-side upon simulation termination.
    - **Functions:** logSimulationStep(), exportToCSV().

Section 2: The "Physics" of the Savanna: Environmental & Resource Parameters

All environmental rules are hard-coded as global constants within the primary <script> block. These values define the immutable "physics" of the simulated world.

2.1 Spatial Environment

- **World Size:** The spatial extent of the environment is defined by const tankSize = 50. As the world is centred at (0,0,0), this variable defines a total simulation area of 100x100 units, extending from -50 to +50 on the X and Z axes.
- **Boundary Conditions:** Agent movement is constrained by const finalBoundary = tankSize * 0.98. This variable (value: 49) is used in the updateAnimals function as a hard "reflecting" boundary. Any agent attempting to move past this coordinate is clamped to it. This creates a "walled garden" or "finite reserve" ecosystem, preventing agent emigration or immigration.

2.2 Resource Model 1: Grass (Grazing Patches)

The grazing resource model is defined by a discrete number of static patches, each with its own internal state.

- **Patch Quantity:** The world is initialised with const numGrazingPatches = 80 patches.
- **Overgrazing Threshold:** A patch is exhausted (its state transitions from 'green' to 'yellow') after it has been visited a specific number of times. This is defined by const grazingPatchVisitThreshold = 3. The updateAnimals function increments animal.targetObject.visitCount++ upon a successful grazing interaction.
- **Fallow Period:** Once overgrazed, a patch begins a recovery timer. This period is defined by const grazingPatchRecoveryTime = 120.0. This value is 120 **simulated seconds** (i.e., real-time seconds, subject to simulationSpeedFactor). To understand this in "simulated days," it must be compared to REAL_SECONDS_PER_SIM_DAY.

2.3 Resource Model 2: Trees (Leaves)

The "leaf" resource model is more complex, involving a two-stage recovery process. Leaves are a property of tree objects, which store their leaves in tree.userData.leaves (available) and tree.userData.eatenLeaves (unavailable) arrays.

- **Regrowth Delay:** After a leaf is consumed by an agent (via eatLeaf(animal.targetObject)), the tree is marked with a lastEatenTime. The tree will not begin to regrow **any** leaves until a delay has passed. This is defined by const leafRegrowthDelay = 120.0 (120 simulated seconds).
- **Regrowth Rate:** The leafRegrowthRate is a "time cost" **per leaf**, not a rate of leaves per second. The updateTrees function shows that after the leafRegrowthDelay has passed, a regrowthProgress timer begins. Only when this timer exceeds const leafRegrowthRate = 15.0 (15 simulated seconds) does a single leaf move from the eatenLeaves array back to the leaves array.

This two-stage model makes leaves a highly vulnerable resource. A tree that is eaten from must first wait 120 seconds (the delay) and **then** wait another 15 seconds for the **first** leaf to

regrow, 15 more seconds for the second, and so on.

2.4 Other Environmental Factors

- **Simulated Day Duration:** The fundamental conversion factor between real-time and simulated-time is const REAL_SECONDS_PER_SIM_DAY = 20. This means one simulated day passes for every 20 seconds of real-world time (assuming simulationSpeed is 1x). All "cooldown" and "timer" values in the code, which are expressed in seconds, must be divided by 20 to be understood in the context of "Simulated Days." For example, the grazingPatchRecoveryTime of 120 seconds translates to exactly 6 simulated days ($120 / 20 = 6$).

Section 3: The "Agents": Behavioural Logic & State Machines

Agent behaviour is governed by an internal Finite State Machine (FSM). The core logic is contained within the updateAnimals(deltaTime) function, which uses a switch (animal.state) block to execute behaviour based on the agent's current state.

3.1 Agent State Machine

The following is an exhaustive list of all possible case states defined in the updateAnimals function:

Universal States (All or Most Agents):

- case 'wandering': The default state for all ground-based animals. The agent moves toward a randomly assigned targetPosition. This is the primary state from which all other behavioural decisions (e.g., hunt, drink, mate) are triggered.
- case 'drinking': The agent is immobile at the water source. This state lasts for a stochastic duration of stateTimer = 5.0 + Math.random() * 3 seconds.
- case 'mating': The agent is immobile and collocated with a partner. This state lasts for a fixed duration of stateTimer = 3 seconds, after which spawnOffspring is called by one of the parents.
- case 'dying': The agent is immobile and plays a "death" animation (e.g., obj.rotation.z += 2.0 * deltaTime). The agent is removed from the simulation (animals array) when its stateTimer expires.

Species-Specific States:

- case 'sated': Lion-only. A sub-state of "wandering" that is entered after a successful hunt. The agent wanders at a reduced speed (currentSpeed *= 0.4) and its hunger is reset to 0.
- case 'hunting': Lion-only. The agent moves toward its animal.targetObject (prey) at an increased speed (speed * huntingSpeedMultiplier, where huntingSpeedMultiplier = 1.5).
- case 'grazing': Herbivore-only (Zebra, Antelope, Warthog). The agent is immobile at a

grazingPatch object. This state lasts for a stochastic duration of stateTimer = 2.0 + Math.random() * 2 seconds.
- case 'interacting_tree': Elephant-only. The agent is immobile at a tree object. This state lasts for a fixed duration of const elephantInteractionTime = 3.0 seconds.
- case 'flying': Bird-only. The agent moves through 3D space toward a targetPosition, which is calculated based on a landingSpot (a tree or mountain).
- case 'perched': Bird-only. The agent is immobile on a landingSpot. This state lasts for a stochastic duration of stateTimer = Math.random() * 10 + 10 seconds.

3.2 Agent Behavioural Triggers (Decision-Making)

The decision to transition from the wandering state to another state is based on a "utility" check, where the agent assesses internal needs (hunger, cooldowns) and external opportunities (proximity to resources or mates).

Lion (Predator):

- **Hunting Trigger:** A Lion's decision to hunt is governed by its animal.hunger variable. This variable is initialised at 30 + Math.random() * 40 and increases over time: animal.hunger += deltaTime * 0.3 * predationRateFactor;.
  - A Lion in the wandering state will transition to hunting if its hunger exceeds a **dynamically calculated** threshold: const hungerThreshold = Math.max(40, 80 - 10 * predationRateFactor);. This is a critical non-obvious rule: the user-controlled predationRateFactor (which ranges from 0.6 to 4.0) inversely affects the hungerThreshold.
  - At a **low** predation policy (predationRateFactor = 0.6), the threshold is high: Math.max(40, 80 - 6) = 74. Lions will wait until they are very hungry to hunt.
  - At a **high** predation policy (predationRateFactor = 4.0), the threshold is low: Math.max(40, 80 - 40) = 40. Lions become far more aggressive, initiating hunts at a much lower hunger level.
- **Giving Up a Hunt:** A Lion in the hunting state will "give up" and return to wandering if its animal.stateTimer (which increments **up** during a hunt) exceeds const huntGiveUpTime = 20.0. This is 20 simulated seconds, or exactly **1 simulated day**.

Elephant (Mega-Herbivore):

- **Tree Interaction Trigger:** An Elephant in the wandering state continuously scans for nearby trees. It transitions to interacting_tree if findNearestTreeWithLeaves returns a valid tree within const treeInteractionRadius = 3.0 units.
- **Interaction Logic:** Once in the interacting_tree state (which lasts 3.0 seconds), the agent deterministically calls eatLeaf(animal.targetObject) three times (at 2.0s, 1.0s, and <0.1s remaining in its stateTimer), consuming up to 3 leaves per interaction.

Herbivores (Grazers & Drinkers):

- **Grazing Trigger:** An agent in the grazers array (Zebra, Antelope, Warthog) will transition from wandering to grazing if it comes within proximity of a 'green' grazing patch. The

check is: currentPos.distanceTo(nearestPatch.position) < grazingPatchSize * 0.5 + animal.radius (where const grazingPatchSize = 1.5).

- **Drinking Trigger:** An agent in the eligibleDrinkers array (Zebra, Antelope, Warthog, Elephant) will transition from wandering to drinking if two conditions are met:
  1. **Internal Cooldown:** animal.drinkingCooldown <= 0
  2. **Proximity:** distToWater < waterInfo.radius + animal.radius + drinkingBuffer (where const drinkingBuffer = 0.5).

3.3 Core Agent Parameters (Internal "Genetics")

The following table (Table 3.3.1) is constructed from a line-by-line analysis of the createBlockyAnimal(species) function. It documents the hard-coded baseline parameters ("genetics") for each agent upon instantiation. These values are the foundation upon which user-controlled policy multipliers (Section 4.1) are applied.

Table 3.3.1: Hard-Coded Base Agent Parameters Upon Instantiation

Species	speed (Range)	baseMatingCooldown (Range, Sim. Seconds)	radius (Constant)	Initial State / Other
Zebra	0.3 - 0.5	120.0 - 210.0	0.9	state: 'wandering'
Antelope	0.4 - 0.6	120.0 - 210.0	0.8	state: 'wandering'
Warthog	0.25 - 0.4	120.0 - 210.0	0.5	state: 'wandering'
Elephant	0.15 - 0.25	300.0 - 450.0	1.4	state: 'wandering'
Lion	0.5 - 0.7	95.0 - 160.0	1.1	state: 'wandering', hunger: 30-70
Bird	0.6 - 1.0	Infinity	0.5	state: 'flying'

Note: baseMatingCooldown values are in simulated seconds. To convert to simulated days, divide by REAL_SECONDS_PER_SIM_DAY = 20. For example, a Zebra's cooldown is 6-10.5 simulated days, while an Elephant's is 15-22.5 simulated days.

Section 4: The "External Force": User-Controlled Intervention Variables

This section deconstructs the mechanisms by which the user, as an external force, perturbs the system. These interventions are critical for experimental design and are divided into passive "Policy" controls and active "Intervention" controls.

4.1 "Policy" Controls (The Sliders)

The setupUIEventListeners() function contains the exact mathematical formulas that translate a user's slider input (sliderVal) into an internal model parameter. These formulas are often non-obvious and represent "hidden rules" of the simulation.

Mating Rate (Herbivores, Elephants, Predators):

- **Code:** The event listeners for herbivoreMatingRate, elephantMatingRate, and predatorMatingRate all use the identical formula:
  - herbivoreMatingCooldownFactor = 2.2 - (sliderVal * 0.2);
  - elephantMatingCooldownFactor = 2.2 - (sliderVal * 0.2);
  - predatorMatingCooldownFactor = 2.2 - (sliderVal * 0.2);
- **Formula:** Final_Cooldown_Factor = 2.2 - (Slider_Value * 0.2)
- **Application:** This Final_Cooldown_Factor is used to modify the baseMatingCooldown (from Table 3.3.1) when an animal reproduces (in checkForMating and spawnOffspring). The final cooldown is Final_Mating_Cooldown = baseMatingCooldown * Final_Cooldown_Factor.

This mathematics is inverted from the UI's "x" label. The slider controls the **cooldown duration**, not a direct "rate."

- A **high** slider setting (e.g., 10x) results in sliderVal = 10. The formula is $2.2 - (10 * 0.2) = 0.2$. This 0.2 factor **multiplies** the baseMatingCooldown, causing an 80% **reduction** in cooldown time (i.e., 5x faster breeding).
- A **low** slider setting (e.g., 1x for herbivores) results in sliderVal = 1. The formula is $2.2 - (1 * 0.2) = 2.0$. This **doubles** the cooldown time, making breeding 50% **slower**.
- A **zero** setting (for lions/elephants) results in sliderVal = 0. The formula is $2.2 - (0 * 0.2) = 2.2$. This makes the cooldown 2.2 times **longer**. This is critical: setting mating to "0" does **not** stop reproduction, it only makes it significantly slower.

Predation Rate:

- **Code:** The predationRate slider listener defines the following:
  - const sliderValue = parseFloat(event.target.value);
  - predationRateFactor = sliderValue / 5.0;
- **UI Sliders:** The HTML for this slider is <input type="range" id="predationRate"... min="3" max="20" value="10"...>.
- **Formula:** predationRateFactor = Slider_Value / 5.0

- **Parameter Range:** This hard-coded formula and slider range (3 to 20) means the predationRateFactor has an effective, non-modifiable range of **0.6x** (slider at 3) to **4.0x** (slider at 20). The default value is **2.0x** (slider at 10).

Crucial Logic: The Lion Starvation Event
The predationRate slider's event listener contains a hard-coded "trap" that deterministically kills lions if the policy is set too low.

- **Trigger 1 (Enabling):** if (predationRateFactor < 1.0) {... isLionStarvationEventActive = true; lastLionStarvationDay = simulatedDays; }. This is triggered if the slider value is set below 5.
- **Trigger 2 (Disabling):** else {... isLionStarvationEventActive = false; }.
- Consequence: The main animate() loop reads this boolean flag. If isLionStarvationEventActive is true, the following logic is executed: if (isLionStarvationEventActive && populationCounts['Lion'] > 0 && simulatedDays >= lastLionStarvationDay + 14) {... setAnimalDying(lionToStarve, 2.0, 'predator_starvation');... lastLionStarvationDay = simulatedDays; }.
  This is a critical "hidden rule." Setting the predationRate slider below 5 (i.e., a factor < 1.0x) does not just make lions "inefficient." It activates a separate, deterministic rule that automatically kills one lion from the animals array every 14 simulated days, regardless of prey availability. This models a policy so "inefficient" that it guarantees the predator population's collapse from starvation.

4.2 "Intervention" Controls (The Hunter)

This subsystem models the ethics and consequences of "culling" vs. "poaching." It is governed by two separate counters: huntsThisPeriod (a short-term, cyclical quota) and successfulHunts (a long-term, cumulative "poaching" tracker).

Sustainable Culling Quota:

- **Quota:** The allowed sustainable quota is **2 hunts**.
- **Code:** This is checked in two places, preventing both the initiation of a hunt (huntButtonElement listener) and the firing of a weapon (fireWeapon function) if the quota is met: if (huntsThisPeriod >= 2) { showWarningMessage("Hunting allowance for this 30-day period is exhausted.", true); return; }.

Management Cycle:

- **Cycle:** The huntsThisPeriod quota resets to 0 every **30 simulated days**.
- **Code:** This logic is found in the animate() function: if (simulatedDays > 0 && simulatedDays % 30 === 0 && huntsThisPeriod !== "reset") {... huntsThisPeriod = 0;... }.

Ethical/Legal Consequences (Poaching Model):
The cumulative successfulHunts counter, which never resets, is checked by the checkHuntingConsequences() function after every successful hunt. This function details a series of progressive failure states based on this single variable.

- **6 Hunts:** if (successfulHunts >= 6 && !warningWelfareShown): Triggers the warning

"Animal welfare organisations are concerned by your high level of hunting activity."
- **10 Hunts:** if (successfulHunts >= 10 && !warningAuthoritiesShown): Triggers the warning "Conservation authorities are now actively monitoring your hunting activities."
- **12 Hunts:** if (successfulHunts >= 12): Triggers the hard-fail state showFailureMessage("ARRESTED").

This two-tiered system is a direct model of a Social-Ecological System (SES). huntsThisPeriod represents a **soft ecological rule** for a "manager" to follow. successfulHunts represents a **hard social rule** (a law or social license). Exceeding 12 cumulative hunts triggers a non-ecological, **social** system failure ("ARRESTED"), demonstrating a hard boundary placed on an ecological action by social rules.

Section 5: Systemic Behaviour & Emergent Properties

5.1 Systemic Failure States

A "systemic failure state" is any condition that terminates the simulation. These are triggered by calls to the showFailureMessage(reasonCode) function. The reasonCode defines the cause of the ecosystem's collapse. The following table (Table 5.1.1) lists every reasonCode from the showFailureMessage function and traces the **exact** line of code or condition that triggers it.

Table 5.1.1: System Failure Triggers

reasonCode	Failure Message (from showFailureMessage)	Triggering Condition & Function
ARRESTED	"You have been arrested!"	successfulHunts >= 12 (in checkHuntingConsequences)
LION_EXTINCTION_STARVED	"The lions starved due to lack of food."	populationCounts['Lion'] === 0 AND animal.causeOfDeath !== 'hunt' (in updateAnimals loop)
LION_EXTINCTION_HUNTED	"All lions were hunted to extinction."	populationCounts['Lion'] === 0 AND animal.causeOfDeath === 'hunt' (in updateAnimals loop)
ELEPHANT_EXTINCTION_H	"All elephants were hunted	populationCounts['Elephan

UNTED	to extinction."	t'] === 0 AND animal.causeOfDeath === 'hunt' (in updateAnimals loop)
NO_LEAVES	"All the leaves have been eaten."	totalLeafCount <= 0 (in eatLeaf function)
HERBIVORES_EXTINCT	"All primary prey species (Zebra, Antelope, Warthog) are extinct."	antelopeExtinct && zebraExtinct && warthogExtinct (in updateAnimals loop)
CARRYING_CAPACITY	"Herbivore population exceeded the savanna's carrying capacity."	currentTotalHerbivoresForFailure > maxTotalHerbivores (where maxTotalHerbivores = 120) (in updateAnimals loop)
PLAYER_EATEN	"You were eaten by a hungry lion!"	Lion's case 'hunting' proximity check successful on hunterObject (in updateAnimals loop)

5.2 Connection to Ecological Theory

The model's hard-coded rules (Sections 2, 3, 4) directly connect to established ecological and systems-theory concepts.

- **Functional Response:** The predationRateFactor (Section 4.1) is a direct model of a predator's "Functional Response"—the intake rate of a predator in relation to prey density (though here, it is abstracted as a user-set policy on **efficiency**). The "Lion Starvation Event" demonstrates a key principle: if a predator's hunting efficiency (the predationRateFactor policy) falls below a subsistence threshold (defined in the code as < 1.0), its population will crash deterministically, **even if prey is abundant**. This models a policy-driven, rather than a density-driven, population collapse.
- **Social-Ecological System (SES):** The Hunter's "ARRESTED" state (Section 4.2, 5.1) is a direct and explicit model of a Social-Ecological System (SES). In a purely ecological model, a poacher (hunter) could hunt indefinitely until the system collapses ecologically. In this SES model, social rules (laws/ethics, represented by the successfulHunts < 12 boundary) create a hard boundary on an ecological action (hunting). The simulation does not allow the ecological system to be destroyed by poaching **past a certain point**; instead, the **social** system intervenes and terminates the simulation with a social,

non-ecological failure code ("ARRESTED").

- **Trophic Cascade & Keystone Species:** The "Kruger Elephant Dilemma" experiment (Section 6.3) is designed to test a classic trophic cascade. Elephants, as keystone species, are primary modifiers of the environment. The model's logic (eatLeaf) allows them to (if overpopulated) eliminate the primary producers (trees). This leads to a "bottom-up" collapse, as evidenced by the "NO_LEAVES" failure state.
- **Carrying Capacity (K):** The model explicitly defines a hard carrying capacity for herbivores: const maxTotalHerbivores = 120. This is a classic concept from logistic growth models (where K = carrying capacity). In this simulation, it is not implemented as a gradual logistic curve (where birth rates decline as populations approach K) but as a hard failure state. If the total number of herbivores exceeds 120, the system **immediately** collapses with the "CARRYING_CAPACITY" failure code. This represents a "tipping point" rather than a "limit".

5.3 Key Tipping Points

Based on the analysis of the failure states in Section 5.1, the model's dynamics are governed by several hard-coded, deterministic "tipping points." Crossing these thresholds will non-linearly and irreversibly shift the system into a failure state.

- **Social Tipping Point:** successfulHunts >= 12.
- **Resource Tipping Point:** totalLeafCount <= 0.
- **Policy Tipping Point:** predationRateFactor < 1.0 (activates the deterministic Lion Starvation timer).
- **Population Tipping Points:**
  - currentTotalHerbivoresForFailure > 120.
  - populationCounts['Zebra'] == 0 AND populationCounts['Antelope'] == 0 AND populationCounts['Warthog'] == 0.

Section 6: Guide to Conceptual Experimental Design

6.1 How to Use This Manual

This manual provides a "glass box" model of the simulator's internal logic, enabling the design of rigorous, repeatable conceptual experiments. A researcher can now move beyond "playing the game" and engage in formal hypothesis testing.

A valid experiment should be designed as follows:

1. **Define Independent Variables:** Select one or more "Policy" controls from Section 4.1 (e.g., elephantMatingRate) or "Intervention" controls from Section 4.2 (e.g., adhering to the "Manager" quota vs. "Poacher" behaviour).
2. **Set Initial State & Constants:** Use the parameters in Section 3.3 (e.g., startElephant in the setup screen) to define the initial state. All other user-controlled variables (Section 4) not being tested must be held at their default values (e.g., Herbivore Mating 5, Predation 10) to serve as experimental constants.

3. **Define Dependent Variables:** The primary outcomes to be measured are the simulation's **Simulated Days Survived**, the **Reason for Collapse** (from the list in Section 5.1), and the **Time-Series Data** collected via the Data Logger.

6.2 Quantitative Data Extraction

To facilitate rigorous analysis, the model now includes a "Black Box Recorder" feature. Upon any systemic failure (Game Over), a purple **"Export Experiment Data"** button appears.

The Dataset:
Downloading this file provides a time-series dataset where each row represents one Simulated Day. This allows researchers to graph population dynamics, identify lag effects, and pinpoint the exact moment of tipping points.
Recorded Variables:

- **Time:** Day (Independent Variable).
- **System State (Dependent Variables):** Pop_Elephant, Pop_Lion, Pop_Zebra, Pop_Antelope, Pop_Warthog, Pop_Trees, Leaves_Remaining.
- **Policy Inputs (Independent Variables):** Input_PredationRate, Input_MatingRate_Herbivore, Input_MatingRate_Elephant.
- **Flow Events:** Total_Hunts, Total_Predations, Count_Matings_Elephant (Cumulative).

Note on Final State Capture: The data logger is triggered one final time inside the showFailureMessage function immediately before the simulation halts. This ensures the final "crash" state (e.g., Pop_Zebra: 0) is recorded in the CSV, preventing "off-by-one" data truncation.

6.3 Sample Experiment 1: The "Kruger Elephant Dilemma"

This experiment models the real-world conservation conflict between "preservation" (no intervention) and "conservation" (active management/culling) in elephant-dense reserves.

Hypothesis: "A hands-off 'Preservation' policy (no culling) in an elephant-dense environment is more destructive to systemic biodiversity (measured by system survival time and leaf-resource stability) than a hands-on 'Conservation' policy (sustainable culling)."

Methodology:

1. **Set Initial State:** On the Setup Screen, set startElephant to its maximum value (8). Set all other populations (startLion, startZebra, etc.) to their default values (e.g., 2, 6).
2. **Set Policy Constants:** In the Dashboard, ensure all sliders (Section 4.1) are set to their default values (Herbivore Mating 5, Elephant Mating 2, Predator Mating 2, Predation 10).
3. **Run Scenario A (Preservation):** Start the simulation. **Do not** press the "Hunt" button. Allow the simulation to run until a failure state is reached.
4. **Run Scenario B (Conservation):** Restart the simulation with identical parameters from steps 1 and 2. Press the "Hunt" button. Actively hunt **only** elephants. Adhere **strictly** to the sustainable culling quota (huntsThisPeriod <= 2) per 30-day management cycle (as

defined in 4.2).

What to Measure: Export the CSV data for both scenarios. Plot Pop_Trees vs. Time and Leaves_Remaining vs. Time. Compare the rate of resource depletion (slope of the curve) between the Preservation and Conservation scenarios.

Expected Observation: Scenario A is hypothesised to fail relatively quickly with the Reason for Collapse code "NO_LEAVES", demonstrating a trophic cascade. Scenario B is hypothesised to survive significantly longer, as the culling intervention prevents the "bottom-up" resource collapse, allowing the system to fail (or not fail) for other reasons.

6.4 Sample Experiment 2: The "Poacher vs. Manager" Ethical Model

This experiment uses the model's built-in SES logic to test the difference between a "manager" (who follows sustainable rules) and a "poacher" (who breaks social rules).

Hypothesis: "The model demonstrates that 'management' is defined by adherence to systemic rules (quotas), while 'poaching' is an action that destabilises the **social-ecological system** itself, leading to a unique, non-ecological type of system failure."

Methodology:

1. **Set Initial State:** Set all initial populations to their default values.
2. **Run Scenario A (Manager):** Follow the methodology from Experiment 6.3, Scenario B. Hunt **any** animal, but **never** exceed the quota of huntsThisPeriod = 2 per 30-day cycle. The cumulative successfulHunts counter will rise slowly, but the "manager" rule is respected.
3. **Run Scenario B (Poacher):** Restart with identical parameters. Press the "Hunt" button. Hunt **any** animal as quickly as possible. Deliberately **ignore** the "allowance exhausted" warnings (huntsThisPeriod >= 2). The goal is to accumulate successfulHunts as fast as possible.

What to Measure: Export the CSV data. Compare the Total_Hunts variable over time. Observe that in Scenario A (Manager), the variable increases linearly but stays within quotas. In Scenario B (Poacher), observe the rapid acceleration of Total_Hunts until the social tipping point (12) triggers the 'ARRESTED' termination code.

Expected Observation: Scenario A will fail from an **ecological** cause, dictated by the user's management choices (e.g., "HERBIVORES_EXTINCT" if too much prey is culled, or "NO_LEAVES" if elephants are ignored). Scenario B, however, is guaranteed to fail **rapidly** with the **social** failure code "ARRESTED" as soon as successfulHunts reaches 12. This demonstrates that the model's "poaching" pathway is a test of a social boundary, not an ecological one.