Digital Twins for Salmon Wellbeing: A Tutorial Series

Based on Giske et al. (2025)

Arvid Lundervold w/ Claude 3.7 Sonnet - February 25, 2025

Abstract

This tutorial provides a step-by-step implementation guide for creating digital twins that model salmon wellbeing based on the paper "Premises for digital twins reporting on Atlantic salmon wellbeing" (Giske et al., 2025). The implementation captures predictions on boredom, stress and wellbeing using a computational evolutionary model of the factors underlying behavior. We demonstrate how to construct an agent-based model of salmon digital twins by modeling subjective wellbeing experience, prediction of near future, and allostasis (the bodily preparation for expected near future). Each section progressively builds the components required for digital twin models that can deliver early warnings about issues affecting salmon health in aquaculture settings. This implementation supports the 3Rs (replacement, reduction, refinement) by providing actionable information without relying on animal experiments.

Contents

1 Introduction

1.1 Background

Digital twins are virtual representations of physical entities that can be used to model and predict the behavior of the corresponding real-world objects. In the context of salmon aquaculture, digital twins can provide insights into fish wellbeing without invasive procedures or extensive experimentation. The paper by Giske et al. (2025) [31] outlines the theoretical framework for such digital twins, focusing on the mechanisms underlying wellbeing prediction in salmon.

1.2 Key Concepts

Before diving into implementation, it's important to understand several key concepts:

- **Agency**: The ability of an autonomous entity to set its own goal-directed behavior
- **Allostasis**: The preparative regulation of bodily resources before a need arises
- **Survival circuits**: Integrated neural pathways responding to specific subjective internal models
- **Global organismic state**: The organism's centralized emotional state
- **Episodic-like memory**: The ability to remember what/where/when information from experiences
- **Subjective internal models**: Internal representations of aspects of self or environment

1.3 Tutorial Overview

This tutorial is structured as follows:

1. Basic building blocks for the digital twin
2. Implementation of survival circuits
3. Learning and memory mechanisms
4. Wellbeing assessment systems
5. Evolutionary adaptation framework
6. Integration into a complete digital twin
7. Validation and analysis methods

Each section provides theoretical background, implementation code, and examples of usage.

2 Prerequisites and Setup

2.1 Software Requirements

The implementation requires:

- Python 3.8+
- NumPy, Pandas, SciPy
- Matplotlib, Seaborn for visualization
- NetworkX for survival circuit modeling
- Gym for reinforcement learning environments

2.2 Installation

```
# Create virtual environment
python -m venv salmon_twin_env
source salmon_twin_env/bin/activate  # On Windows: salmon_twin_env\Scripts\
    activate

# Install requirements
pip install numpy pandas scipy matplotlib seaborn networkx gym
```

3 Basic Building Blocks

3.1 Core Data Structures

The first step is to implement the core data structures for the digital twin.

3.1.1 Basic Needs Representation

Based on Figure 1 from the paper, we implement the basic needs of Atlantic salmon:

```
class BasicNeeds:
    """Representation of basic needs categories for salmon wellbeing"""

    def __init__(self):
        # Cognitive needs
        self.exploration = 0.0
        self.protection = 0.0
        self.safety = 0.0

        # Social needs
        self.social_contact = 0.0
        self.sexual_behavior = 0.0

        # Bodily needs
        self.feeding = 0.0
        self.nutrition = 0.0
        self.health = 0.0
        self.rest = 0.0
        self.kinesis = 0.0
        self.body_care = 0.0

        # Physical needs
        self.thermal_regulation = 0.0
        self.osmotic_balance = 0.0
        self.respiration = 0.0

        # Behavior control need
        self.behavior_control = 0.0

    def get_all_needs(self):
        """Return all needs as a dictionary"""
        return {
            "exploration": self.exploration,
            "protection": self.protection,
            "safety": self.safety,
            "social_contact": self.social_contact,
            "sexual_behavior": self.sexual_behavior,
            "feeding": self.feeding,
            "nutrition": self.nutrition,
            "health": self.health,
            "rest": self.rest,
```

```python
            "kinesis": self.kinesis,
            "body_care": self.body_care,
            "thermal_regulation": self.thermal_regulation,
            "osmotic_balance": self.osmotic_balance,
            "respiration": self.respiration,
            "behavior_control": self.behavior_control
        }

    def get_most_urgent_need(self):
        """Return the most urgent need (highest value)"""
        needs = self.get_all_needs()
        return max(needs.items(), key=lambda x: x[1])
```

3.1.2 Neuronal Response Function

The neuronal response function converts metric values (such as temperature or oxygen levels) into subjective values in the salmon's brain:

```python
import numpy as np

class NeuronalResponse:
    """
    Converts metric input values to subjective values in the salmon brain
    using a non-linear function (sigmoid by default)
    """

    def __init__(self, threshold, sensitivity):
        """
        Args:
            threshold: The inflection point of the sigmoid function
            sensitivity: The steepness of the sigmoid curve
        """
        self.threshold = threshold
        self.sensitivity = sensitivity

    def activate(self, input_value):
        """Convert metric input to subjective value"""
        return 1 / (1 + np.exp(-self.sensitivity * (input_value - self.
    threshold)))

    def __call__(self, input_value):
        """Allow direct calling of the object as a function"""
        return self.activate(input_value)
```

3.1.3 Sensor Implementation

Sensors process environmental inputs:

```python
class Sensor:
    """
    Base class for sensors that detect environmental conditions
    """

    def __init__(self, name, neuronal_response):
        self.name = name
        self.neuronal_response = neuronal_response
        self.last_value = None
        self.last_processed = None

    def sense(self, environment_value):
        """
        Sense a value from the environment and process it
```

```
        through the neuronal response function
        """
        self.last_value = environment_value
        self.last_processed = self.neuronal_response(environment_value)
        return self.last_processed
```

3.2 Environment Representation

The environment provides inputs to the digital twin:

```
class Environment:
    """
    Representation of the aquaculture environment
    """

    def __init__(self,
                 temperature=10,
                 oxygen_level=8.5,
                 light_intensity=100,
                 food_availability=1.0,
                 social_density=50,
                 noise_level=0.1):
        self.temperature = temperature
        self.oxygen_level = oxygen_level
        self.light_intensity = light_intensity
        self.food_availability = food_availability
        self.social_density = social_density
        self.noise_level = noise_level
        self.time = 0

    def get_state(self):
        """Return the current state of the environment"""
        return {
            "temperature": self.temperature,
            "oxygen_level": self.oxygen_level,
            "light_intensity": self.light_intensity,
            "food_availability": self.food_availability,
            "social_density": self.social_density,
            "noise_level": self.noise_level,
            "time": self.time
        }

    def step(self, delta_t=1):
        """Advance the environment by time delta_t"""
        self.time += delta_t

        # Simulate some environmental fluctuations
        self.temperature += np.random.normal(0, 0.1)
        self.oxygen_level += np.random.normal(0, 0.05)
        self.light_intensity = max(0, self.light_intensity + np.random.normal
    (0, 5))

        # Ensure values stay in reasonable ranges
        self.temperature = np.clip(self.temperature, 5, 20)
        self.oxygen_level = np.clip(self.oxygen_level, 4, 12)
```

3.3 Visualization Tools

Implementing basic visualization for the needs:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

```python

def visualize_needs(basic_needs):
    """
    Create a bar chart of basic needs

    Args:
        basic_needs: BasicNeeds object
    """
    needs = basic_needs.get_all_needs()

    # Create grouped bars by category
    cognitive = ["exploration", "protection", "safety"]
    social = ["social_contact", "sexual_behavior"]
    bodily = ["feeding", "nutrition", "health", "rest", "kinesis", "body_care"]
    physical = ["thermal_regulation", "osmotic_balance", "respiration"]
    control = ["behavior_control"]

    categories = {
        "Cognitive": [needs[n] for n in cognitive],
        "Social": [needs[n] for n in social],
        "Bodily": [needs[n] for n in bodily],
        "Physical": [needs[n] for n in physical],
        "Control": [needs[n] for n in control]
    }

    fig, ax = plt.subplots(figsize=(12, 6))

    x = np.arange(len(categories))
    width = 0.8 / max(len(v) for v in categories.values())

    for i, (category, values) in enumerate(categories.items()):
        bars = []
        for j, val in enumerate(values):
            bars.append(ax.bar(i + j*width, val, width, label=f"{category} {j
+1}"))

    ax.set_ylabel('Need Intensity')
    ax.set_title('Basic Needs of Digital Salmon')
    ax.set_xticks(x + width/2)
    ax.set_xticklabels(categories.keys())

    plt.tight_layout()
    plt.show()
```

4 Survival Circuits

Based on Figure 2 from the paper, we implement the survival circuits that drive salmon behavior.

4.1 Survival Circuit Implementation

```python
class SurvivalCircuit:
    """
    Implementation of a survival circuit as described in the paper.
    A survival circuit is a highly integrated neural pathway from memory
    or new sensing via attention to behavior.
    """

    def __init__(self, name, sensors=None):
        self.name = name
        self.sensors = sensors or []  # List of Sensor objects
        self.neurobiological_state = 0.0  # Current activation level
```

```python
        self.hormone_modulation = 1.0  # Default hormone influence

    def add_sensor(self, sensor):
        """Add a sensor to this circuit"""
        self.sensors.append(sensor)

    def process_inputs(self, environment):
        """
        Process environmental inputs through the circuit's sensors

        Args:
            environment: Environment object with current state

        Returns:
            The neurobiological state activation level (0-1)
        """
        if not self.sensors:
            return 0.0

        env_state = environment.get_state()

        # Process each sensor input
        activations = []
        for sensor in self.sensors:
            if sensor.name in env_state:
                activations.append(sensor.sense(env_state[sensor.name]))

        # If we have activations, compute the neurobiological state
        if activations:
            # Apply sigmoid function to combine inputs
            activation = np.mean(activations)  # Simple averaging for now

            # Apply hormone modulation
            activation *= self.hormone_modulation

            # Update the neurobiological state
            self.neurobiological_state = activation

        return self.neurobiological_state

    def set_hormone_modulation(self, modulation_value):
        """
        Set hormone modulation to adjust the circuit's sensitivity

        Args:
            modulation_value: Value between 0-2 where 1 is neutral
        """
        self.hormone_modulation = max(0, modulation_value)
```

4.2 Global Organismic State

The global organismic state (GOS) represents the dominant emotional state:

```python
class GlobalOrganismicState:
    """
    The organism's centralized emotional state as defined by
    the currently dominant survival circuit
    """

    def __init__(self):
        self.active = False
        self.dominant_circuit = None
        self.attention_focus = None
```

```python
        self.intensity = 0.0
        self.predicted_emotions = {}  # Emotional predictions for options

    def update(self, survival_circuits, attention_threshold=0.3):
        """
        Update the GOS based on competition between survival circuits

        Args:
            survival_circuits: List of SurvivalCircuit objects
            attention_threshold: Minimum activation needed for GOS

        Returns:
            True if GOS is active, False otherwise
        """
        # Find the most active circuit
        if not survival_circuits:
            self.active = False
            self.dominant_circuit = None
            self.intensity = 0.0
            return False

        # Get the circuit with highest activation
        most_active = max(
            survival_circuits,
            key=lambda circ: circ.neurobiological_state
        )

        # Only establish GOS if the activation exceeds threshold
        if most_active.neurobiological_state >= attention_threshold:
            self.active = True
            self.dominant_circuit = most_active
            self.attention_focus = most_active.name
            self.intensity = most_active.neurobiological_state
            return True
        else:
            self.active = False
            self.dominant_circuit = None
            self.intensity = 0.0
            return False
```

4.3 Integrated Example

Let's put these components together:

```python
def create_basic_survival_circuits():
    """Create a set of basic survival circuits for salmon"""

    # Create neuronal responses
    temp_response = NeuronalResponse(threshold=12, sensitivity=0.5)
    oxygen_response = NeuronalResponse(threshold=6, sensitivity=2.0)
    food_response = NeuronalResponse(threshold=0.3, sensitivity=5.0)
    light_response = NeuronalResponse(threshold=50, sensitivity=0.05)
    noise_response = NeuronalResponse(threshold=0.5, sensitivity=-5.0)

    # Create sensors
    temp_sensor = Sensor("temperature", temp_response)
    oxygen_sensor = Sensor("oxygen_level", oxygen_response)
    food_sensor = Sensor("food_availability", food_response)
    light_sensor = Sensor("light_intensity", light_response)
    noise_sensor = Sensor("noise_level", noise_response)

    # Create survival circuits
    growth_circuit = SurvivalCircuit("growth")
```

```python
    growth_circuit.add_sensor(temp_sensor)
    growth_circuit.add_sensor(food_sensor)

    defence_circuit = SurvivalCircuit("defence")
    defence_circuit.add_sensor(noise_sensor)

    reproduction_circuit = SurvivalCircuit("reproduction")

    respiration_circuit = SurvivalCircuit("respiration")
    respiration_circuit.add_sensor(oxygen_sensor)

    exploration_circuit = SurvivalCircuit("exploration")
    exploration_circuit.add_sensor(light_sensor)

    return [
        growth_circuit,
        defence_circuit,
        reproduction_circuit,
        respiration_circuit,
        exploration_circuit
    ]
```

5 Learning and Memory

5.1 Episodic-like Memory

Implementing episodic-like memory for salmon to remember experiences:

```python
class EpisodicMemory:
    """
    Implementation of episodic-like memory that stores what/where/when/emotion
    information from experiences
    """

    def __init__(self, capacity=100):
        self.capacity = capacity
        self.episodes = []

    def store(self, what, where, when, emotion):
        """
        Store a new memory episode

        Args:
            what: What happened (e.g., "feeding")
            where: Location information
            when: Timestamp
            emotion: Emotional valence of the experience
        """
        # Create new episode
        episode = {
            "what": what,
            "where": where,
            "when": when,
            "emotion": emotion,
            "retrieval_count": 0  # Track how often this is retrieved
        }

        # Add to episodes, maintaining capacity
        self.episodes.append(episode)
        if len(self.episodes) > self.capacity:
            # Remove least accessed episode if we're over capacity
            self.episodes.sort(key=lambda e: e["retrieval_count"])
```

```
            self.episodes.pop(0)

    def retrieve_by_similarity(self, what=None, where=None, when=None):
        """
        Retrieve episodes that match the given criteria

        Returns:
            List of matching episodes
        """
        matches = []

        for episode in self.episodes:
            score = 0

            if what and episode["what"] == what:
                score += 1
            if where and episode["where"] == where:
                score += 1
            if when and abs(episode["when"] - when) < 24:  # Within 24 time
units
                score += 1

            if score > 0:
                matches.append({
                    "episode": episode,
                    "score": score
                })
                episode["retrieval_count"] += 1

        # Sort matches by similarity score
        matches.sort(key=lambda m: m["score"], reverse=True)
        return [m["episode"] for m in matches]

    def retrieve_emotional_prediction(self, what, where=None):
        """
        Retrieve emotional prediction for a given situation

        Args:
            what: The situation to predict emotion for
            where: Optional location context

        Returns:
            Predicted emotion value or None if no matching experiences
        """
        relevant = self.retrieve_by_similarity(what=what, where=where)

        if not relevant:
            return None

        # Calculate the average emotional value, weighted by recency
        total_emotion = 0
        total_weight = 0

        for i, episode in enumerate(relevant):
            # More recent episodes get higher weight
            weight = 1.0 / (i + 1)
            total_emotion += episode["emotion"] * weight
            total_weight += weight

        if total_weight > 0:
            return total_emotion / total_weight
        else:
            return None
```

5.2 Learning Implementation

Implementing the learning mechanism:

```python
class Learning:
    """
    Implementation of learning mechanisms for the digital twin
    """

    def __init__(self, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.associations = {}  # Learned associations

    def update_association(self, stimulus, response, reward):
        """
        Update association between stimulus and response based on reward

        Args:
            stimulus: The stimulus (input)
            response: The response (action)
            reward: The reward value (-1 to 1)
        """
        key = (stimulus, response)

        if key in self.associations:
            # Update existing association using learning rate
            current = self.associations[key]
            self.associations[key] = current + self.learning_rate * (reward -
    current)
        else:
            # Create new association
            self.associations[key] = self.learning_rate * reward

    def predict_reward(self, stimulus, response):
        """
        Predict reward for a stimulus-response pair

        Returns:
            Predicted reward or 0 if no association exists
        """
        key = (stimulus, response)
        return self.associations.get(key, 0.0)

    def get_best_response(self, stimulus, possible_responses):
        """
        Get the response with highest predicted reward

        Args:
            stimulus: The stimulus to respond to
            possible_responses: List of possible responses

        Returns:
            The response with highest predicted reward
        """
        if not possible_responses:
            return None

        best_response = possible_responses[0]
        best_reward = self.predict_reward(stimulus, best_response)

        for response in possible_responses[1:]:
            reward = self.predict_reward(stimulus, response)
            if reward > best_reward:
                best_reward = reward
```

```
                best_response = response

        return best_response
```

5.3 Prediction Error

Implementing prediction error calculation:

```
class PredictionError:
    """
    Calculates prediction errors between expected and observed states
    """

    def __init__(self):
        self.predictions = {}
        self.error_history = []

    def set_prediction(self, variable, expected_value):
        """Set a prediction for a variable"""
        self.predictions[variable] = expected_value

    def calculate_error(self, variable, observed_value):
        """
        Calculate prediction error for a variable

        Returns:
            Error value or None if no prediction exists
        """
        if variable not in self.predictions:
            return None

        expected = self.predictions[variable]
        error = observed_value - expected

        # Store in history
        self.error_history.append({
            "variable": variable,
            "expected": expected,
            "observed": observed_value,
            "error": error
        })

        # Limit history size
        if len(self.error_history) > 1000:
            self.error_history.pop(0)

        return error

    def get_recent_errors(self, n=10):
        """Get the n most recent errors"""
        return self.error_history[-n:]

    def get_average_error(self, n=10):
        """Get the average absolute error over the last n observations"""
        recent = self.get_recent_errors(n)
        if not recent:
            return 0.0

        return sum(abs(e["error"]) for e in recent) / len(recent)
```

6 Wellbeing Assessment

Based on Figure 3 from the paper, we implement the wellbeing assessment components.

6.1 Wellbeing Metrics

```python
class WellbeingAssessment:
    """
    Assesses wellbeing based on various metrics
    """

    def __init__(self):
        self.stress_level = 0.0
        self.boredom_level = 0.0
        self.wellbeing_score = 0.5  # Start at neutral
        self.history = []

    def assess_stress(self, gos, prediction_error):
        """
        Assess stress level based on GOS and prediction errors

        Args:
            gos: GlobalOrganismicState object
            prediction_error: PredictionError object

        Returns:
            Stress level between 0-1
        """
        # Factors that contribute to stress:
        # 1. High GOS intensity indicates acute stress
        # 2. High prediction errors indicate uncertainty
        # 3. Duration of GOS activation

        gos_intensity = gos.intensity if gos.active else 0.0
        avg_error = prediction_error.get_average_error(10)

        # Combine factors (weighted sum)
        stress = 0.5 * gos_intensity + 0.5 * min(1.0, avg_error)

        # Update stress level (with smoothing)
        self.stress_level = 0.8 * self.stress_level + 0.2 * stress

        return self.stress_level

    def assess_boredom(self, gos, prediction_error, time_without_gos):
        """
        Assess boredom level

        Args:
            gos: GlobalOrganismicState object
            prediction_error: PredictionError object
            time_without_gos: Time units without GOS activation

        Returns:
            Boredom level between 0-1
        """
        # Factors that contribute to boredom:
        # 1. Long time without GOS activation
        # 2. Low prediction errors (environment too predictable)
        # 3. Low sensory variation

        # Calculate boredom factors
```

```
        gos_inactivity = min(1.0, time_without_gos / 100.0)  # Saturate at 100 time units
        error_factor = max(0.0, 1.0 - prediction_error.get_average_error(20))

        # Combine factors
        boredom = 0.7 * gos_inactivity + 0.3 * error_factor

        # Update boredom level (with smoothing)
        self.boredom_level = 0.9 * self.boredom_level + 0.1 * boredom

        return self.boredom_level

    def assess_wellbeing(self, stress, boredom, predicted_emotions):
        """
        Calculate overall wellbeing

        Args:
            stress: Current stress level (0-1)
            boredom: Current boredom level (0-1)
            predicted_emotions: Dict of predicted emotional outcomes

        Returns:
            Wellbeing score between 0-1
        """
        # Wellbeing is reduced by stress and boredom
        wellbeing = 1.0 - 0.5 * stress - 0.3 * boredom

        # Factor in predicted emotions if available
        if predicted_emotions:
            avg_prediction = sum(predicted_emotions.values()) / len(predicted_emotions)
            wellbeing = 0.7 * wellbeing + 0.3 * avg_prediction

        # Ensure value is in range [0, 1]
        wellbeing = max(0.0, min(1.0, wellbeing))

        # Update wellbeing score (with smoothing)
        self.wellbeing_score = 0.8 * self.wellbeing_score + 0.2 * wellbeing

        # Record history
        self.history.append({
            "stress": stress,
            "boredom": boredom,
            "wellbeing": self.wellbeing_score
        })

        return self.wellbeing_score

    def get_wellbeing_report(self):
        """
        Generate a detailed wellbeing report

        Returns:
            Dictionary with wellbeing metrics
        """
        return {
            "wellbeing_score": self.wellbeing_score,
            "stress_level": self.stress_level,
            "boredom_level": self.boredom_level,
            "history": self.history[-10:] if len(self.history) > 10 else self.history
        }
```

6.2 Decision Making Process

Implementing the decision-making process based on wellbeing:

6.3 Decision Making Process

Implementing the decision-making process based on wellbeing:

```python
class DecisionMaking:
    """
    Implements the decision-making process based on wellbeing predictions
    """

    def __init__(self, episodic_memory, learning):
        self.episodic_memory = episodic_memory
        self.learning = learning
        self.time_without_gos = 0
        self.last_decision = None
        self.last_reward = None

    def decide(self, gos, environment, available_actions):
        """
        Make a decision based on wellbeing predictions

        Args:
            gos: GlobalOrganismicState object
            environment: Environment object
            available_actions: List of possible actions

        Returns:
            The selected action
        """
        # Track time without GOS
        if not gos.active:
            self.time_without_gos += 1
        else:
            self.time_without_gos = 0

        # Decision process differs based on GOS activation
        if gos.active:
            # With active GOS, decision is focused on the dominant need
            action_type = gos.attention_focus

            # Filter actions relevant to the focus
            relevant_actions = [a for a in available_actions
                                if a.startswith(action_type)]

            if not relevant_actions:
                # If no relevant actions, pick best available
                selected_action = self._select_best_action(
                    available_actions, environment)
            else:
                # Pick best relevant action
                selected_action = self._select_best_action(
                    relevant_actions, environment)
        else:
            # Without GOS, use broader wellbeing prediction
            selected_action = self._select_best_action(
                available_actions, environment)

        # Store the decision for later learning
        self.last_decision = selected_action

```

```
        return selected_action

    def _select_best_action(self, actions, environment):
        """
        Select the action with the best predicted wellbeing outcome
        """
        # No actions available
        if not actions:
            return None

        # Get current environment state as stimulus
        current_state = str(environment.get_state())

        # Use learning model to select best response
        return self.learning.get_best_response(current_state, actions)

    def update_from_reward(self, reward):
        """
        Update learning from received reward

        Args:
            reward: Reward value (-1 to 1)
        """
        if self.last_decision is not None:
            # Update learning
            current_state = str(environment.get_state())
            self.learning.update_association(
                current_state, self.last_decision, reward)

            # Update episodic memory
            self.episodic_memory.store(
                what=self.last_decision,
                where=str(environment.get_state()),
                when=environment.time,
                emotion=reward
            )

            self.last_reward = reward
```

7 Evolution and Adaptation

The genetic algorithm enables the digital twins to adapt over generations, reflecting the evolutionary process.

7.1 Genetic Algorithm Components

First, we define the gene structure for our digital twins:

```
class SalmonGenes:
    """
    Represents the genetic makeup of a salmon digital twin
    """

    def __init__(self):
        # Genes for neuronal response thresholds
        self.thresholds = {
            "temperature": np.random.uniform(5, 15),
            "oxygen_level": np.random.uniform(4, 10),
            "food_availability": np.random.uniform(0.1, 0.5),
            "light_intensity": np.random.uniform(20, 100),
            "noise_level": np.random.uniform(0.1, 1.0),
```

```
            "social_density": np.random.uniform(10, 100)
        }

        # Genes for neuronal response sensitivities
        self.sensitivities = {
            "temperature": np.random.uniform(0.1, 2.0),
            "oxygen_level": np.random.uniform(0.5, 5.0),
            "food_availability": np.random.uniform(1.0, 10.0),
            "light_intensity": np.random.uniform(0.01, 0.2),
            "noise_level": np.random.uniform(-10.0, -1.0),
            "social_density": np.random.uniform(-1.0, 1.0)
        }

        # Genes for hormonal modulation
        self.hormone_base_levels = {
            "growth": np.random.uniform(0.5, 1.5),
            "defence": np.random.uniform(0.5, 1.5),
            "reproduction": np.random.uniform(0.5, 1.5),
            "respiration": np.random.uniform(0.5, 1.5),
            "exploration": np.random.uniform(0.5, 1.5)
        }

        # Genes for attention threshold
        self.attention_threshold = np.random.uniform(0.2, 0.5)

    def mutate(self, mutation_rate=0.1, mutation_size=0.2):
        """
        Mutate genes with given probability and magnitude

        Args:
            mutation_rate: Probability of each gene mutating
            mutation_size: Relative size of mutation
        """
        # Mutate thresholds
        for key in self.thresholds:
            if np.random.random() < mutation_rate:
                # Add random change
                change = np.random.normal(0, self.thresholds[key] * mutation_size)
                self.thresholds[key] += change

        # Mutate sensitivities
        for key in self.sensitivities:
            if np.random.random() < mutation_rate:
                change = np.random.normal(0, abs(self.sensitivities[key]) * mutation_size)
                self.sensitivities[key] += change

        # Mutate hormone base levels
        for key in self.hormone_base_levels:
            if np.random.random() < mutation_rate:
                change = np.random.normal(0, self.hormone_base_levels[key] * mutation_size)
                self.hormone_base_levels[key] += change

        # Mutate attention threshold
        if np.random.random() < mutation_rate:
            change = np.random.normal(0, self.attention_threshold * mutation_size)
            self.attention_threshold += change
            self.attention_threshold = max(0.1, min(0.9, self.attention_threshold))
```

7.2 Crossover and Selection

Implementing crossover for genetic mixing and selection for evolution:

```python
def crossover(parent1, parent2):
    """
    Create offspring by crossing over genes from two parents

    Args:
        parent1, parent2: SalmonGenes objects

    Returns:
        New SalmonGenes object
    """
    child = SalmonGenes()

    # Crossover thresholds
    for key in child.thresholds:
        # 50% chance of inheriting from each parent
        if np.random.random() < 0.5:
            child.thresholds[key] = parent1.thresholds[key]
        else:
            child.thresholds[key] = parent2.thresholds[key]

    # Crossover sensitivities
    for key in child.sensitivities:
        if np.random.random() < 0.5:
            child.sensitivities[key] = parent1.sensitivities[key]
        else:
            child.sensitivities[key] = parent2.sensitivities[key]

    # Crossover hormone base levels
    for key in child.hormone_base_levels:
        if np.random.random() < 0.5:
            child.hormone_base_levels[key] = parent1.hormone_base_levels[key]
        else:
            child.hormone_base_levels[key] = parent2.hormone_base_levels[key]

    # Crossover attention threshold
    if np.random.random() < 0.5:
        child.attention_threshold = parent1.attention_threshold
    else:
        child.attention_threshold = parent2.attention_threshold

    return child

def select_parents(population, fitness_scores, num_parents):
    """
    Select parents using tournament selection

    Args:
        population: List of SalmonGenes objects
        fitness_scores: List of fitness values matching population
        num_parents: Number of parents to select

    Returns:
        List of selected parent indexes
    """
    selected = []

    for _ in range(num_parents):
        # Tournament selection
        tournament_size = 3
        tournament = np.random.choice(
```

```
            len(population),
            size=tournament_size,
            replace=False)

        # Find the winner (highest fitness)
        winner_idx = tournament[0]
        winner_fitness = fitness_scores[tournament[0]]

        for idx in tournament[1:]:
            if fitness_scores[idx] > winner_fitness:
                winner_idx = idx
                winner_fitness = fitness_scores[idx]

        selected.append(winner_idx)

    return selected
```

7.3 Evolutionary Process

Putting together the evolutionary process:

```
class GeneticAlgorithm:
    """
    Implements the genetic algorithm for evolving digital twins
    """

    def __init__(self, pop_size=50, mutation_rate=0.1):
        self.pop_size = pop_size
        self.mutation_rate = mutation_rate
        self.population = [SalmonGenes() for _ in range(pop_size)]
        self.fitness_scores = np.zeros(pop_size)
        self.generation = 0

    def evolve(self, fitness_function):
        """
        Evolve the population for one generation

        Args:
            fitness_function: Function that takes SalmonGenes and returns
    fitness
        """
        # Evaluate fitness
        for i, genes in enumerate(self.population):
            self.fitness_scores[i] = fitness_function(genes)

        # Create new population
        new_population = []

        # Elitism: keep the best individual
        best_idx = np.argmax(self.fitness_scores)
        new_population.append(self.population[best_idx])

        # Select parents and create offspring
        while len(new_population) < self.pop_size:
            # Select parents
            parent_indices = select_parents(
                self.population, self.fitness_scores, 2)

            # Create offspring
            child = crossover(
                self.population[parent_indices[0]],
                self.population[parent_indices[1]]
            )
```

```
            # Apply mutation
            child.mutate(self.mutation_rate)

            # Add to new population
            new_population.append(child)

        # Replace old population
        self.population = new_population
        self.generation += 1

        # Return statistics
        return {
            "generation": self.generation,
            "best_fitness": np.max(self.fitness_scores),
            "avg_fitness": np.mean(self.fitness_scores),
            "worst_fitness": np.min(self.fitness_scores)
        }
```

7.4 Fitness Function Example

An example fitness function for evaluating digital twins:

```
def evaluate_twin_fitness(genes, num_trials=5):
    """
    Evaluate fitness of a digital twin with given genes

    Args:
        genes: SalmonGenes object
        num_trials: Number of simulation trials to run

    Returns:
        Fitness score
    """
    total_wellbeing = 0
    total_lifespan = 0

    for _ in range(num_trials):
        # Create digital twin with these genes
        twin = create_digital_twin_from_genes(genes)

        # Create environment
        env = Environment()

        # Run simulation until death or max time
        max_time = 1000
        for t in range(max_time):
            # Update environment
            env.step()

            # Update twin
            twin.update(env)

            # Check if twin died
            if twin.is_dead():
                break

        # Record results
        lifespan = t
        total_lifespan += lifespan
        total_wellbeing += twin.wellbeing_assessment.wellbeing_score * lifespan

    # Fitness is a combination of wellbeing and lifespan
```

```
    avg_wellbeing = total_wellbeing / total_lifespan if total_lifespan > 0 else
     0
    avg_lifespan = total_lifespan / num_trials

    # Weight wellbeing more than lifespan
    fitness = 0.7 * avg_wellbeing + 0.3 * (avg_lifespan / max_time)

    return fitness
```

8 Complete Digital Twin

Now we integrate all components into a complete digital twin.

8.1 Digital Twin Architecture

```
class DigitalTwin:
    """
    Complete implementation of a salmon digital twin
    """

    def __init__(self, genes=None):
        # Initialize genes
        self.genes = genes or SalmonGenes()

        # Create basic needs
        self.basic_needs = BasicNeeds()

        # Create sensors from genes
        self.sensors = self._create_sensors()

        # Create survival circuits
        self.survival_circuits = self._create_survival_circuits()

        # Create global organismic state
        self.gos = GlobalOrganismicState()

        # Create memory and learning
        self.episodic_memory = EpisodicMemory()
        self.learning = Learning()

        # Create prediction system
        self.prediction_error = PredictionError()

        # Create decision making
        self.decision_making = DecisionMaking(
            self.episodic_memory, self.learning)

        # Create wellbeing assessment
        self.wellbeing_assessment = WellbeingAssessment()

        # State variables
        self.age = 0
        self.health = 1.0
        self._dead = False

    def _create_sensors(self):
        """Create sensors based on genes"""
        sensors = {}

        # Create a sensor for each input type
        for input_name, threshold in self.genes.thresholds.items():
```

```python
            sensitivity = self.genes.sensitivities[input_name]
            response = NeuronalResponse(threshold, sensitivity)
            sensors[input_name] = Sensor(input_name, response)

        return sensors

    def _create_survival_circuits(self):
        """Create survival circuits based on genes"""
        circuits = []

        # Create growth circuit
        growth = SurvivalCircuit("growth")
        growth.add_sensor(self.sensors["temperature"])
        growth.add_sensor(self.sensors["food_availability"])
        growth.set_hormone_modulation(self.genes.hormone_base_levels["growth"])
        circuits.append(growth)

        # Create defence circuit
        defence = SurvivalCircuit("defence")
        defence.add_sensor(self.sensors["noise_level"])
        defence.set_hormone_modulation(self.genes.hormone_base_levels["defence"
    ])
        circuits.append(defence)

        # Create reproduction circuit
        reproduction = SurvivalCircuit("reproduction")
        reproduction.set_hormone_modulation(
            self.genes.hormone_base_levels["reproduction"])
        circuits.append(reproduction)

        # Create respiration circuit
        respiration = SurvivalCircuit("respiration")
        respiration.add_sensor(self.sensors["oxygen_level"])
        respiration.set_hormone_modulation(
            self.genes.hormone_base_levels["respiration"])
        circuits.append(respiration)

        # Create exploration circuit
        exploration = SurvivalCircuit("exploration")
        exploration.add_sensor(self.sensors["light_intensity"])
        exploration.set_hormone_modulation(
            self.genes.hormone_base_levels["exploration"])
        circuits.append(exploration)

        return circuits

    def update(self, environment):
        """
        Update the digital twin state based on environment

        Args:
            environment: Environment object
        """
        if self._dead:
            return

        # Increment age
        self.age += 1

        # Process inputs through survival circuits
        for circuit in self.survival_circuits:
            circuit.process_inputs(environment)

```

```python
        # Update global organismic state
        self.gos.update(
            self.survival_circuits,
            attention_threshold=self.genes.attention_threshold
        )

        # Check for prediction errors
        env_state = environment.get_state()
        for var, value in env_state.items():
            error = self.prediction_error.calculate_error(var, value)

        # Assess wellbeing
        stress = self.wellbeing_assessment.assess_stress(
            self.gos, self.prediction_error)

        boredom = self.wellbeing_assessment.assess_boredom(
            self.gos, self.prediction_error,
            self.decision_making.time_without_gos)

        # Get predicted emotions from memory
        predicted_emotions = {}
        # TODO: Implement emotion prediction

        wellbeing = self.wellbeing_assessment.assess_wellbeing(
            stress, boredom, predicted_emotions)

        # Make decisions
        available_actions = self._get_available_actions(environment)
        action = self.decision_making.decide(
            self.gos, environment, available_actions)

        # Execute action
        self._execute_action(action, environment)

        # Update health based on wellbeing
        if wellbeing < 0.2:
            # Poor wellbeing decreases health
            self.health -= 0.01
        elif wellbeing > 0.8:
            # Good wellbeing increases health (up to 1.0)
            self.health = min(1.0, self.health + 0.005)

        # Check if dead
        if self.health <= 0:
            self._dead = True

    def is_dead(self):
        """Check if the twin is dead"""
        return self._dead

    def _get_available_actions(self, environment):
        """Get available actions based on environment"""
        # Simplified actions
        return [
            "feed",
            "hide",
            "explore",
            "rest",
            "move_to_oxygen"
        ]

    def _execute_action(self, action, environment):
        """Execute selected action"""
```

```
        # In a real implementation, this would affect the environment
        # and provide feedback for learning
        reward = 0.0

        if action == "feed" and environment.food_availability > 0.5:
            reward = 0.5
        elif action == "hide" and environment.noise_level > 0.7:
            reward = 0.4
        elif action == "move_to_oxygen" and environment.oxygen_level < 6.0:
            reward = 0.6
        elif action == "explore" and self.decision_making.time_without_gos >
    50:
            reward = 0.3

        # Update learning
        self.decision_making.update_from_reward(reward)

    def get_wellbeing_report(self):
        """
        Get a complete wellbeing report

        Returns:
            Dictionary with wellbeing information
        """
        report = self.wellbeing_assessment.get_wellbeing_report()
        report.update({
            "age": self.age,
            "health": self.health,
            "gos_active": self.gos.active,
            "attention_focus": self.gos.attention_focus,
            "prediction_errors": self.prediction_error.get_average_error(),
            "time_without_gos": self.decision_making.time_without_gos
        })

        return report
```

8.2 Factory Function

A factory function to create digital twins from genes:

```
def create_digital_twin_from_genes(genes):
    """
    Create a digital twin from specific genes

    Args:
        genes: SalmonGenes object

    Returns:
        DigitalTwin object
    """
    return DigitalTwin(genes)
```

8.3 Population Simulation

Running a population of digital twins:

```
def simulate_population(population_size=10, simulation_steps=1000):
    """
    Simulate a population of digital twins

    Args:
        population_size: Number of twins to simulate
```

```
        simulation_steps: Number of time steps to run

    Returns:
        List of reports from all twins
    """
    # Create population
    twins = [DigitalTwin() for _ in range(population_size)]

    # Create environment
    env = Environment()

    # Storage for reports
    reports = []

    # Run simulation
    for step in range(simulation_steps):
        # Update environment
        env.step()

        # Update each twin
        step_reports = []
        for twin in twins:
            if not twin.is_dead():
                twin.update(env)
                report = twin.get_wellbeing_report()
                report["step"] = step
                step_reports.append(report)

        reports.append({
            "step": step,
            "environment": env.get_state(),
            "twin_reports": step_reports
        })

    return reports
```

9 Validation and Analysis

In this section, we implement methods to validate and analyze the digital twin.

9.1 Visualization Tools

```
def plot_wellbeing_over_time(reports):
    """
    Plot wellbeing metrics over time

    Args:
        reports: List of reports from simulation
    """
    steps = [r["step"] for r in reports]

    # Extract average metrics at each time step
    avg_wellbeing = []
    avg_stress = []
    avg_boredom = []

    for report in reports:
        twin_reports = report["twin_reports"]
        if twin_reports:
            wellbeing = [r["wellbeing_score"] for r in twin_reports]
            stress = [r["stress_level"] for r in twin_reports]
```

```
            boredom = [r["boredom_level"] for r in twin_reports]

            avg_wellbeing.append(np.mean(wellbeing))
            avg_stress.append(np.mean(stress))
            avg_boredom.append(np.mean(boredom))
        else:
            # No living twins
            avg_wellbeing.append(np.nan)
            avg_stress.append(np.nan)
            avg_boredom.append(np.nan)

    # Plot
    plt.figure(figsize=(12, 6))

    plt.plot(steps, avg_wellbeing, label="Wellbeing", color="green")
    plt.plot(steps, avg_stress, label="Stress", color="red")
    plt.plot(steps, avg_boredom, label="Boredom", color="blue")

    plt.xlabel("Time Step")
    plt.ylabel("Level")
    plt.title("Population Wellbeing Metrics Over Time")
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

9.2 Testing Scenarios

Implementing scenarios to test digital twin responses:

```
def test_stress_scenario():
    """
    Test digital twin response to stressful scenario
    """
    # Create twin
    twin = DigitalTwin()

    # Create environment with baseline conditions
    env = Environment(
        temperature=12,
        oxygen_level=9,
        light_intensity=100,
        food_availability=1.0,
        social_density=50,
        noise_level=0.1
    )

    # Simulate baseline period
    baseline_reports = []
    for _ in range(100):
        env.step()
        twin.update(env)
        baseline_reports.append(twin.get_wellbeing_report())

    # Introduce stressor - high noise and low oxygen
    env.noise_level = 0.9
    env.oxygen_level = 5.0

    # Simulate stress period
    stress_reports = []
    for _ in range(100):
        env.step()
```

```python
        twin.update(env)
        stress_reports.append(twin.get_wellbeing_report())

    # Recovery period
    env.noise_level = 0.1
    env.oxygen_level = 9.0

    recovery_reports = []
    for _ in range(100):
        env.step()
        twin.update(env)
        recovery_reports.append(twin.get_wellbeing_report())

    # Analyze results
    baseline_stress = np.mean([r["stress_level"] for r in baseline_reports])
    stress_stress = np.mean([r["stress_level"] for r in stress_reports])
    recovery_stress = np.mean([r["stress_level"] for r in recovery_reports])

    print(f"Baseline stress: {baseline_stress:.2f}")
    print(f"Stress period: {stress_stress:.2f}")
    print(f"Recovery period: {recovery_stress:.2f}")

    # Plot results
    all_reports = baseline_reports + stress_reports + recovery_reports
    steps = range(len(all_reports))
    stress = [r["stress_level"] for r in all_reports]
    wellbeing = [r["wellbeing_score"] for r in all_reports]

    plt.figure(figsize=(12, 6))

    plt.plot(steps, stress, label="Stress", color="red")
    plt.plot(steps, wellbeing, label="Wellbeing", color="green")

    plt.axvline(x=100, color="gray", linestyle="--")
    plt.axvline(x=200, color="gray", linestyle="--")

    plt.text(50, 0.9, "Baseline", ha="center")
    plt.text(150, 0.9, "Stress", ha="center")
    plt.text(250, 0.9, "Recovery", ha="center")

    plt.xlabel("Time Step")
    plt.ylabel("Level")
    plt.title("Stress Response Test")
    plt.legend()
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

9.3 Validation Functions

Functions to validate digital twin behavior against empirical data:

```python
def validate_against_empirical(twin_reports, empirical_data):
    """
    Compare digital twin predictions with empirical data

    Args:
        twin_reports: Reports from digital twin simulation
        empirical_data: Dictionary with empirical measurements

    Returns:
        Dictionary with validation metrics
```

```python
    """
    # Extract metrics from twin reports
    twin_stress = np.array([r["stress_level"] for r in twin_reports])
    twin_wellbeing = np.array([r["wellbeing_score"] for r in twin_reports])

    # Compare with empirical data
    empirical_stress = np.array(empirical_data["stress_measurements"])
    empirical_wellbeing = np.array(empirical_data["wellbeing_indicators"])

    # Compute correlation
    stress_correlation = np.corrcoef(twin_stress, empirical_stress)[0, 1]
    wellbeing_correlation = np.corrcoef(twin_wellbeing, empirical_wellbeing)[0,
     1]

    # Compute mean absolute error
    stress_mae = np.mean(np.abs(twin_stress - empirical_stress))
    wellbeing_mae = np.mean(np.abs(twin_wellbeing - empirical_wellbeing))

    # Return validation metrics
    return {
        "stress_correlation": stress_correlation,
        "wellbeing_correlation": wellbeing_correlation,
        "stress_mae": stress_mae,
        "wellbeing_mae": wellbeing_mae
    }
```

10 Practical Applications

In this section, we provide practical examples of using the digital twin for real-world applications.

10.1 Early Warning System

Implementing an early warning system for detecting wellbeing issues:

```python
class WellbeingMonitor:
    """
    Monitor that provides early warnings about wellbeing issues
    """

    def __init__(self, population, warning_thresholds=None):
        self.population = population  # List of DigitalTwin objects

        # Default warning thresholds
        self.thresholds = warning_thresholds or {
            "high_stress": 0.7,
            "chronic_stress": 0.6,
            "high_boredom": 0.7,
            "low_wellbeing": 0.3
        }

        self.alerts = []

    def update(self, environment):
        """
        Update all twins and check for warnings

        Args:
            environment: Current environment

        Returns:
            List of alerts
```

```
        """
        reports = []

        # Update each twin
        for twin in self.population:
            if not twin.is_dead():
                twin.update(environment)
                reports.append(twin.get_wellbeing_report())

        # Check for warnings
        new_alerts = self._check_warnings(reports)
        self.alerts.extend(new_alerts)

        return new_alerts

    def _check_warnings(self, reports):
        """Check for warning conditions in the reports"""
        alerts = []

        # Check for high stress
        high_stress_count = sum(1 for r in reports
                                    if r["stress_level"] >= self.thresholds["high_stress"])
        if high_stress_count > len(reports) * 0.3:  # >30% of population
            alerts.append({
                "type": "high_stress",
                "severity": "high",
                "affected_percentage": high_stress_count / len(reports) * 100,
                "description": "High stress levels detected in significant portion of population"
            })

        # Check for chronic stress (sustained mid-level stress)
        chronic_stress_count = sum(1 for r in reports
                                       if r["stress_level"] >= self.thresholds["chronic_stress"])
        if chronic_stress_count > len(reports) * 0.5:  # >50% of population
            alerts.append({
                "type": "chronic_stress",
                "severity": "medium",
                "affected_percentage": chronic_stress_count / len(reports) * 100,
                "description": "Chronic stress detected - may lead to health issues"
            })

        # Check for high boredom
        high_boredom_count = sum(1 for r in reports
                                     if r["boredom_level"] >= self.thresholds["high_boredom"])
        if high_boredom_count > len(reports) * 0.4:  # >40% of population
            alerts.append({
                "type": "high_boredom",
                "severity": "medium",
                "affected_percentage": high_boredom_count / len(reports) * 100,
                "description": "High boredom levels detected - may impair learning and development"
            })

        # Check for low wellbeing
        low_wellbeing_count = sum(1 for r in reports
                                      if r["wellbeing_score"] <= self.thresholds["low_wellbeing"])
```

```
        if low_wellbeing_count > len(reports) * 0.3:  # >30% of population
            alerts.append({
                "type": "low_wellbeing",
                "severity": "high",
                "affected_percentage": low_wellbeing_count / len(reports) * 100,
                "description": "Low wellbeing detected - immediate attention required"
            })

        return alerts
```

10.2 Facility Optimization

Using the digital twin to optimize aquaculture facility parameters:

```
def optimize_facility_parameters(parameter_ranges, population_size=20,
                               simulation_days=30, steps_per_day=24):
    """
    Find optimal facility parameters for salmon wellbeing

    Args:
        parameter_ranges: Dictionary with min/max for each parameter
        population_size: Number of digital twins to simulate
        simulation_days: Number of days to simulate
        steps_per_day: Simulation steps per day

    Returns:
        Dictionary with optimal parameter values
    """
    best_score = -float('inf')
    best_params = None

    # Number of optimization iterations
    iterations = 50

    for iteration in range(iterations):
        # Sample parameters from ranges
        params = {}
        for param, (min_val, max_val) in parameter_ranges.items():
            params[param] = np.random.uniform(min_val, max_val)

        print(f"Testing parameters: {params}")

        # Create environment with these parameters
        env = Environment(
            temperature=params.get("temperature", 12),
            oxygen_level=params.get("oxygen_level", 8.5),
            light_intensity=params.get("light_intensity", 100),
            food_availability=params.get("food_availability", 1.0),
            social_density=params.get("social_density", 50),
            noise_level=params.get("noise_level", 0.1)
        )

        # Create population
        twins = [DigitalTwin() for _ in range(population_size)]

        # Run simulation
        total_steps = simulation_days * steps_per_day
        wellbeing_scores = []
        stress_scores = []
        mortality = 0

```

```
        for step in range(total_steps):
            # Update environment (with small variation)
            env.step()

            # Update each twin
            for twin in twins:
                if not twin.is_dead():
                    twin.update(env)
                    report = twin.get_wellbeing_report()
                    wellbeing_scores.append(report["wellbeing_score"])
                    stress_scores.append(report["stress_level"])
                else:
                    mortality += 1

        # Calculate performance score
        avg_wellbeing = np.mean(wellbeing_scores) if wellbeing_scores else 0
        avg_stress = np.mean(stress_scores) if stress_scores else 1
        survival_rate = 1 - (mortality / (population_size * total_steps))

        # Combined score (higher is better)
        score = (0.5 * avg_wellbeing) + (0.3 * (1 - avg_stress)) + (0.2 * survival_rate)

        print(f"Score: {score:.4f} (wellbeing: {avg_wellbeing:.2f}, " +
              f"stress: {avg_stress:.2f}, survival: {survival_rate:.2f})")

        # Update best parameters
        if score > best_score:
            best_score = score
            best_params = params.copy()

    print(f"\nOptimal parameters found: {best_params}")
    print(f"Optimization score: {best_score:.4f}")

    return best_params
```

10.3 Policy Analysis

Analyzing the impact of different husbandry policies:

```
def analyze_husbandry_policy(policy, population_size=50, simulation_days=60):
    """
    Analyze the impact of a husbandry policy on salmon wellbeing

    Args:
        policy: Dictionary defining the policy
        population_size: Number of digital twins to simulate
        simulation_days: Number of days to simulate

    Returns:
        Dictionary with analysis results
    """
    # Create population
    twins = [DigitalTwin() for _ in range(population_size)]

    # Create environment
    env = Environment()

    # Policy implementation
    feeding_schedule = policy.get("feeding_schedule", "regular")  # regular, variable
    light_regime = policy.get("light_regime", "natural")  # natural, constant, gradual
```

```python
    handling_frequency = policy.get("handling_frequency", "low")  # low, medium, high

    # Simulation parameters
    steps_per_day = 24
    total_steps = simulation_days * steps_per_day

    # Storage for metrics
    daily_metrics = []

    # Run simulation
    for day in range(simulation_days):
        day_metrics = {
            "day": day,
            "wellbeing": [],
            "stress": [],
            "boredom": [],
            "alive_count": 0
        }

        for step in range(steps_per_day):
            current_step = day * steps_per_day + step

            # Apply policy
            self._apply_policy(env, policy, day, step)

            # Update environment
            env.step()

            # Handle scheduled events
            if handling_frequency == "high" and current_step % 48 == 0:
                # Simulate handling stress
                env.noise_level = 0.9
            elif handling_frequency == "medium" and current_step % 120 == 0:
                env.noise_level = 0.9
            elif handling_frequency == "low" and current_step % 336 == 0:
                env.noise_level = 0.9
            else:
                # Return to baseline
                env.noise_level = policy.get("baseline_noise", 0.1)

            # Update all twins
            for twin in twins:
                if not twin.is_dead():
                    twin.update(env)
                    report = twin.get_wellbeing_report()

                    # Store metrics
                    day_metrics["wellbeing"].append(report["wellbeing_score"])
                    day_metrics["stress"].append(report["stress_level"])
                    day_metrics["boredom"].append(report["boredom_level"])
                    day_metrics["alive_count"] += 1

        # Calculate daily averages
        day_metrics["avg_wellbeing"] = np.mean(day_metrics["wellbeing"]) if day_metrics["wellbeing"] else 0
        day_metrics["avg_stress"] = np.mean(day_metrics["stress"]) if day_metrics["stress"] else 0
        day_metrics["avg_boredom"] = np.mean(day_metrics["boredom"]) if day_metrics["boredom"] else 0
        day_metrics["survival_rate"] = day_metrics["alive_count"] / (population_size * steps_per_day)

```

```python
            daily_metrics.append(day_metrics)

    # Calculate overall metrics
    avg_wellbeing = np.mean([d["avg_wellbeing"] for d in daily_metrics])
    avg_stress = np.mean([d["avg_stress"] for d in daily_metrics])
    avg_boredom = np.mean([d["avg_boredom"] for d in daily_metrics])
    final_survival = daily_metrics[-1]["survival_rate"] if daily_metrics else 0

    # Prepare report
    report = {
        "policy": policy,
        "avg_wellbeing": avg_wellbeing,
        "avg_stress": avg_stress,
        "avg_boredom": avg_boredom,
        "final_survival_rate": final_survival,
        "daily_metrics": daily_metrics
    }

    return report

def _apply_policy(env, policy, day, hour):
    """Apply policy effects to environment"""
    # Feeding schedule
    feeding_schedule = policy.get("feeding_schedule", "regular")
    if feeding_schedule == "regular":
        # Regular feeding at fixed times
        if hour == 8 or hour == 16:
            env.food_availability = 1.0
        else:
            env.food_availability = 0.1
    elif feeding_schedule == "variable":
        # Variable feeding (unpredictable)
        if hour == (day % 24) or hour == ((day + 12) % 24):
            env.food_availability = 1.0
        else:
            env.food_availability = 0.1

    # Light regime
    light_regime = policy.get("light_regime", "natural")
    if light_regime == "natural":
        # Natural light cycle
        env.light_intensity = 100 * np.sin(np.pi * hour / 12) ** 2
    elif light_regime == "constant":
        # Constant light
        env.light_intensity = 100
    elif light_regime == "gradual":
        # Gradual changes
        if hour < 6:
            env.light_intensity = hour * 16.67  # 0 to 100 over 6 hours
        elif hour < 18:
            env.light_intensity = 100
        else:
            env.light_intensity = 100 - ((hour - 18) * 16.67)  # 100 to 0 over
    6 hours
```

10.4 Integrating with Environmental Sensors

Framework for integrating digital twins with real-time sensor data:

```python
class SensorIntegration:
    """
    Integration with real-time environmental sensors
    """
```

```python

    def __init__(self, sensor_config, twins):
        self.sensor_config = sensor_config  # Mapping of sensor IDs to parameters
        self.twins = twins
        self.last_readings = {}
        self.history = []

    def process_sensor_data(self, sensor_readings):
        """
        Process incoming sensor data

        Args:
            sensor_readings: Dictionary with sensor readings

        Returns:
            List of alerts
        """
        # Store readings
        self.last_readings = sensor_readings
        self.history.append({
            "timestamp": time.time(),
            "readings": sensor_readings.copy()
        })

        # Convert sensor readings to environment parameters
        env_params = self._convert_to_environment(sensor_readings)

        # Create environment
        env = Environment(**env_params)

        # Update twins and check for warnings
        monitor = WellbeingMonitor(self.twins)
        alerts = monitor.update(env)

        return alerts

    def _convert_to_environment(self, sensor_readings):
        """Convert raw sensor readings to environment parameters"""
        env_params = {}

        # Map sensor readings to environment parameters
        for sensor_id, value in sensor_readings.items():
            if sensor_id in self.sensor_config:
                param = self.sensor_config[sensor_id]["parameter"]

                # Apply any conversion formula
                if "conversion" in self.sensor_config[sensor_id]:
                    conversion = self.sensor_config[sensor_id]["conversion"]
                    if conversion == "linear":
                        a = self.sensor_config[sensor_id].get("a", 1.0)
                        b = self.sensor_config[sensor_id].get("b", 0.0)
                        value = a * value + b

                env_params[param] = value

        return env_params

    def get_history(self, start_time=None, end_time=None):
        """
        Get historical data within time range

        Args:
```

```
            start_time: Start timestamp (or None for all)
            end_time: End timestamp (or None for all)

        Returns:
            List of historical readings
        """
        filtered = []

        for record in self.history:
            timestamp = record["timestamp"]
            if (start_time is None or timestamp >= start_time) and \
               (end_time is None or timestamp <= end_time):
                filtered.append(record)

        return filtered
```

11 Discussion and Future Directions

11.1 Limitations of the Current Approach

While the digital twin provides valuable insights, several limitations should be acknowledged:

- **Simplification of Biology:** The model simplifies complex biological processes that may be important for accurate wellbeing prediction.
- **Parameter Uncertainty:** Many parameters (e.g., neuronal response sensitivities) are difficult to calibrate against real salmon.
- **Cognitive Assumptions:** The implementation makes assumptions about salmon cognition that may need refinement as research advances.
- **Validation Challenges:** Validating subjective states like wellbeing against empirical data presents methodological challenges.

11.2 Future Research Directions

Several promising research directions could enhance the digital twin approach:

- **Integration with Physiological Models:** Incorporating more detailed physiological models would improve prediction accuracy.
- **Individual Variation:** Expanding the representation of individual variation in neuronal responses and behavior.
- **Social Dynamics:** Including social interactions and hierarchies within the salmon population.
- **Explainable AI:** Developing methods to better explain the relationship between environmental factors and wellbeing outcomes.
- **Model Validation:** Conducting targeted experiments to validate specific aspects of the digital twin predictions.

11.3 Ethical Considerations

The development and use of digital twins for salmon wellbeing raises several ethical considerations:

- **Reliability:** Ensuring the reliability of digital twin predictions before using them to make decisions about real animals.
- **Transparency:** Being transparent about model assumptions and limitations when reporting results.
- **Balance:** Balancing economic considerations with animal welfare in aquaculture operations.
- **Responsibility:** Using digital twins to enhance rather than replace human responsibility for animal welfare.

12 Conclusion

This tutorial has provided a comprehensive implementation of digital twins for modeling salmon wellbeing based on the conceptual framework described by Giske et al. (2025). The approach combines insights from neuroscience, behavioral ecology, and computational modeling to create virtual representatives of salmon that can predict stress, boredom, and overall wellbeing.

By implementing survival circuits, episodic-like memory, and wellbeing assessment systems, the digital twin captures the key mechanisms underlying salmon behavior and experience. The evolutionary framework enables the model to adapt and improve over time, making it more representative of real salmon populations.

The practical applications of this approach are numerous, from optimizing aquaculture facilities to developing early warning systems for wellbeing issues. By providing actionable information to fish farmers, regulators, and researchers, digital twins can support the implementation of the 3Rs (replacement, reduction, refinement) in animal research and improve the welfare of farmed salmon.

As research in animal cognition and wellbeing continues to advance, the digital twin approach can be refined and extended to provide even more accurate predictions. This creates a positive feedback loop where digital simulations inform empirical research, which in turn improves the simulations.

The ultimate goal is to create a tool that benefits both salmon welfare and aquaculture productivity, demonstrating that these objectives can be aligned rather than in conflict. By understanding and predicting wellbeing at a deeper level, we can create conditions where salmon thrive rather than merely survive in captivity.

13 Appendix: Complete Code Repository

The complete implementation code discussed in this tutorial is available in the accompanying GitHub repository: `https://github.com/arvidl/salmon-digital-twin`

The repository includes:

- Core implementation files (`notebooks`)
- Example scripts
- Test scenarios
- Documentation (`papers`)

- Sample data for validation (`data`)
- Conda environment (`environment.yml`)

13.1 Installation and Usage

To install and use the `salmon-digital-twin`:

```
# Clone repository
git clone https://github.com/arvidl/salmon-digital-twin.git
cd salmon-digital-twin

# Install dependencies
#pip install -r requirements.txt
conda env update -f environment.yml

# Run example simulation
#python examples/run_simulation.py
```

13.2 Contributing

Contributions to the project are welcome. Please see the contribution guidelines in the repository for more information.

Annotated Reference Guide

This section provides an annotated guide to key references organized by topic area, highlighting their relevance to digital twin implementation for salmon wellbeing.

Core Frameworks and Concepts

- **Giske et al. (2025)** [31] – The foundational paper on salmon digital twins that outlines the theoretical basis for monitoring and predicting salmon wellbeing through computational modeling.
- **Budaev et al. (2020)** [10] – Introduces a computational architecture for modeling animal sentience, emotions, and wellbeing that serves as a basis for digital twin development.
- **Budaev et al. (2019)** [11] – Bridges ecology and subjective cognition in animal decision-making, providing a framework for implementing cognition in digital twins.
- **Budaev et al. (2018)** [9] – Introduces the AHA (Adapted Heuristics and Architecture) cognitive architecture for Darwinian agents that can be adapted for salmon digital twins.

Fish Cognition and Emotions

- **Giske et al. (2013)** [30] – Examines how emotions affect adaptive behavior in fish, providing insight into implementing emotional systems in digital twins.
- **Vindas et al. (2016)** [70] – Investigates serotonergic activation in farmed salmon, distinguishing between adaptation and pathology in stress responses.
- **Vindas et al. (2014)** [71] – Explores dopaminergic and neurotrophic responses in salmon when expected rewards are omitted, relevant for implementing prediction errors in digital twins.

- **Cabanac (1992)** [13] – Presents the concept of pleasure as a common currency for decision-making across species, informing the implementation of wellbeing assessment.
- **Mendl & Paul (2020)** [44] – Reviews current understanding of animal affect and its role in decision-making, providing a foundation for modeling emotional states.
- **Crump et al. (2020)** [16] – Examines the role of emotion in animal contests, offering insights into modeling social interactions and competitive behavior.

Consciousness and Sentience

- **Ginsburg & Jablonka (2019)** [29] – Comprehensive exploration of the evolution of consciousness and learning, providing theoretical foundation for digital twin cognition.
- **Low et al. (2012)** [39] – The Cambridge Declaration on Consciousness, affirming the presence of consciousness in non-human animals including fish.
- **Andrews et al. (2024)** [3] – The New York Declaration on Animal Consciousness, updating scientific consensus on animal consciousness.
- **Barron & Klein (2016)** [6] – Examines what insects can tell us about consciousness origins, providing insights for implementing minimal consciousness models.
- **Feinberg & Mallatt (2016)** [22] – Explores the ancient origins of consciousness and how the brain created experience, informing digital twin cognitive models.
- **Seth (2021)** [61] – Presents new scientific approaches to understanding consciousness, offering perspectives for modeling subjective experience.
- **Zacks et al. (2022)** [76] – Investigates the evolution of imaginative animals and episodic-like memory, crucial for modeling prediction in digital twins.

Digital Twins and Computational Modeling

- **Rasheed et al. (2020)** [52] – Reviews digital twin values, challenges, and enablers from a modeling perspective, providing practical implementation guidance.
- **VanderHorn & Mahadevan (2021)** [69] – Offers a framework for digital twin characterization and implementation applicable to biological systems.
- **Tao et al. (2022)** [67] – Presents comprehensive approaches to digital twin modeling that can be adapted for biological applications.
- **Eliassen et al. (2016)** [20] – Demonstrates how to model proximate architecture for decision-making from sensing to emergent adaptations.
- **Giske et al. (2014)** [32] – Shows how emotion systems promote diversity and evolvability in evolutionary models, informing genetic algorithm implementation.
- **Andersen et al. (2016)** [1] – Details the proximate architecture for decision-making in fish that can be directly implemented in digital twins.
- **Grimm & Railsback (2013)** [34] – Provides foundational methods for individual-based modeling in ecology applicable to digital twin populations.

Neuroscience and Decision-Making

- **LeDoux (2012)** [38] – Rethinks the emotional brain, introducing concepts like survival circuits central to digital twin decision architecture.
- **Anderson & Adolphs (2014)** [2] – Presents a framework for studying emotions across species that can be applied to salmon emotion modeling.
- **Schultz (2024)** [59] – Examines dopamine mechanisms for reward maximization, crucial for implementing learning in digital twins.
- **Friston et al. (2010)** [26] – Introduces free-energy formulations for action and behavior that inform prediction-based decision making.
- **Peters et al. (2017)** [49] – Explores how uncertainty and stress are processed by the brain, informing stress modeling in digital twins.
- **McNamara & Houston (1986)** [42] – Classic paper on common currency for behavioral decisions that informs wellbeing-based decision models.
- **McNamara & Houston (2009)** [41] – Discusses integrating function and mechanism in behavioral models, relevant for digital twin architecture.

Stress, Allostasis, and Boredom

- **Korte et al. (2007)** [37] – Presents a new animal welfare concept based on allostasis that informs wellbeing modeling in digital twins.
- **Sterling (2012)** [64] – Details allostasis as a model of predictive regulation central to digital twin physiological modeling.
- **McEwen et al. (2015)** [40] – Explores mechanisms of stress in the brain that can be implemented in digital twin stress response systems.
- **Wingfield et al. (1998)** [75] – Introduces the "emergency life history stage" concept relevant for modeling extreme stress in digital twins.
- **Meagher (2019)** [43] – Examines whether boredom is an animal welfare concern, providing foundation for modeling boredom in digital twins.
- **Burn (2017)** [12] – Presents a biological perspective on animal boredom with suggestions for scientific investigation.
- **Spruijt et al. (2001)** [63] – Offers a concept of welfare based on reward mechanisms applicable to digital twin wellbeing assessment.

Environmental Enrichment and Learning

- **Salvanes et al. (2013)** [57] – Demonstrates how environmental enrichment promotes neural plasticity and cognitive ability in fish.
- **Zupanc (2006)** [77] – Explores neurogenesis and neuronal regeneration in adult fish brains, relevant for modeling brain development.
- **Näslund et al. (2019)** [46] – Investigates how rearing environment affects brain development in hatchery-reared Atlantic salmon.

- **Arechavala-Lopez et al. (2022)** [4] – Reviews environmental enrichment in fish aquaculture, offering practical applications.
- **Folkedal et al. (2010)** [24] – Studies habituation rates in Atlantic salmon, providing data for learning implementation.
- **Bratland et al. (2010)** [8] – Examines the transition from fright to anticipation in salmon, informing predictive models.
- **Dumitru & Opdal (2024)** [18] – Discusses how rearing environment defines brain plasticity, challenging the mosaic model of brain evolution.

Aquaculture Welfare and Management

- **Stien et al. (2013)** [65] – Introduces the Salmon Welfare Index Model (SWIM 1.0) that catalogs key welfare indicators.
- **Pettersen et al. (2014)** [50] – Presents SWIM 2.0, an extended model for overall welfare assessment of caged Atlantic salmon.
- **Overton et al. (2019)** [48] – Reviews salmon lice treatments and mortality in Norwegian aquaculture, highlighting welfare challenges.
- **Bracke et al. (1999)** [7] – Presents overall animal welfare assessment based on needs and expert opinion.
- **van de Vis et al. (2020)** [72] – Compares welfare of fishes in different production systems, providing benchmarks for digital twin validation.
- **Dawkins (2023)** [17] – Discusses farm animal welfare beyond "natural" behavior, offering perspectives for defining appropriate wellbeing metrics.
- **Segner et al. (2019)** [60] – Presents FAO's approach to welfare of fishes in aquaculture, providing regulatory context.

Precision Aquaculture and Monitoring

- **Føre et al. (2018)** [25] – Introduces precision fish farming as a framework for improving production in aquaculture.
- **Mustapha et al. (2021)** [45] – Reviews roles of cloud computing, Internet of Things and AI in sustainable aquaculture.
- **Royer & Pastres (2023)** [54] – Demonstrates data assimilation for efficient management of dissolved oxygen in aquaculture.
- **Eguiraun et al. (2018)** [19] – Applies Shannon entropy to construct a biological warning system model for fish monitoring.
- **Neethirajan (2021)** [47] – Reviews the use of AI in assessing affective states in livestock, with potential applications for fish.

3Rs and Ethical Considerations

- **Russell & Burch (1959)** [56] – The original work introducing the 3Rs (replacement, reduction, refinement) principles.
- **Grimm et al. (2023)** [33] – Discusses advancing the 3Rs through innovation, implementation, ethics, and society.
- **Hawkins et al. (2011)** [35] – Provides guidance on severity classification of scientific procedures involving fish.
- **Sloman et al. (2019)** [62] – Examines ethical considerations in fish research, offering guidelines for digital twin development.
- **Collins & Part (2013)** [15] – Reviews approaches to modeling farm animal welfare, including methodological considerations.
- **Pielke (2007)** [51] – Introduces the concept of the "honest broker" in science and policy that digital twins could fulfill.
- **Gaffney & Lavery (2022)** [27] – Identifies research gaps in salmonid welfare that digital twins could address.

Machine Learning and Data Science

- **Reichstein et al. (2024)** [53] – Demonstrates early warning of complex risk with integrated AI, applicable to wellbeing monitoring.
- **Schölkopf et al. (2021)** [58] – Discusses approaches to causal representation learning applicable to digital twin modeling.
- **Elkan (2001)** [21] – Explores foundations of cost-sensitive learning for handling imbalanced wellbeing states.
- **Garcia & Fernández (2015)** [28] – Provides a comprehensive survey on safe reinforcement learning applicable to digital twin development.

Robustness and Systems Approaches

- **Kitano (2004)** [36] – Explores biological robustness concepts applicable to digital twin design.
- **Fernandez-Leon (2011)** [23] – Examines evolving cognitive-behavioral dependencies for robustness in situated agents.
- **Ruiz-Mirazo et al. (2004)** [55] – Discusses autonomy and open-ended evolution as universal life properties relevant to digital twin design.
- **Thompson (2007)** [68] – Presents a biology and phenomenology approach to mind that can inform digital twin consciousness models.
- **Colditz (2023)** [14] – Proposes a biological integrity framework for describing animal welfare and wellbeing.

Methodological Resources

- **Way (2017)** [73] – Discusses Feynman's famous quote "What I cannot create, I do not understand" in the context of biological modeling.
- **Taborsky et al. (2021)** [66] – Presents an evolutionary theory of stress responses that can inform digital twin stress models.
- **Wingfield (2013)** [74] – Explores comparative biology of environmental stress and ability to cope with changing environments.
- **Barrett (2020)** [5] – Provides seven and a half lessons about the brain with implications for cognitive modeling.

References

[1] Bjørn S Andersen et al. "The proximate architecture for decision-making in fish". In: *Fish and Fisheries* 17.3 (2016), pp. 680–695. DOI: 10.1111/faf.12139.

[2] David J Anderson and Ralph Adolphs. "A framework for studying emotions across species". In: *Cell* 157.1 (2014), pp. 187–200. DOI: 10.1016/j.cell.2014.03.003.

[3] Kristin Andrews et al. "The New York Declaration on Animal Consciousness". In: (2024). URL: https://www.nydeclaration.com/.

[4] Pablo Arechavala-Lopez et al. "Environmental enrichment in fish aquaculture: A review of fundamental and practical aspects". In: *Reviews in Aquaculture* 14.2 (2022), pp. 704–728. DOI: 10.1111/raq.12620.

[5] Lisa Feldman Barrett. "Seven and a half lessons about the brain". In: *Houghton Mifflin Harcourt* (2020).

[6] Andrew B Barron and Colin Klein. "What insects can tell us about the origins of consciousness". In: *Proceedings of the National Academy of Sciences* 113.18 (2016), pp. 4900–4908. DOI: 10.1073/pnas.1520084113.

[7] MBM Bracke, BM Spruijt, and JHM Metz. "Overall animal welfare reviewed. Part 3: welfare assessment based on needs and supported by expert opinion". In: *NJAS wageningen journal of life sciences* 47.3 (1999), pp. 307–322. DOI: 10.18174/njas.v47i3.468.

[8] Stine Bratland et al. "From fright to anticipation: using aversive light stimuli to investigate reward conditioning in large groups of Atlantic salmon (Salmo salar)". In: *Aquaculture International* 18.6 (2010), pp. 991–1001. DOI: 10.1007/s10499-009-9317-8.

[9] Sergey Budaev, Jarl Giske, and Sigrunn Eliassen. "AHA: A general cognitive architecture for Darwinian agents". In: *Biologically Inspired Cognitive Architectures* 25 (2018), pp. 51–57. DOI: 10.1016/j.bica.2018.07.009.

[10] Sergey Budaev et al. "Computational animal welfare: towards cognitive architecture models of animal sentience, emotion and wellbeing". In: *Royal Society Open Science* 7.7 (2020), p. 201886. DOI: 10.1098/rsos.201886.

[11] Sergey Budaev et al. "Decision-making from the animal perspective: bridging ecology and subjective cognition". In: *Frontiers in Ecology and Evolution* 7 (2019), p. 164. DOI: 10.3389/fevo.2019.00164.

[12] Charlotte C Burn. "Bestial boredom: A biological perspective on animal boredom and suggestions for its scientific investigation". In: *Animal Behaviour* 130 (2017), pp. 141–151. DOI: 10.1016/j.anbehav.2017.06.006.

[13] Michel Cabanac. “Pleasure: the common currency”. In: *Journal of theoretical Biology* 155.2 (1992), pp. 173–200. DOI: 10.1016/S0022-5193(05)80594-6.

[14] Ian G Colditz. “A biological integrity framework for describing animal welfare and wellbeing”. In: *Animal Production Science* 63.5 (2023), pp. 423–440. DOI: 10.1071/AN22285.

[15] Lisa M Collins and Chérie E Part. “Modelling farm animal welfare”. In: *Animals* 3.2 (2013), pp. 416–441. DOI: 10.3390/ani3020416.

[16] Andrew Crump et al. “Emotion in animal contests”. In: *Proceedings of the Royal Society B* 287.1939 (2020), p. 20201715. DOI: 10.1098/rspb.2020.1715.

[17] Marian Stamp Dawkins. “Farm animal welfare: Beyond "natural" behavior”. In: *Science* 379.6630 (2023), pp. 326–328. DOI: 10.1126/science.ade5437.

[18] Magda L Dumitru and Anders M F Opdal. “Beyond the mosaic model of brain evolution: Rearing environment defines local and global plasticity”. In: *Annals of the New York Academy of Sciences* 1542.1 (2024), pp. 58–66. DOI: 10.1111/nyas.15267.

[19] Harkaitz Eguiraun et al. “Reducing the number of individuals to monitor shoaling fish systems- Application of the Shannon entropy to construct a biological warning system model”. In: *Frontiers in physiology* 9 (2018), p. 493. DOI: 10.3389/fphys.2018.00493.

[20] Sigrunn Eliassen et al. “From sensing to emergent adaptations: Modelling the proximate architecture for decision-making”. In: *Ecological modelling* 326 (2016), pp. 90–100. DOI: 10.1016/j.ecolmodel.2015.09.001.

[21] Charles Elkan. “The foundations of cost-sensitive learning”. In: *International joint conference on artificial intelligence* 17.1 (2001), pp. 973–978.

[22] Todd E Feinberg and Jon M Mallatt. “The ancient origins of consciousness: How the brain created experience”. In: *MIT Press* (2016).

[23] Jose A Fernandez-Leon. “Evolving cognitive-behavioural dependencies in situated agents for behavioural robustness”. In: *Biosystems* 106.2-3 (2011), pp. 94–110. DOI: 10.1016/j.biosystems.2011.07.003.

[24] Ole Folkedal et al. “Habituation rate and capacity of Atlantic salmon (Salmo salar) parr to sudden transitions from darkness to light”. In: *Aquaculture* 307.1-2 (2010), pp. 170–172. DOI: 10.1016/j.aquaculture.2010.06.001.

[25] Martin Føre et al. “Precision fish farming: A new framework to improve production in aquaculture”. In: *Biosystems engineering* 173 (2018), pp. 176–193. DOI: 10.1016/j.biosystemseng.2017.10.014.

[26] Karl Friston et al. “Action and behavior: a free-energy formulation”. In: *Biological cybernetics* 102.3 (2010), pp. 227–260. DOI: 10.1007/s00422-010-0364-z.

[27] Lauren P Gaffney and James M Lavery. “Research before policy: identifying gaps in salmonid welfare research that require further study to inform evidence-based aquaculture guidelines in Canada”. In: *Frontiers in Veterinary Science* 8 (2022), p. 768558. DOI: 10.3389/fvets.2021.768558.

[28] Javier García and Fernando Fernández. “A comprehensive survey on safe reinforcement learning”. In: *Journal of Machine Learning Research* 16 (2015), pp. 1437–1480.

[29] Simona Ginsburg and Eva Jablonka. “The evolution of the sensitive soul: Learning and the origins of consciousness”. In: *MIT Press* (2019).

[30] Jarl Giske et al. “Effects of the emotion system on adaptive behavior”. In: *The American Naturalist* 182.6 (2013), pp. 689–703. DOI: 10.1086/673533.

[31] Jarl Giske et al. “Premises for digital twins reporting on Atlantic salmon wellbeing”. In: *Behavioural Processes* 226 (2025), p. 105163. DOI: 10.1016/j.beproc.2025.105163.

[32] Jarl Giske et al. "The emotion system promotes diversity and evolvability". In: *Proceedings of the Royal Society B: Biological Sciences* 281.1791 (2014), p. 20141096. DOI: 10.1098/rspb.2014.1096.

[33] Herwig Grimm et al. "Advancing the 3Rs: innovation, implementation, ethics and society". In: *Frontiers in Veterinary Science* 10 (2023), p. 1185706. DOI: 10.3389/fvets.2023.1185706.

[34] Volker Grimm and Steven F Railsback. "Individual-based modeling and ecology". In: *Princeton University Press* (2013).

[35] Penny Hawkins et al. "Guidance on the severity classification of scientific procedures involving fish: report of a Working Group appointed by the Norwegian Consensus-Platform for the Replacement, Reduction and Refinement of animal experiments (Norecopa)". In: *Laboratory Animals* 45.4 (2011), pp. 219–224. DOI: 10.1258/la.2011.010181.

[36] Hiroaki Kitano. "Biological robustness". In: *Nature Reviews Genetics* 5.11 (2004), pp. 826–837. DOI: 10.1038/nrg1471.

[37] S Mechiel Korte, Berend Olivier, and Jaap M Koolhaas. "A new animal welfare concept based on allostasis". In: *Physiology & behavior* 92.3 (2007), pp. 422–428. DOI: 10.1016/j.physbeh.2006.10.018.

[38] Joseph LeDoux. "Rethinking the emotional brain". In: *Neuron* 73.4 (2012), pp. 653–676. DOI: 10.1016/j.neuron.2012.02.004.

[39] Philip Low et al. "The Cambridge declaration on consciousness". In: *Francis Crick Memorial Conference, Cambridge, England* (2012).

[40] Bruce S McEwen et al. "Mechanisms of stress in the brain". In: *Nature neuroscience* 18.10 (2015), pp. 1353–1363. DOI: 10.1038/nn.4086.

[41] John M McNamara and Alasdair I Houston. "Integrating function and mechanism". In: *Trends in ecology & evolution* 24.12 (2009), pp. 670–675. DOI: 10.1016/j.tree.2009.05.011.

[42] John M McNamara and Alasdair I Houston. "The common currency for behavioral decisions". In: *The American Naturalist* 127.3 (1986), pp. 358–378. DOI: 10.1086/284489.

[43] Rebecca K Meagher. "Is boredom an animal welfare concern?" In: *Animal Welfare* 28.1 (2019), pp. 21–32. DOI: 10.7120/09627286.28.1.021.

[44] Michael Mendl and Elizabeth S Paul. "Animal affect and decision-making". In: *Neuroscience & Biobehavioral Reviews* 112 (2020), pp. 144–163. DOI: 10.1016/j.neubiorev.2020.01.025.

[45] Umar Faruk Mustapha et al. "Sustainable aquaculture development: A review on the roles of cloud computing, Internet of things and artificial intelligence (CIA)". In: *Reviews in Aquaculture* 13.1 (2021), pp. 2076–2091. DOI: 10.1111/raq.12559.

[46] Joacim Näslund, Mårten Rosengren, and Jörgen I Johnsson. "Fish density, but not environmental enrichment, affects the size of cerebellum in the brain of juvenile hatchery-reared Atlantic salmon". In: *Environmental Biology of Fishes* 102.5 (2019), pp. 705–712. DOI: 10.1007/s10641-019-00864-9.

[47] Suresh Neethirajan. "The use of artificial intelligence in assessing affective states in livestock". In: *Frontiers in Veterinary Science* 8 (2021), p. 282. DOI: 10.3389/fvets.2021.715261.

[48] Kathy Overton et al. "Salmon lice treatments and salmon mortality in Norwegian aquaculture: a review". In: *Reviews in Aquaculture* 11.4 (2019), pp. 1398–1417. DOI: 10.1111/raq.12299.

[49] Achim Peters, Bruce S McEwen, and Karl Friston. "Uncertainty and stress: Why it causes diseases and how it is mastered by the brain". In: *Progress in neurobiology* 156 (2017), pp. 164–188. DOI: 10.1016/j.pneurobio.2017.05.004.

[50] Jeremie M Pettersen et al. "Salmon welfare index model 2.0: an extended model for overall welfare assessment of caged Atlantic salmon, based on a review of selected welfare indicators and intended for fish health professionals". In: *Reviews in Aquaculture* 6.3 (2014), pp. 162–179. DOI: 10.1111/raq.12039.

[51] Roger A Pielke Jr. "The honest broker: making sense of science in policy and politics". In: *Cambridge University Press* (2007).

[52] Adil Rasheed, Omer San, and Trond Kvamsdal. "Digital twin: Values, challenges and enablers from a modeling perspective". In: *IEEE Access* 8 (2020), pp. 21980–22012. DOI: 10.1109/ACCESS.2020.2970143.

[53] Markus Reichstein et al. "Early warning of complex climate risk with integrated artificial intelligence". In: *Research Square* (2024). DOI: 10.21203/rs.3.rs-4248340/v1.

[54] Eve Royer and Roberto Pastres. "Data assimilation as a key step towards the implementation of an efficient management of dissolved oxygen in land-based aquaculture". In: *Aquaculture International* 31.3 (2023), pp. 1287–1301. DOI: 10.1007/s10499-022-01028-w.

[55] Kepa Ruiz-Mirazo, Juli Peretó, and Alvaro Moreno. "A universal definition of life: autonomy and open-ended evolution". In: *Origins of Life and Evolution of the Biosphere* 34.3 (2004), pp. 323–346. DOI: 10.1023/B:ORIG.0000016440.53346.dc.

[56] William Moy Stratton Russell and Rex Leonard Burch. "The principles of humane experimental technique". In: *Methuen* (1959).

[57] Anne Gro Vea Salvanes et al. "Environmental enrichment promotes neural plasticity and cognitive ability in fish". In: *Proceedings of the Royal Society B: Biological Sciences* 280.1767 (2013), p. 20131331. DOI: 10.1098/rspb.2013.1331.

[58] Bernhard Schölkopf et al. "Toward causal representation learning". In: *Proceedings of the IEEE* 109.5 (2021), pp. 612–634. DOI: 10.1109/JPROC.2021.3058954.

[59] Wolfram Schultz. "A dopamine mechanism for reward maximization". In: *Proceedings of the National Academy of Sciences* 121.4 (2024), e2316658121. DOI: 10.1073/pnas.2316658121.

[60] Helmut Segner et al. "Welfare of fishes in aquaculture". In: *FAO Fisheries and Aquaculture Circular* C1189 (2019), pp. 1–97.

[61] Anil Seth. "Being you: A new science of consciousness". In: *Penguin* (2021).

[62] Katherine A Sloman et al. "Ethical considerations in fish research". In: *Journal of fish biology* 94.4 (2019), pp. 556–577. DOI: 10.1111/jfb.13946.

[63] Berry M Spruijt, Ruud Van den Bos, and Francisca TA Pijlman. "A concept of welfare based on reward evaluating mechanisms in the brain: anticipatory behaviour as an indicator for the state of reward systems". In: *Applied Animal Behaviour Science* 72.2 (2001), pp. 145–171. DOI: 10.1016/S0168-1591(00)00204-5.

[64] Peter Sterling. "Allostasis: a model of predictive regulation". In: *Physiology & behavior* 106.1 (2012), pp. 5–15. DOI: 10.1016/j.physbeh.2011.06.004.

[65] Lars H Stien et al. "Salmon Welfare Index Model (SWIM 1.0): a semantic model for overall welfare assessment of caged Atlantic salmon: review of the selected welfare indicators and model presentation". In: *Reviews in Aquaculture* 5.1 (2013), pp. 33–57. DOI: 10.1111/j.1753-5131.2012.01083.x.

[66] Barbara Taborsky et al. “Towards an evolutionary theory of stress responses”. In: *Trends in ecology & evolution* 36.1 (2021), pp. 39–48. DOI: 10.1016/j.tree.2020.09.003.

[67] Fei Tao et al. “Digital twin modeling”. In: *Journal of Manufacturing Systems* 64 (2022), pp. 372–389. DOI: 10.1016/j.jmsy.2022.06.015.

[68] Evan Thompson. “Mind in life: Biology, phenomenology, and the sciences of mind”. In: *Harvard University Press* (2007).

[69] Erik VanderHorn and Sankaran Mahadevan. “Digital Twin: Generalization, characterization and implementation”. In: *Decision Support Systems* 145 (2021), p. 113524. DOI: 10.1016/j.dss.2021.113524.

[70] Marco A Vindas et al. “Brain serotonergic activation in growth-stunted farmed salmon: adaption versus pathology”. In: *Royal Society open science* 3.5 (2016), p. 160030. DOI: 10.1098/rsos.160030.

[71] Marco A Vindas et al. “Coping with unpredictability: dopaminergic and neurotrophic responses to omission of expected reward in Atlantic salmon (Salmo salar L.)” In: *PLoS One* 9.1 (2014), e85543. DOI: 10.1371/journal.pone.0085543.

[72] Hans van de Vis et al. “The welfare of fishes in different EU production systems”. In: *The Welfare of Fish* (2020), pp. 199–215. DOI: 10.1007/978-3-030-41675-1_9.

[73] Michael Way. ““What I cannot create, I do not understand””. In: *Journal of Cell Science* 130.18 (2017), pp. 2941–2942. DOI: 10.1242/jcs.209791.

[74] John C Wingfield. “The comparative biology of environmental stress: behavioural endocrinology and variation in ability to cope with novel, changing environments”. In: *Animal Behaviour* 85.5 (2013), pp. 1127–1133. DOI: 10.1016/j.anbehav.2013.02.018.

[75] John C Wingfield et al. “Ecological bases of hormone-behavior interactions: the "emergency life history stage"”. In: *American Zoologist* 38.1 (1998), pp. 191–206. DOI: 10.1093/icb/38.1.191.

[76] Orr Zacks, Simona Ginsburg, and Eva Jablonka. “The futures of the past. The evolution of imaginative animals”. In: *Journal of Consciousness Studies* 29.3-4 (2022), pp. 29–61. DOI: 10.53765/20512201.29.3.029.

[77] Günther KH Zupanc. “Neurogenesis and neuronal regeneration in the adult fish brain”. In: *Journal of Comparative Physiology A* 192.6 (2006), pp. 649–670. DOI: 10.1007/s00359-006-0104-y.