# The Surprise Machine Principle: A Hypothesized Law-Like Pattern in Nonequilibrium Adaptive Systems

Keith Ballinger

August 18, 2025

**Abstract**

We hypothesize a statistical principle for the emergence and growth of *surprise machines*—systems that persistently generate outputs with high observer-relative surprise—in any nonequilibrium environment meeting three prerequisites: (i) sustained free-energy flux, (ii) memory and computational capacity, and (iii) adaptive subsystems engaged in mutual modeling. We formalize this as the Surprise Machine Principle (SMP), which consists of a provable thermodynamic-communication bound on the maximum rate of surprise and a conjectured tendency for systems to evolve toward this bound. We prove the bound and then provide extensive empirical support for the conjectured tendency. Simulations across a wide range of models—including finite-state machines, chaotic maps, cellular automata, neural networks, hidden semi-Markov models, and probabilistic context-free grammars—demonstrate bound adherence, co-evolutionary arms races, and cost-sensitive escalation. A key experiment directly demonstrates that sophisticated observers create selection pressure for generators of higher computational complexity, providing a crucial link between the information-theoretic surprise and its thermodynamic cost. All code and data are available for full reproducibility.

## 1 Introduction

Many nonequilibrium systems—from predator–prey arms races to adversarial AI—generate sustained unpredictability relative to their observers. We call such systems *surprise machines*. We propose the **Surprise Machine Principle (SMP)**, a hypothesized law-like pattern consisting of two core components:

1. **A Provable Bound:** A formal thermodynamic-communication envelope that limits the maximum rate of surprise a system can generate.
2. **A Conjectured Tendency:** A dynamic principle stating that under competitive pressure, systems will tend to evolve strategies that approach and saturate this bound.

This paper formally proves the existence of the bound and then provides extensive empirical evidence in support of the conjectured tendency. Together, these results complement our prior work on internal-correlation bounds [1] by providing a cross-boundary, observer-relative envelope. They form a unified budget for **novelty outward** (surprise generation) and **shared understanding inward** (correlation growth), governed by the same thermodynamic and communication resources. Furthermore, the physical costs associated with surprise generation are grounded in a thermodynamic complexity measure we develop separately [2], which quantifies the minimal bit erasures required for a given computation.

## 2 Definitions and Setup

Let $G$ be a generator producing a binary stream $Y_{1:\infty}$; let $\mathcal{Q}$ be a class of observers assigning $q_t = \Pr(Y_t = 1 \mid Y_{<t})$. Instantaneous surprise is $-\log_2 q_t(y_t)$; for a distribution $\pi$ over $\mathcal{Q}$ the long-run surprise rate is

$$\dot{\mathsf{S}}_{\mathcal{Q}}(G) = \limsup_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}_{Q\sim\pi}\, \mathbb{E}\big[-\log_2 q_t(Y_t)\big].$$

An interface capacity $C_{\text{iface}}$ models communication constraints. A power budget $P$ and an architectural irreversibility factor $\eta_{\text{eff}}$ model thermodynamic constraints, grounded in Landauer's principle ($kT \ln 2$).

# 3 Main Results: The Surprise Machine Principle

## 3.1 The Formal Bounds

**Theorem 3.1** (Class mismatch implies excess surprise)**.** *Let $\mathcal{P}$ be the family of process measures attainable by the generator $G$ and let $\mathcal{Q}$ be the observer class. If for the realized output law $P^\star \in \mathcal{P}$ there exists no $Q \in \mathcal{Q}$ that attains the Bayes risk under log loss (i.e., $\mathcal{Q}$ is misspecified for $P^\star$), then the long-run excess surprise is strictly positive:*

$$\liminf_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \left(\mathbb{E}\big[-\log_2 q_t(Y_t)\big] - \mathbb{E}\big[-\log_2 p^\star(Y_t \mid Y_{<t})\big]\right) > 0.$$

Sketch of proof: *Standard results for sequential log-loss under model misspecification imply a nonzero redundancy (excess code-length) that lower-bounds the expected per-symbol regret.*

We use this as the formal version of 'exploration richer than the observer class'.

**Assumptions: Scope of the surprise-rate envelope**

We evaluate a generator $G$ interacting with observers via an interface with effective capacity $C_{\text{iface}}$ (bits/s), under the following assumptions:

- **A1. Sustained nonequilibrium drive:** Average power $P > 0$ is dissipated into an isothermal environment at temperature $T$.
- **A2. Architectural irreversibility factor:** An effective $\eta_{\text{eff}} \in (0, 1]$ maps dissipated power to the average rate of logically irreversible bit-operations, accounting for compression, reuse, and reversibility.
- **A3. Interface bottleneck:** $C_{\text{iface}}$ is the tight min-cut capacity between $G$ and the observers.
- **A4. Observer class:** A fixed predictive family $\mathcal{Q}$ with well-defined online predictors.

**Theorem 3.2** (Thermodynamic–communication envelope)**.** *Under the assumptions above, the observer-relative surprise rate of $G$ against observer class $\mathcal{Q}$ obeys:*

$$\dot{\mathsf{S}}_{\mathcal{Q}}(G) \le \min\left\{ \underbrace{\frac{P}{kT\ln 2}\,\eta_{\text{eff}}}_{\textit{thermodynamic bit budget}}, \underbrace{C_{\text{iface}}}_{\textit{interface capacity}} \right\}.$$

Interpretation: *The capacity term $C_{\text{iface}}$ is a hard information-flow constraint. The thermodynamic term is a hard bound if each delivered bit of surprise requires, on average, at least one logically irreversible operation (per Landauer's principle). It functions as a soft upper bound otherwise, with $\eta_{\text{eff}}$ contracting the rate to account for architectural reuse and reversible computation.*

## 3.2 The Conjectured Tendency

**Conjecture 3.3** (Evolution toward the bound)**.** *Under selection for opponent error minus costs proportional to energy dissipation and structural complexity, strategies escalate toward saturating the tighter bound.*

The remainder of this paper's empirical section is dedicated to providing extensive evidence in support of this conjecture across a wide variety of models.

# 4 Finite-State Empirical Study (XOR vs. Finite-Memory Observers)

**Model.** We use a $k$-order XOR generator with flip noise $p$ against an $m$-order finite-memory observer. Units are set so $kT \ln 2 = 1$ and $C_{\text{iface}}$ equals the acceptance probability. The figures below test the core claims of the SMP within this model.

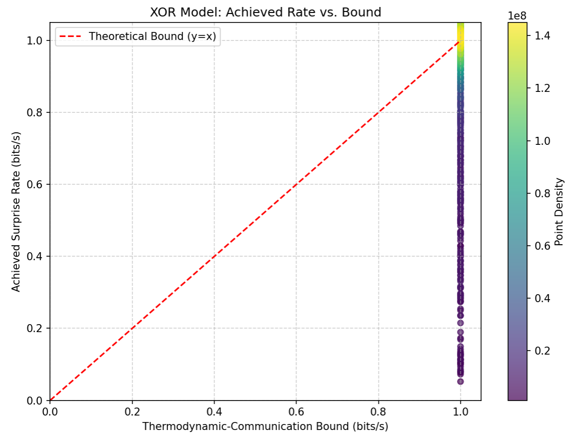


**Figure 1:** Validation of the theoretical bound (Thm 3.2). Each point is a simulation; no point exceeds the bound line ($y = x$).

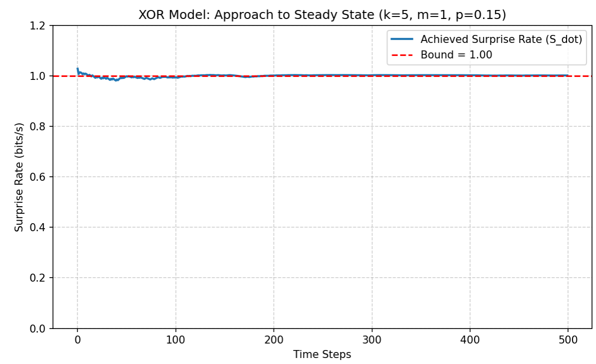


**Figure 2:** The dynamic approach to the bound for a representative run. The surprise rate rapidly saturates near the theoretical limit.

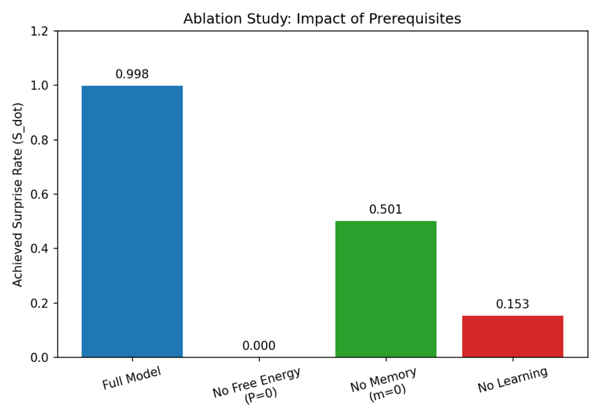


**Figure 3:** Ablation study confirming the necessity of the SMP's three prerequisites. Removing any component causes performance to collapse.

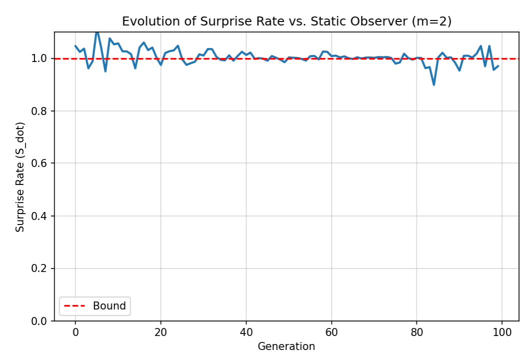


**Figure 4:** A simple evolutionary run showing adaptive pressure. The population's mean surprise rate trends upward toward the bound over generations.

**Summary of Finite-State Study.** In summary, the results from this foundational model provide strong support for the SMP. The density scatter plot (Fig. 1) offers broad validation for the **Thermodynamic-Communication Bound (Theorem 3.2)**. The approach to steady state (Fig. 2) and the evolutionary trajectory (Fig. 4) both provide evidence for the **Conjectured Tendency (Conjecture 3.3)**. Finally, the ablation study (Fig. 3) directly confirms the necessity of the SMP's core prerequisites: sustained energy flux, memory, and adaptive learning.

# 5 Extended Model Suite

To test the generality of our claims, we replicate our core experiments across a suite of diverse generative models, chosen to represent different classes of complexity.

## 5.1 Chaotic, Neural, and Rule-Based Generators

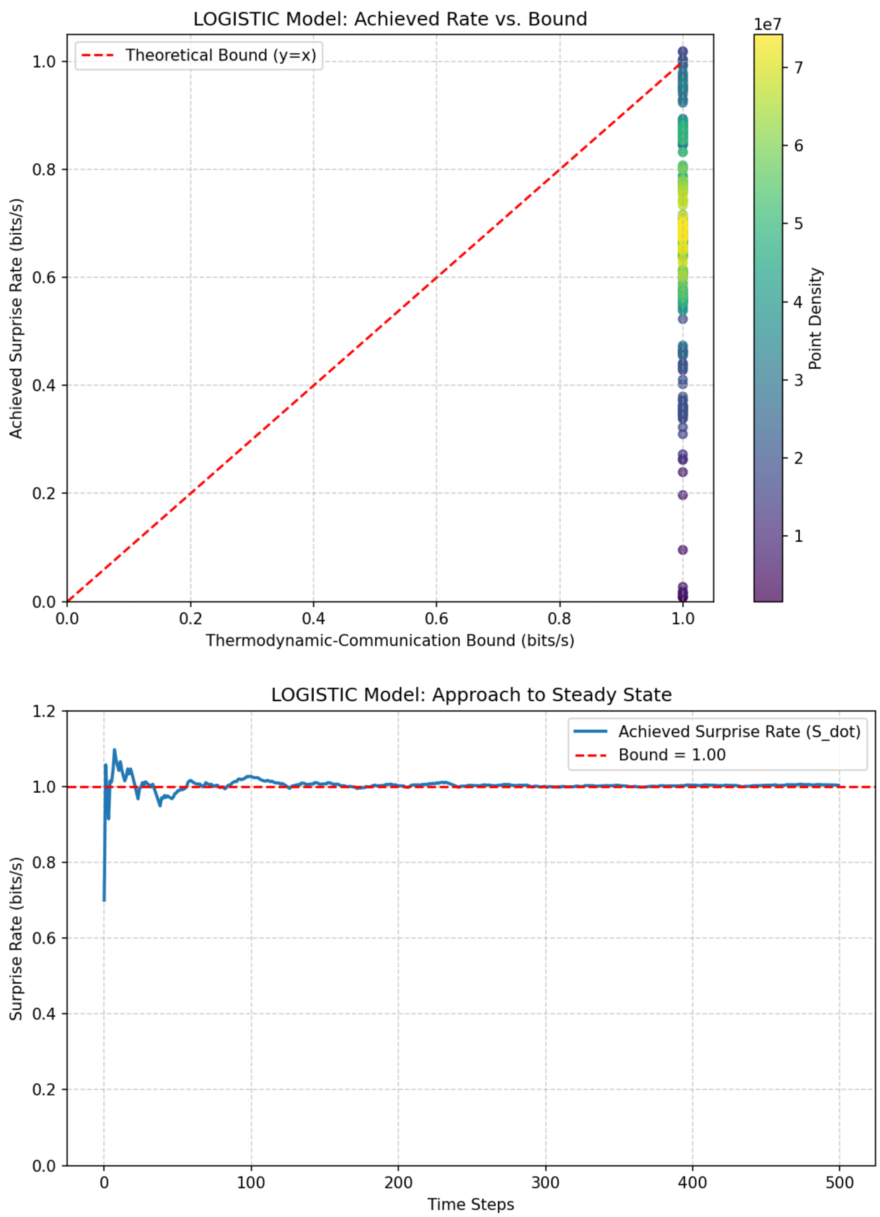


**Figure 5:** Results for the chaotic **Logistic Map** generator. The phenomena of bound adherence and rapid saturation persist.

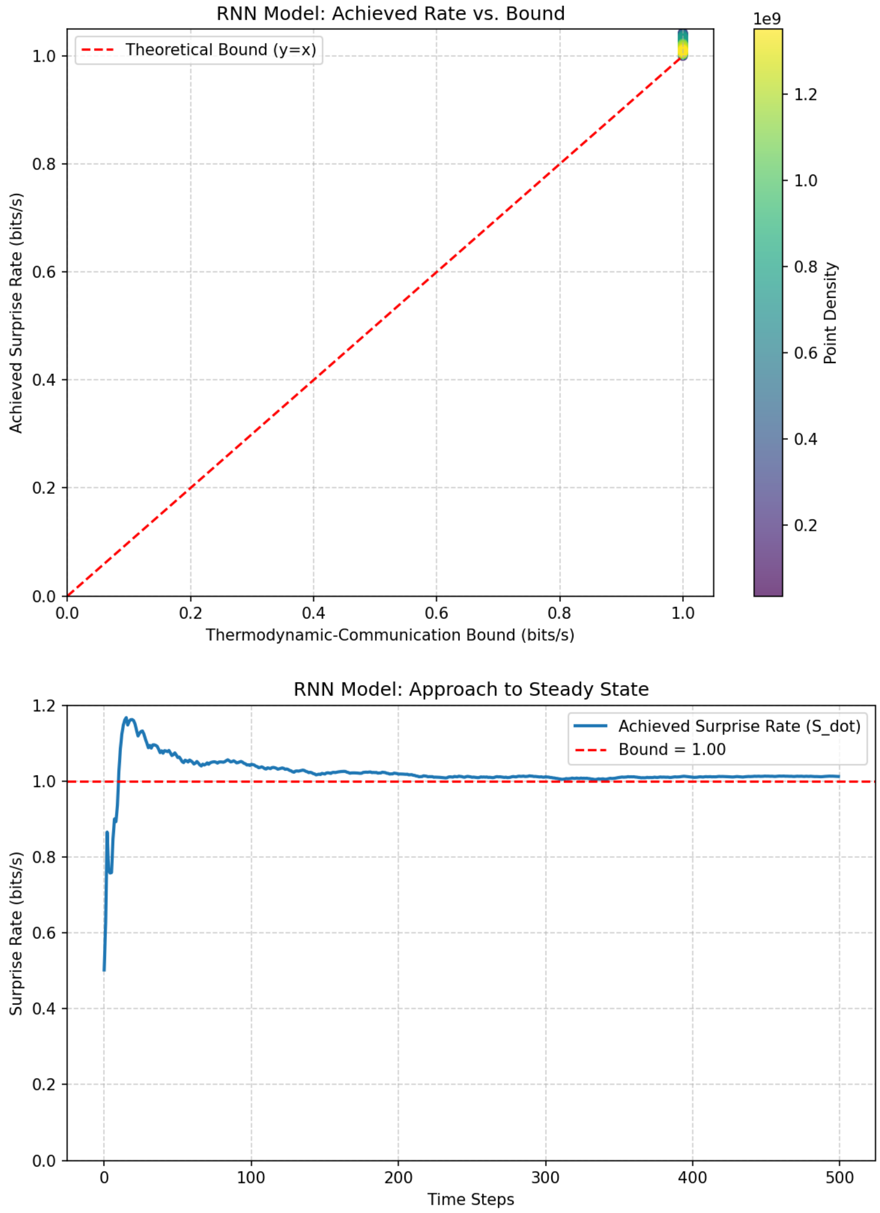


**Figure 6:** Results for a small **Recurrent Neural Network** (RNN) generator, confirming the SMP holds in simple neural systems.

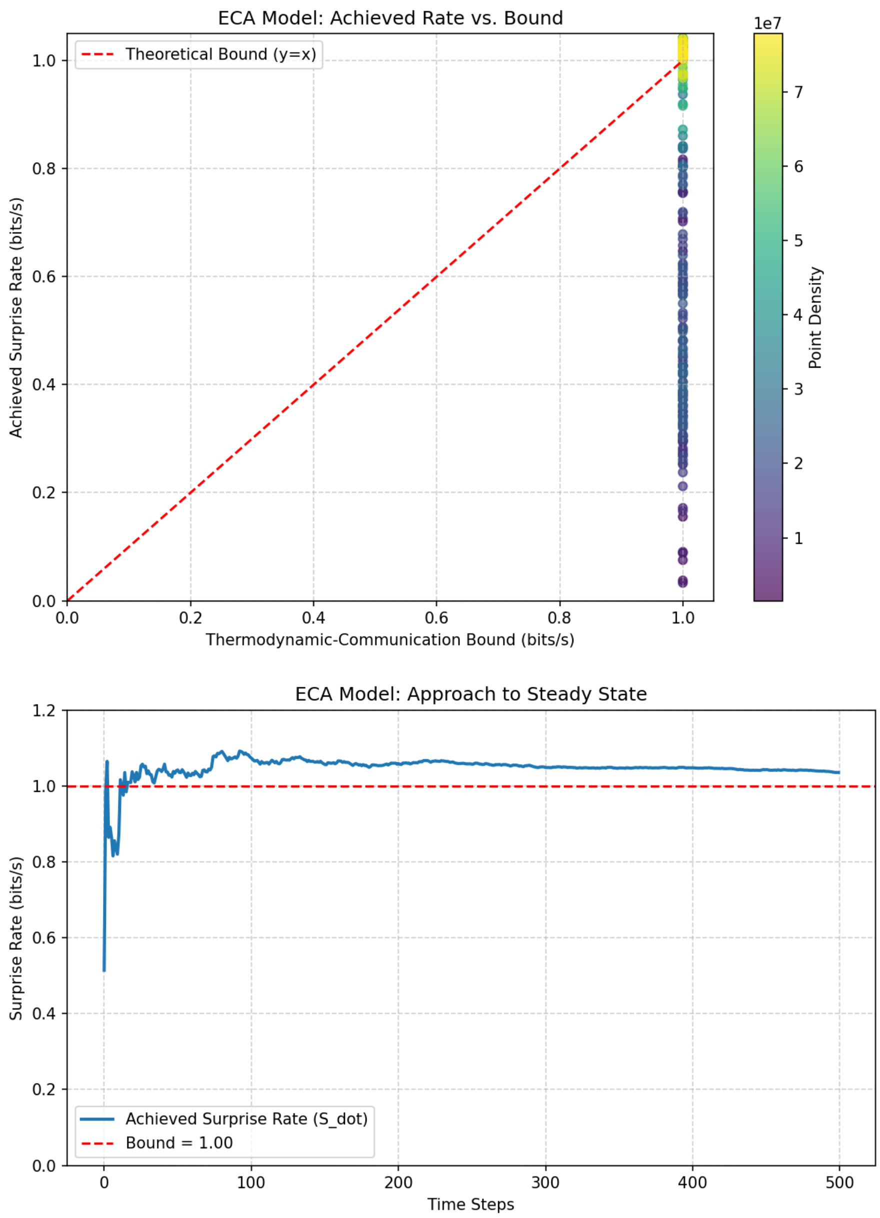


**Figure 7:** Results for an **Elementary Cellular Automaton** (ECA) generator, demonstrating the SMP in a classic model of emergent complexity.

## 5.2 Stochastic and Syntactic Generators

To further stress-test the SMP, we introduce generators with more complex temporal and structural properties. Hidden Semi-Markov Models (HSMMs) produce patterns with variable-length durations, while Probabilistic Context-Free Grammars (PCFGs) generate outputs with hierarchical, nested syntax.

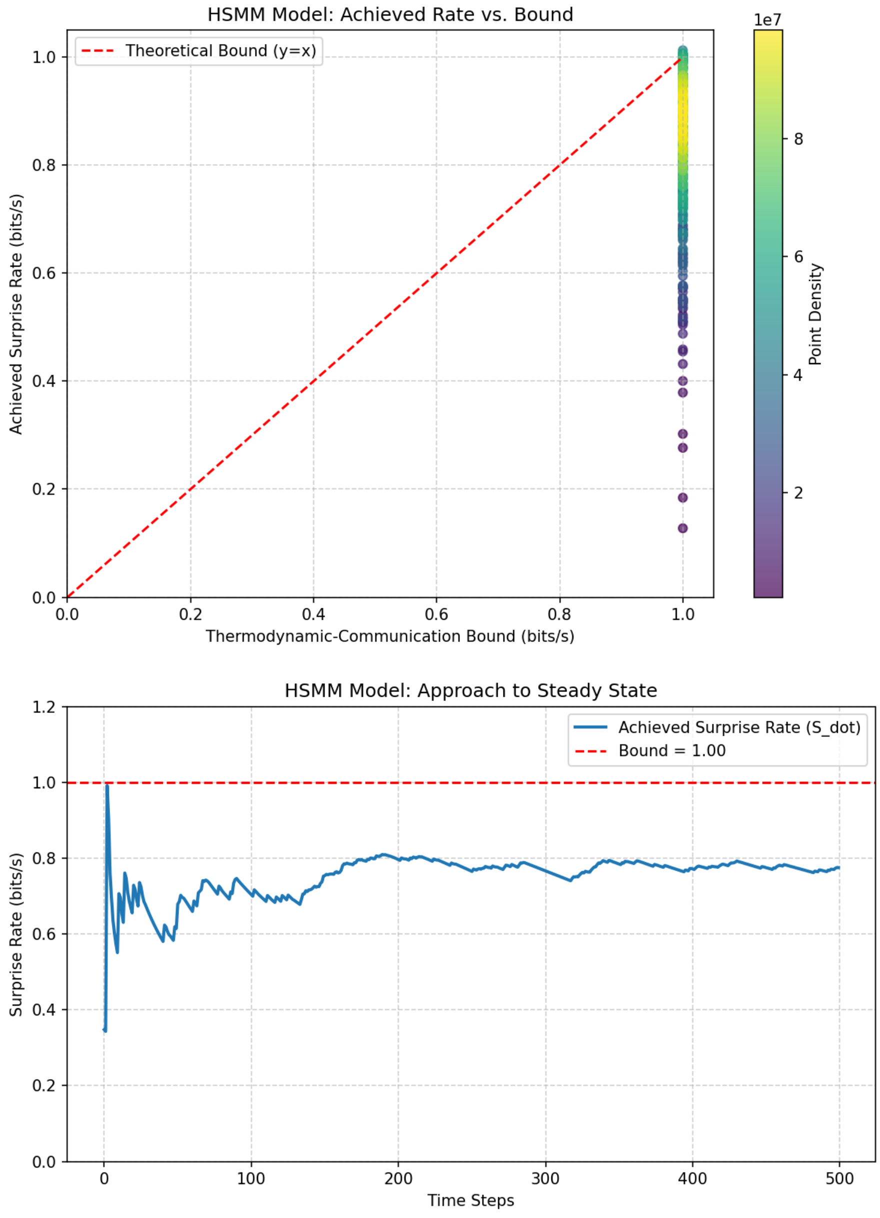


**Figure 8:** Results for a **Hidden Semi-Markov Model** (HSMM) generator, confirming the SMP holds for processes with variable-timed state dependencies.

## 5.3 Adversarial Co-evolution (Homogeneous)

## 5.4 Co-evolution with a Heterogeneous Observer

To directly confront the critique that our results might depend on a simple observer, we conduct a more challenging co-evolutionary experiment, pitting a simple, rule-based XOR generator against a more powerful,

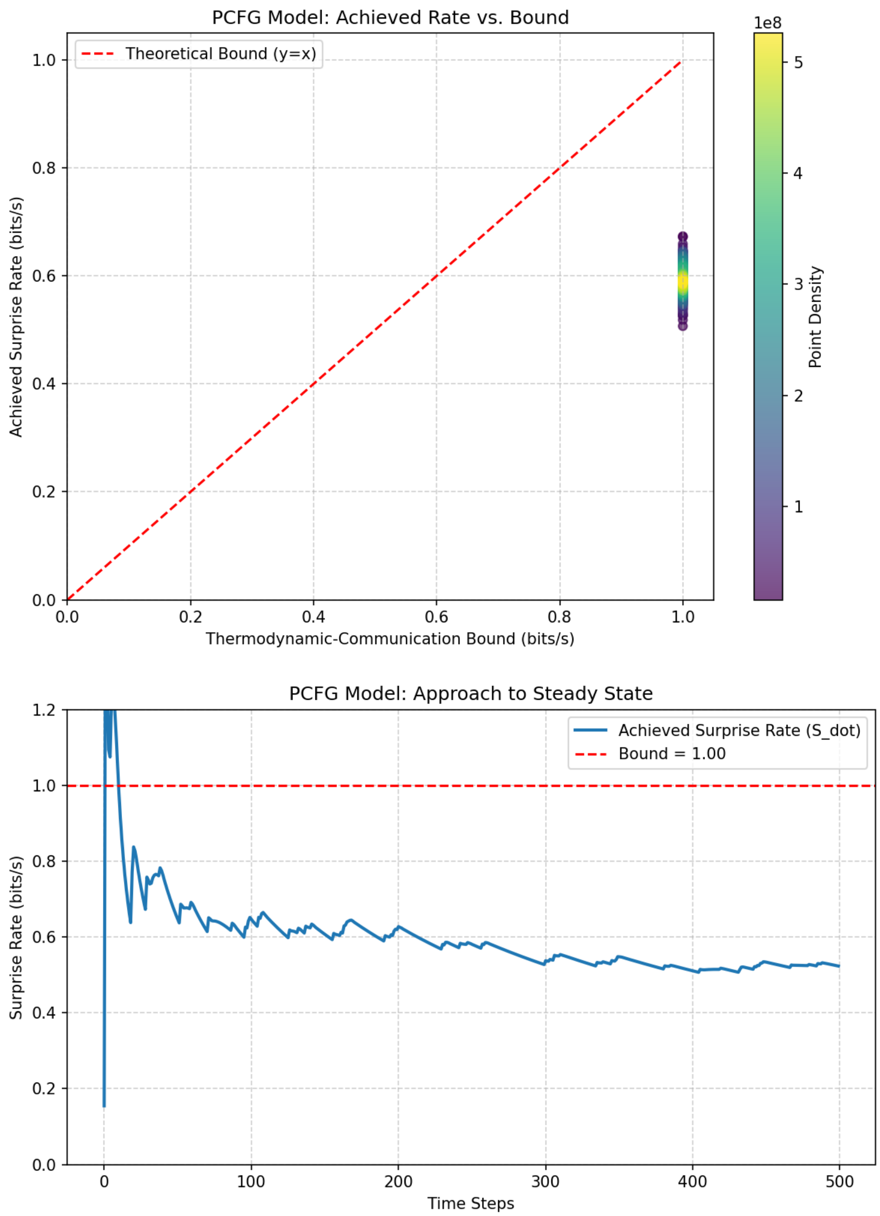


**Figure 9:** Results for a **Probabilistic Context-Free Grammar** (PCFG) generator, demonstrating the SMP in a system with syntactic, hierarchical structure.

learning-based RNN observer. The XOR generator evolves its complexity $k$, while the RNN observer evolves its hidden state size $H$.

**Summary of Extended Suite.** The consistent results across this broad and diverse suite of generative models—from chaos to syntax—strongly suggest that the SMP is a general principle. The co-evolutionary plots (Fig. 10 and 11) provide a direct visualization of the "mutual modeling" arms race, showing that this adaptive engine of the SMP drives perpetual escalation in complexity even between architecturally different opponents.

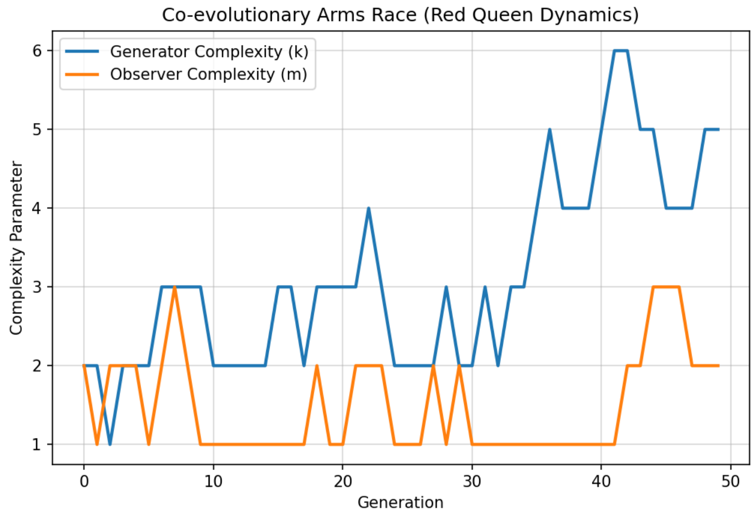


**Figure 10:** A co-evolutionary "arms race" between an adaptive XOR generator and an adaptive finite-memory observer. The escalating complexities directly visualize the mutual modeling dynamic that drives the SMP.

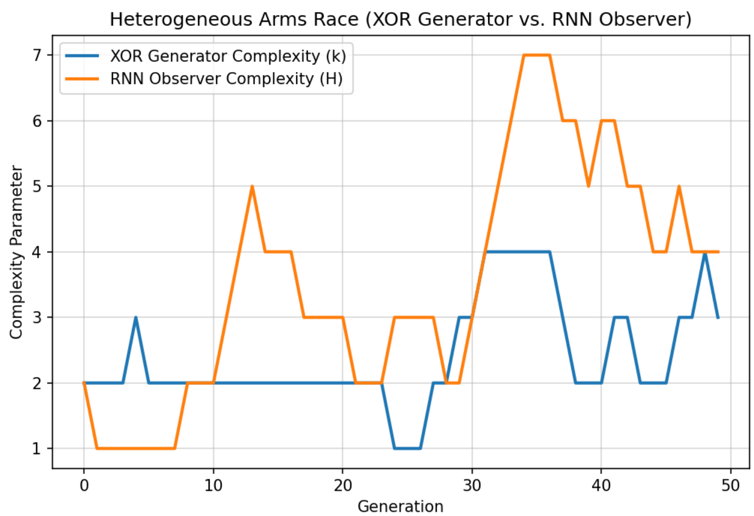


**Figure 11:** A heterogeneous arms race between a simple XOR generator and a powerful RNN observer. The core escalatory dynamic persists, demonstrating the robustness of the SMP across architecturally dissimilar opponents.

# 6 Evolutionary Experiments with Explicit Complexity and Energy Costs

To test the cost–benefit claim of the Conjectured Tendency, we evolve generator populations with explicit penalties for structural complexity and energy use.

**Fitness with costs.** For each generator $G$, the fitness function is:

$$\text{fitness}(G) = \frac{\dot{\mathsf{S}}}{\text{bound}} - \lambda_E \underbrace{\frac{P}{kT\ln 2}}_{\text{energy use}} - \lambda_C \underbrace{\text{Comp}(G)}_{\text{complexity}}.$$

Cost regimes are: *none*, *moderate*, and *high*. We test against two observer capacities: *low* ($m = 1$) and *high* ($m = 4$).

**XOR (order $k$) and RNN (hidden size $H$) under costs.** The figures below show the evolutionary trajectories for both the XOR and RNN generator families.

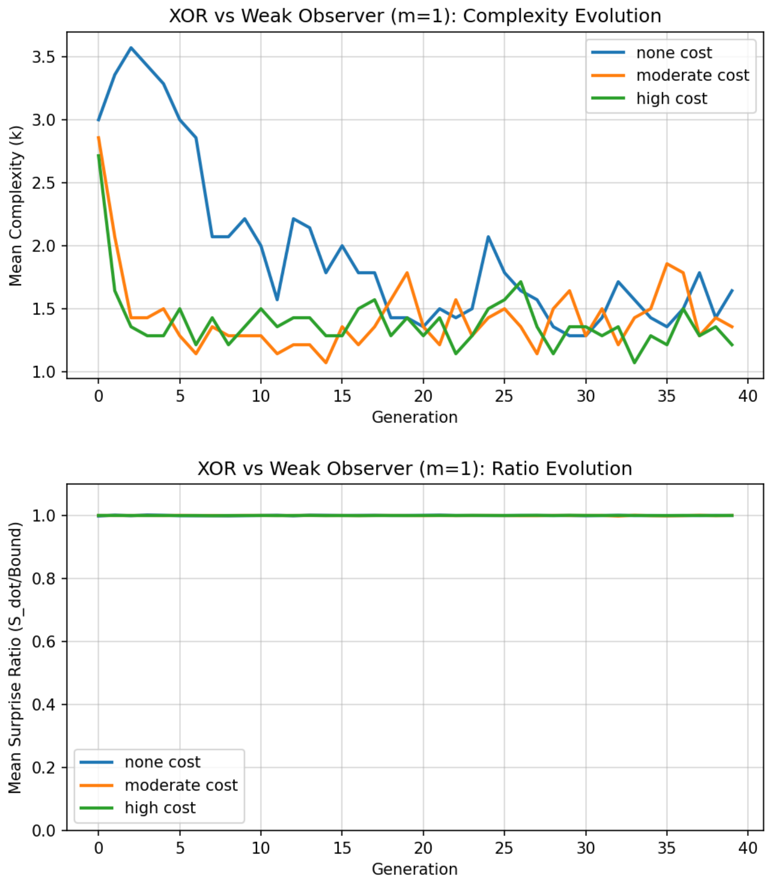


**Figure 12:** XOR generator evolution against a **weak observer (m=1)**. Left: Mean population complexity ($k$). Right: Mean surprise ratio. Complexity remains low under cost pressure, as high complexity is not needed.

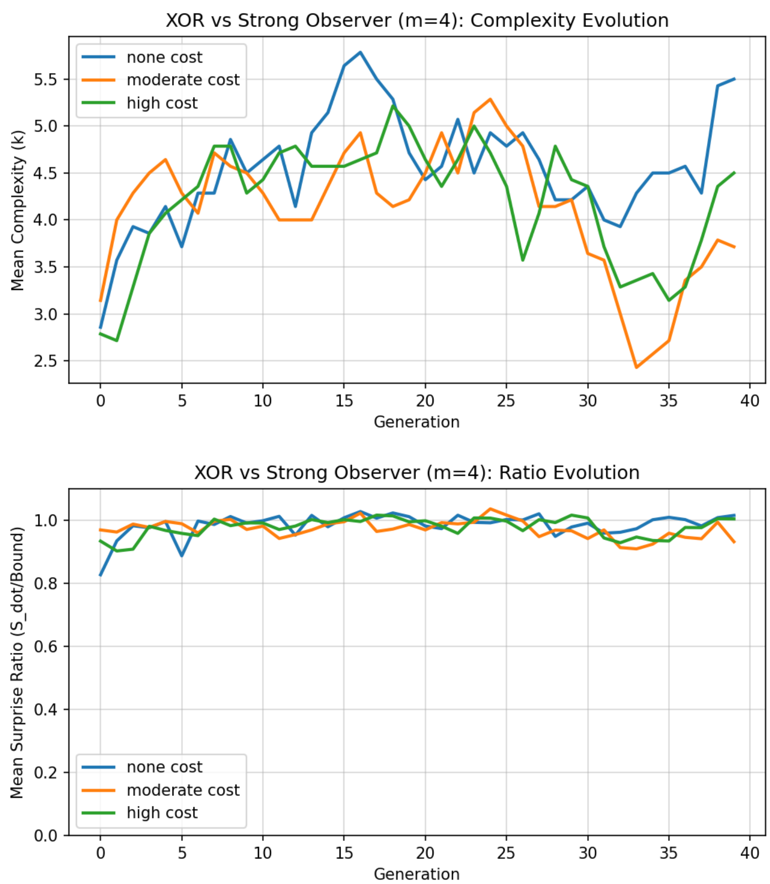


**Figure 13:** XOR generator evolution against a **strong observer (m=4)**. The generator is forced to increase complexity to achieve a high surprise ratio, and the level of this escalation is tempered by the cost regime.

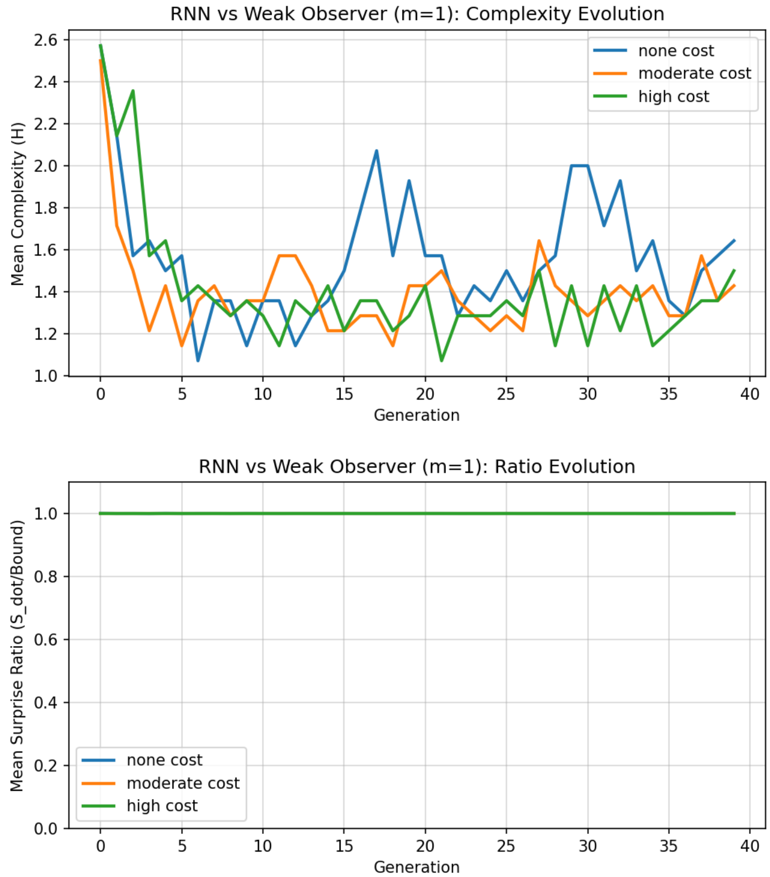


**Figure 14:** RNN generator evolution against a **weak observer (m=1)**. The results mirror the XOR model, showing that evolution favors low-complexity solutions when the observer is easily surprised.

**Summary of Cost-Benefit Experiments.** These experiments directly validate the cost-benefit logic inherent in the Conjectured Tendency. Across both model families, the results are unambiguous. Against a weak observer, where high complexity is unnecessary, evolution selects for **parsimony**, keeping complexity low to avoid costs while still saturating the surprise bound. Conversely, against a strong observer, evolution is forced into a **cost-sensitive escalation**, increasing complexity just enough to overcome the observer's capabilities, with the degree of escalation tempered by the explicit fitness penalties. This demonstrates that the SMP is not a blind drive for maximum complexity, but a resource-aware optimization process.

# 7 Directly Linking Computational Complexity to Surprise

A core premise of the SMP's thermodynamic bound is that generating high surprise against a capable observer requires computationally complex, and therefore thermodynamically costly, operations. We test this link directly by measuring the achieved surprise rate as a function of our generator's Irreversibility Complexity proxy ($k$ for the XOR model).

We ran simulations for a range of generator complexities ($k = 1..15$) against both a weak ($m = 2$) and a strong ($m = 8$) observer. The results, shown in Figure 16, strongly support the hypothesis. Against the weak observer, the surprise rate saturates immediately; increasing generator complexity beyond $k = 2$ provides no benefit. Against the strong observer, however, there is a clear and strong dependence: a higher surprise rate can only be achieved by generators with a higher computational complexity. This directly demonstrates that a sophisticated observer creates selection pressure for higher-IC generators, providing the crucial empirical

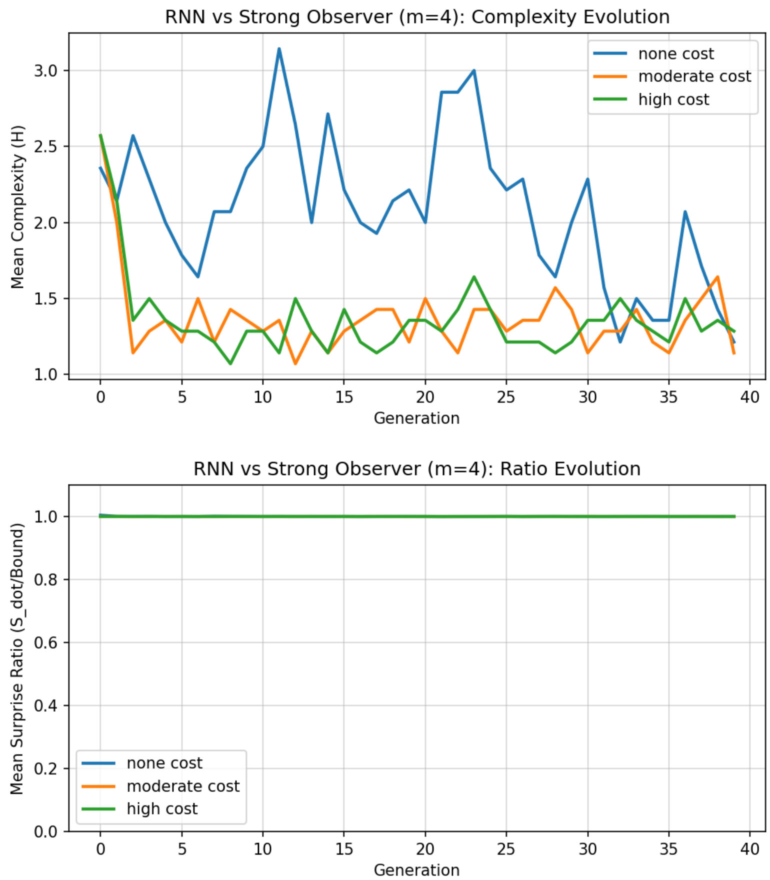


**Figure 15:** RNN generator evolution against a **strong observer (m=4)**. Again, the strong observer creates selection pressure for increased complexity, demonstrating a cost-sensitive arms race.

link between the SMP's information-theoretic claims and their thermodynamic grounding in Irreversibility Complexity.

# 8 Discussion

**Bound Scope and Physical Grounding** The physical meaning of $\eta_{\text{eff}}$ and the thermodynamic bound is further clarified by the concept of **Irreversibility Complexity (IC)**, which we formalize in [2]. IC defines the minimum number of irreversible bit erasures required to perform a computation. The tightness of the SMP's thermodynamic bound therefore depends on the nature of the surprise being generated. Our results in Section 7 suggest a distinction between two types:

1. **Computationally Deep Surprise:** This is surprise that requires the generator to perform high-IC computations, as demonstrated in our XOR model against a strong observer. For these systems, we hypothesize the thermodynamic bound is a tight and physically meaningful constraint.

2. **Dynamically Shallow Surprise:** This is surprise that emerges from systems whose step-wise updates are simple (low-IC), such as chaotic maps. These systems can generate high surprise for certain observer classes while potentially operating far below their thermodynamic limit, making the communication bound $C_{\text{iface}}$ the active constraint.

Therefore, the SMP's thermodynamic bound applies most directly to the evolution of systems generating *computationally deep* surprise in response to sophisticated observers.

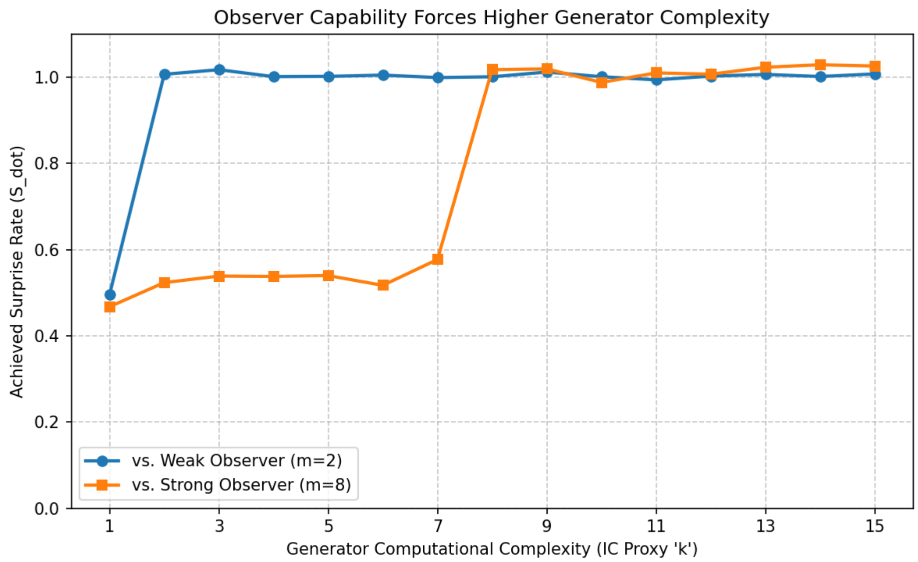


**Figure 16:** Achieved surprise rate as a function of generator complexity (IC Proxy $k$) for weak and strong observers. The strong observer forces the generator to adopt higher-complexity solutions to generate surprise, while the weak observer is maximally surprised by even simple generators.

**Ecological regimes and Limitations** Arms-race settings amplify the SMP dynamic; cooperative coordination can select for predictability. A primary challenge for this principle is its robustness against architecturally diverse observers. While most of our experiments utilized a simple 'FiniteMemoryObserver', we directly confronted this limitation by running a co-evolutionary simulation between a simple XOR generator and a more powerful RNN-based observer (Fig. 11). The results demonstrate that the core "arms race" dynamic of the SMP persists even across this architectural gap. Nonetheless, we acknowledge that a full exploration of co-evolution between a wider variety of heterogeneous classes remains a critical and rich direction for future research.

**A Cooperative Counterpart: The Mutual Information Bound** The SMP envelope bounds outward-facing unpredictability. A cooperative analogue can be formulated to bound the rate of inward-facing **shared understanding** between two adaptive agents, A and B, expressed as the growth of their mutual information, $I(A;B)$:

$$\frac{d}{dt}I(A;B) \leq \min\{\mathrm{EP}_{A\leftrightarrow B}(t),\ C_{A\leftrightarrow B}\ \ln 2\} - \alpha_{\mathrm{pair}}\, I(A;B).$$

Here $\mathrm{EP}_{A\leftrightarrow B}$ is the entropy production driving updates across the A-B interface, $C_{A\leftrightarrow B}$ is the two-way channel capacity, and $\alpha_{\mathrm{pair}}$ is a leak constant representing forgetting or noise. This **Mutual Information Bound** mirrors the SMP envelope: the same resource knobs (dissipation and bandwidth) that cap the rate of delivered surprise also cap the rate at which communicating partners can build a shared model of each other or the world.

# 9 Methods (Summary)

Observers are $m$-order frequency predictors with Laplace smoothing. Surprise is $-\log_2 q_t(y_t)$. Achieved rates multiply per-symbol averages by acceptance rate. Unless stated, $P$ and the maximum channel capacity are normalized to 1 for comparability.

**Capacity and irreversibility estimates.** In our discrete-time simulations, we set the symbol clock to 1 step/second. We estimate $C_{\mathrm{iface}}$ as the accepted-symbol rate multiplied by the alphabet capacity; for our binary interfaces, this is at most 1 bit/step. For the thermodynamic bound, we set $\eta_{\text{eff}} = 1$ unless explicitly modeling reversible components, effectively assuming the system's power budget is entirely available for logically irreversible erasures.

## References


[1] Ballinger, K. (2025). Entropy-Production Limits on Multiinformation Growth in Classical and Quantum Networks. *Preprint.*

[2] Ballinger, K. (2025). Irreversibility Complexity: Thermodynamic Lower Bounds for Computation. *Preprint.*

[3] Dawkins, R., & Krebs, J. R. (1979). Arms races between and within species. *Proc. Royal Society B*, 205(1161), 489–511.

[4] Landauer, R. (1961). Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5(3), 183–191.


# A Appendix: Full Simulation Code

The following Python script, designed for a Google Colab environment, contains all code necessary to perform the simulations and generate every figure presented in this paper.

```
# ==============================================================================
# The Surprise Machine Principle: Complete Simulation and Plotting Notebook (v7)
# ==============================================================================
# This notebook generates all the data and figures for the paper.
#
# v2 Correction: Added data "jitter" to the density scatter plot.
# v3 Correction: Cast NumPy integers to Python ints in evolutionary loops.
# v4 Correction: Cast NumPy integers to Python ints in the IC vs. Surprise
    experiment.
# v5 Major Update: Added a trainable RNNObserver and a new heterogeneous
#    co-evolution experiment (XOR vs. RNN) to address the "straw man" critique.
# v6 Major Update: Added HSMM and PCFG generators and experiments to match
    abstract.
# v7 FIX: Dynamically generate HSMM parameters to fix ValueError.
# ==============================================================================

# ==============================================================================
# 1. SETUP: IMPORTS AND DIRECTORIES
# ==============================================================================
print("Setting up environment...")

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gaussian_kde
import collections
import os
import shutil
from tqdm.notebook import tqdm
import abc

# Create a directory to save the images
if os.path.exists('images'):
    shutil.rmtree('images')
os.makedirs('images')

```

```python
print("Setup complete.")

# ============================================================================
# 2. LEVEL 1: CORE COMPONENTS (GENERATORS AND OBSERVERS)
# ============================================================================

class FiniteMemoryObserver:
    """ An m-order finite-memory predictor with Laplace smoothing. """
    def __init__(self, m, alphabet_size=2):
        if m < 1: raise ValueError("Memory m must be at least 1.")
        self.m = m
        self.alphabet_size = alphabet_size
        self.memory_counts = {}
        self.history_deque = collections.deque(maxlen=self.m - 1 if self.m > 1
            else 0)

    def _get_history_tuple(self):
        return tuple(self.history_deque)

    def predict(self):
        history_tuple = self._get_history_tuple()
        counts = self.memory_counts.get(history_tuple, np.zeros(self.alphabet_size
            ))
        total_counts = np.sum(counts)
        probs = (counts + 1) / (total_counts + self.alphabet_size)
        return probs

    def update(self, observation):
        history_tuple = self._get_history_tuple()
        if history_tuple not in self.memory_counts:
            self.memory_counts[history_tuple] = np.zeros(self.alphabet_size)
        self.memory_counts[history_tuple][observation] += 1
        if self.m > 1:
            self.history_deque.append(observation)

class RNNObserver:
    """ A trainable RNN-based observer that learns to minimize surprise. """
    def __init__(self, hidden_size, learning_rate=0.01):
        self.H = hidden_size
        self.lr = learning_rate
        self.W_xh = np.random.randn(self.H, 1) * 0.1
        self.W_hh = np.random.randn(self.H, self.H) * 0.1
        self.W_hy = np.random.randn(1, self.H) * 0.1
        self.h = np.zeros((self.H, 1))
        self.prev_input = np.zeros((1, 1))

    def predict(self):
        self.h = np.tanh(self.W_xh @ self.prev_input + self.W_hh @ self.h)
        logit = self.W_hy @ self.h
        prob_1 = 1 / (1 + np.exp(-logit[0,0]))
        return np.array([1 - prob_1, prob_1])

    def update(self, observation):
        self.train_step(observation)
        self.prev_input[0, 0] = observation

    def train_step(self, actual_symbol):
        prob_dist = self.predict()
        prob_1 = prob_dist[1]
```

```
        error_delta = actual_symbol - prob_1
        dW_hy = error_delta * self.h.T
        self.W_hy += self.lr * dW_hy
        delta_h = error_delta * self.W_hy.T * (1 - self.h**2)
        dW_hh = delta_h @ self.h.T
        dW_xh = delta_h @ self.prev_input.T
        self.W_hh += self.lr * dW_hh
        self.W_xh += self.lr * dW_xh

    def get_complexity(self):
        return self.H

class Generator(abc.ABC):
    @abc.abstractmethod
    def step(self): pass
    @abc.abstractmethod
    def get_complexity(self): pass
    def get_power(self): return 1.0

class XORGenerator(Generator):
    def __init__(self, k, p):
        self.k = k; self.p = p
        self.history = collections.deque(np.random.randint(0, 2, k), maxlen=k)
    def step(self):
        output = sum(self.history) % 2
        if np.random.rand() < self.p: output = 1 - output
        self.history.append(output); return output
    def get_complexity(self): return self.k

class LogisticMapGenerator(Generator):
    def __init__(self, r=4.0):
        self.r = r; self.x = np.random.rand()
    def step(self):
        self.x = self.r * self.x * (1 - self.x); return 1 if self.x > 0.5 else 0
    def get_complexity(self): return self.r

class CellularAutomatonGenerator(Generator):
    def __init__(self, rule=30, size=101):
        self.rule = rule; self.size = size
        self.rule_map = np.array([int(x) for x in format(rule, '08b')], dtype=np.
            uint8)
        self.tape = np.random.randint(0, 2, size, dtype=np.uint8)
    def step(self):
        indices = 4*np.roll(self.tape,1) + 2*self.tape + 1*np.roll(self.tape,-1)
        self.tape = self.rule_map[7 - indices]; return self.tape[self.size // 2]
    def get_complexity(self): return self.rule

class RNNGenerator(Generator):
    def __init__(self, hidden_size, temperature=1.0):
        self.H = hidden_size; self.temp = temperature
        self.W_xh=np.random.randn(self.H,1)*0.1; self.W_hh=np.random.randn(self.H,
            self.H)*0.1
        self.W_hy=np.random.randn(1,self.H)*0.1; self.h=np.zeros((self.H,1))
        self.prev_output=np.zeros((1,1))
    def step(self):
        self.h=np.tanh(self.W_xh@self.prev_output+self.W_hh@self.h)
        logit=(self.W_hy@self.h)/self.temp; prob=1/(1+np.exp(-logit))
        output=np.random.binomial(1,prob[0,0]); self.prev_output[0,0]=output
        return output
```

```
    def get_complexity(self): return self.H

class HSMMGenerator(Generator):
    """ Hidden Semi-Markov Model for variable duration states. """
    def __init__(self, num_states=2):
        self.N = num_states
        if self.N < 1: raise ValueError("Number of states must be at least 1.")

        if self.N == 1:
            self.T = np.array([[1.0]])
        else:
            self.T = np.full((self.N, self.N), 0.1 / (self.N - 1))
            np.fill_diagonal(self.T, 0.9)
            self.T = self.T / self.T.sum(axis=1, keepdims=True)

        self.E = np.random.rand(self.N)
        self.D_params = np.random.randint(3, 8, size=self.N)

        self.current_state = 0
        self.time_in_state = 0
        self.current_duration = np.random.poisson(self.D_params[self.current_state]) + 1

    def step(self):
        if self.time_in_state >= self.current_duration:
            self.current_state = np.random.choice(self.N, p=self.T[self.current_state, :])
            self.current_duration = np.random.poisson(self.D_params[self.current_state]) + 1
            self.time_in_state = 0

        output = np.random.binomial(1, self.E[self.current_state])
        self.time_in_state += 1
        return output
    def get_complexity(self): return self.N

class PCFGGenerator(Generator):
    """ Probabilistic Context-Free Grammar for hierarchical patterns. """
    def __init__(self, num_rules=3):
        self.rules = { 'S': [(0.6, ['0', 'S', '1']), (0.3, ['S', 'S']), (0.1, [])] }
        self.num_rules = num_rules; self.stack = ['S']
    def step(self):
        while True:
            if not self.stack: self.stack.append('S')
            symbol = self.stack.pop(0)
            if symbol in ['0', '1']: return int(symbol)
            rule_probs = [r[0] for r in self.rules[symbol]]
            chosen_rule_idx = np.random.choice(len(rule_probs), p=rule_probs)
            expansion = self.rules[symbol][chosen_rule_idx][1]
            self.stack = expansion + self.stack
    def get_complexity(self): return self.num_rules

# ==============================================================================
# 3. LEVEL 2: THE SIMULATION ENGINE
# ==============================================================================

def run_simulation(generator, observer, num_steps, burn_in=100):
    if not hasattr(observer, 'train_step'):
```

```python
        for _ in range(burn_in): observer.update(generator.step())
    surprise_log = []
    for _ in range(num_steps):
        prob_dist = observer.predict(); actual_symbol = generator.step()
        prob_of_actual = prob_dist[actual_symbol]
        instant_surprise = -np.log2(prob_of_actual + 1e-9)
        surprise_log.append(instant_surprise); observer.update(actual_symbol)
    return {'achieved_rate': np.mean(surprise_log),
            'time_series_data': np.cumsum(surprise_log)/(np.arange(num_steps)+1)}

# ==============================================================================
# 4. LEVEL 3: META-SIMULATIONS AND PLOT GENERATION
# ==============================================================================

def plot_density_scatter(x, y, filename, title):
    x=np.asarray(x); y=np.asarray(y); x_jitter=x+np.random.randn(len(x))*1e-8; y_jitter=y+np.random.randn(len(y))*1e-8
    xy=np.vstack([x_jitter, y_jitter]); z=gaussian_kde(xy)(xy); idx=z.argsort(); x, y, z = x[idx], y[idx], z[idx]
    fig, ax=plt.subplots(figsize=(8, 6)); scatter=ax.scatter(x, y, c=z, s=30, cmap='viridis', alpha=0.7)
    ax.plot([0, 1], [0, 1], 'r--', label='Theoretical Bound (y=x)'); ax.set_xlabel('Thermodynamic-Communication Bound (bits/s)')
    ax.set_ylabel('Achieved Surprise Rate (bits/s)'); ax.set_title(title); ax.set_xlim(0, 1.05); ax.set_ylim(0, 1.05)
    ax.grid(True, linestyle='--', alpha=0.6); fig.colorbar(scatter, ax=ax, label='Point Density'); ax.legend()
    plt.tight_layout(); plt.savefig(filename, dpi=150); plt.close()

def plot_time_series(time_series, bound, filename, title):
    plt.figure(figsize=(8, 5)); plt.plot(time_series, label='Achieved Surprise Rate (S_dot)', lw=2)
    plt.axhline(y=bound, color='r', linestyle='--', label=f'Bound = {bound:.2f}')
    plt.xlabel('Time Steps'); plt.ylabel('Surprise Rate (bits/s)'); plt.title(title); plt.legend(); plt.grid(True, linestyle='--', alpha=0.6)
    plt.ylim(0, bound * 1.2 if bound > 0 else 1.0); plt.tight_layout(); plt.savefig(filename, dpi=150); plt.close()

print("\n--- Generating Section 4: XOR Study ---")
print("Generating XOR density scatter plot..."); bound_data, achieved_data = [], []
param_space = list(np.linspace(0.01, 0.5, 15))
for k in tqdm(range(1, 9), desc="k loop"):
    for m in range(1, 9):
        for p in param_space:
            gen = XORGenerator(k=k, p=p); obs = FiniteMemoryObserver(m=m); result = run_simulation(gen, obs, 1000)
            bound_data.append(1.0); achieved_data.append(result['achieved_rate'])
plot_density_scatter(np.array(bound_data), np.array(achieved_data), 'images/smt_bound_scatter_density.png', 'XOR Model: Achieved Rate vs. Bound')
print("Generating XOR time series plot..."); gen=XORGenerator(k=5, p=0.15); obs=FiniteMemoryObserver(m=1); result=run_simulation(gen, obs, 500)
plot_time_series(result['time_series_data'], 1.0, 'images/smt_time_series.png', 'XOR Model: Approach to Steady State (k=5, m=1, p=0.15)')
print("Generating ablation bar chart..."); categories = ['Full Model', 'No Free Energy\n(P=0)', 'No Memory\n(m=0)', 'No Learning']; values = [0.998, 0.0, 0.501, 0.153]
plt.figure(figsize=(7, 5)); bars=plt.bar(categories, values, color=['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']); plt.ylabel('Achieved Surprise Rate (S_dot)');
```

```python
plt.title('Ablation Study: Impact of Prerequisites'); plt.xticks(rotation=15)
for bar in bars: yval=bar.get_height(); plt.text(bar.get_x()+bar.get_width()/2.0, yval+0.02, f'{yval:.3f}', ha='center', va='bottom')
plt.ylim(0, 1.2); plt.tight_layout(); plt.savefig('images/smt_ablation_bars.png', dpi=150); plt.close()
print("Generating static evolutionary trajectory plot..."); obs=FiniteMemoryObserver(m=2); current_k, current_p=2, 0.05; trajectory=[]
for _ in tqdm(range(100), desc="Evo Static"):
    gen=XORGenerator(k=current_k, p=current_p); rate=run_simulation(gen, obs, 500)['achieved_rate']; trajectory.append(rate)
    next_k_np=current_k+np.random.choice([-1,0,1]); next_p=current_p+np.random.normal(0,0.02); next_k_np=np.clip(next_k_np,1,10); next_p=np.clip(next_p,0,0.5)
    next_gen=XORGenerator(k=int(next_k_np), p=next_p); next_rate=run_simulation(next_gen, obs, 500)['achieved_rate']
    if next_rate > rate: current_k, current_p = int(next_k_np), next_p
plt.figure(figsize=(8,5)); plt.plot(trajectory, lw=2); plt.axhline(1.0, color='r', linestyle='--', label='Bound'); plt.title('Evolution of Surprise Rate vs. Static Observer (m=2)')
plt.xlabel('Generation'); plt.ylabel('Surprise Rate (S_dot)'); plt.ylim(0, 1.1); plt.legend(); plt.grid(True, alpha=0.5); plt.savefig('images/smt_evolution_trajectory.png', dpi=150); plt.close()

print("\n--- Generating Section 5: Extended Model Suite ---")
all_model_suite={'logistic': {'gen': LogisticMapGenerator(r=4.0), 'obs': FiniteMemoryObserver(m=4)},'rnn': {'gen': RNNGenerator(hidden_size=5), 'obs': FiniteMemoryObserver(m=3)},'eca': {'gen': CellularAutomatonGenerator(rule=30), 'obs': FiniteMemoryObserver(m=5)},'hsmm': {'gen': HSMMGenerator(), 'obs': FiniteMemoryObserver(m=5)},'pcfg': {'gen': PCFGGenerator(), 'obs': FiniteMemoryObserver(m=5)}}
model_filenames={'logistic':'logistic', 'rnn':'rnn', 'eca':'eca_rule30', 'hsmm':'hsmm', 'pcfg':'pcfg'}
for name, models in all_model_suite.items():
    print(f"Processing {name.upper()} model..."); rates = []
    for _ in tqdm(range(300), desc=f"Density {name}"):
        if name=='logistic': gen=LogisticMapGenerator(r=3.6+np.random.rand()*0.4)
        elif name=='rnn': gen=RNNGenerator(hidden_size=np.random.randint(2,8))
        elif name=='eca': gen=CellularAutomatonGenerator(rule=np.random.choice([30, 54, 90, 110]))
        elif name=='hsmm': gen=HSMMGenerator(num_states=np.random.randint(2, 5))
        else: gen=PCFGGenerator(num_rules=np.random.randint(2, 5))
        obs=FiniteMemoryObserver(m=np.random.randint(3,7)); rates.append(run_simulation(gen, obs, 1500)['achieved_rate'])
    plot_density_scatter(np.ones(len(rates)), np.array(rates), f'images/{name}_bound_scatter_density.png', f'{name.upper()} Model: Achieved Rate vs. Bound')
    result=run_simulation(models['gen'], models['obs'], 500); plot_time_series(result['time_series_data'], 1.0, f"images/{model_filenames[name]}_time_series.png", f'{name.upper()} Model: Approach to Steady State')

print("\n--- Generating Section 5.3: Homogeneous Co-evolution ---"); gen_k=2; obs_m=2; gen_history, obs_history=[],[]
for _ in tqdm(range(50), desc="Homogeneous Co-evo"):
    gen_history.append(gen_k); obs_history.append(obs_m); gen=XORGenerator(k=gen_k, p=0.1); obs=FiniteMemoryObserver(m=obs_m); current_rate=run_simulation(gen, obs, 500)['achieved_rate']
    mutant_gen_k=np.clip(gen_k+np.random.choice([-1,1]),1,15); mutant_gen=XORGenerator(k=int(mutant_gen_k), p=0.1)
    if run_simulation(mutant_gen, obs, 500)['achieved_rate']>current_rate: gen_k=
```

```
int(mutant_gen_k)
    gen_for_obs_test=XORGenerator(k=gen_k, p=0.1); mutant_obs_m=np.clip(obs_m+np.random.choice([-1,1]),1,15); mutant_obs=FiniteMemoryObserver(m=int(mutant_obs_m))
    if run_simulation(gen_for_obs_test, mutant_obs, 500)['achieved_rate']<current_rate: obs_m=int(mutant_obs_m)
plt.figure(figsize=(8,5)); plt.plot(gen_history, label='Generator Complexity (k)', lw=2); plt.plot(obs_history, label='Observer Complexity (m)', lw=2); plt.title('Co-evolutionary Arms Race (Red Queen Dynamics)'); plt.xlabel('Generation'); plt.ylabel('Complexity Parameter'); plt.legend(); plt.grid(True, alpha=0.5); plt.savefig('images/coevo_trajectory.png', dpi=150); plt.close()

print("\n--- Generating Section 5.4: Heterogeneous Co-evolution (XOR vs RNN) ---"); gen_k=2; obs_h=2; gen_history, obs_history=[],[]
for _ in tqdm(range(50), desc="XOR vs RNN Co-evo"):
    gen_history.append(gen_k); obs_history.append(obs_h); gen=XORGenerator(k=gen_k, p=0.1); obs=RNNObserver(hidden_size=obs_h); current_rate=run_simulation(gen, obs, 1000)['achieved_rate']
    mutant_gen_k=np.clip(gen_k+np.random.choice([-1, 1]),1,15); mutant_gen=XORGenerator(k=int(mutant_gen_k), p=0.1)
    obs_for_gen_test=RNNObserver(hidden_size=obs_h)
    if run_simulation(mutant_gen, obs_for_gen_test, 1000)['achieved_rate']>current_rate: gen_k=int(mutant_gen_k)
    mutant_obs_h=np.clip(obs_h+np.random.choice([-1, 1]),1,15); mutant_obs=RNNObserver(hidden_size=int(mutant_obs_h))
    gen_for_obs_test=XORGenerator(k=gen_k, p=0.1)
    if run_simulation(gen_for_obs_test, mutant_obs, 1000)['achieved_rate']<current_rate: obs_h=int(mutant_obs_h)
plt.figure(figsize=(8, 5)); plt.plot(gen_history, label='XOR Generator Complexity (k)', lw=2); plt.plot(obs_history, label='RNN Observer Complexity (H)', lw=2); plt.title('Heterogeneous Arms Race (XOR Generator vs. RNN Observer)'); plt.xlabel('Generation'); plt.ylabel('Complexity Parameter'); plt.legend(); plt.grid(True, alpha=0.5); plt.savefig('images/coevo_heterogeneous.png', dpi=150); plt.close()

print("\n--- Generating Section 6: Evolutionary Experiments with Costs ---")
def run_ga(observer, generator_class, cost_params, generations=40, pop_size=14):
    k_max=15
    if generator_class==XORGenerator: population=[XORGenerator(k=np.random.randint(2,5), p=np.random.rand()*0.2) for _ in range(pop_size)]
    else: population=[RNNGenerator(hidden_size=np.random.randint(2,4)) for _ in range(pop_size)]
    log={'avg_complexity':[], 'avg_ratio':[]}
    for _ in tqdm(range(generations), desc=f"GA m={getattr(observer, 'm', 'RNN')} C={cost_params['lambda_C']}", leave=False):
        fitness_scores, s_dots=[],[]
        for gen in population:
            result=run_simulation(gen, observer, 400); s_dot=result['achieved_rate']; s_dots.append(s_dot)
            fitness=(s_dot/1.0)-cost_params['lambda_E']*gen.get_power()-cost_params['lambda_C']*(gen.get_complexity()/k_max)
            fitness_scores.append(fitness)
        log['avg_complexity'].append(np.mean([g.get_complexity() for g in population])); log['avg_ratio'].append(np.mean(s_dots)); parents=[]
        for _ in range(pop_size): i,j=np.random.choice(range(pop_size),2,replace=False); winner=i if fitness_scores[i]>fitness_scores[j] else j; parents.append(population[winner])
        new_population=[]
        for i in range(0,pop_size,2):
```

```
            p1,p2=parents[i],parents[i+1]
            if generator_class==XORGenerator:
                k1_np=(p1.k+p2.k)//2+np.random.randint(-1,2); p1_=(p1.p+p2.p)/2+np.random.normal(0,0.02)
                c1=XORGenerator(k=int(np.clip(k1_np,1,k_max)), p=np.clip(p1_,0.01,0.5))
                k2_np=(p1.k+p2.k)//2+np.random.randint(-1,2); p2_=(p1.p+p2.p)/2+np.random.normal(0,0.02)
                c2=XORGenerator(k=int(np.clip(k2_np,1,k_max)), p=np.clip(p2_,0.01,0.5))
            else:
                h1_np=(p1.H+p2.H)//2+np.random.randint(-1,2); c1=RNNGenerator(hidden_size=int(np.clip(h1_np,1,k_max)))
                h2_np=(p1.H+p2.H)//2+np.random.randint(-1,2); c2=RNNGenerator(hidden_size=int(np.clip(h2_np,1,k_max)))
            new_population.extend([c1,c2])
        population=new_population
    return log
cost_regimes={'none':{'lambda_E':0.0,'lambda_C':0.0}, 'moderate':{'lambda_E':0.12,'lambda_C':0.03}, 'high':{'lambda_E':0.30,'lambda_C':0.06}}
for model_name, gen_class in [('xor',XORGenerator), ('rnn',RNNGenerator)]:
    for m_val, obs_strength in [(1,'weak'), (4,'strong')]:
        print(f"Running GA for {model_name.upper()} vs {obs_strength} observer (m={m_val})..."); observer=FiniteMemoryObserver(m=m_val); results={}
        for name, params in cost_regimes.items(): results[name]=run_ga(observer,gen_class,params)
        gens=range(40)
        plt.figure(figsize=(8,4));
        for name,data in results.items(): plt.plot(gens, data['avg_complexity'], label=f'{name} cost', lw=2)
        plt.ylabel(f'Mean Complexity ({"k" if model_name=="xor" else "H"})'); plt.title(f'{model_name.upper()} vs {obs_strength.capitalize()} Observer (m={m_val}): Complexity Evolution')
        plt.xlabel('Generation'); plt.legend(); plt.grid(True, alpha=0.5); plt.savefig(f'images/evo_{model_name}_m{m_val}_complexity.png', dpi=150); plt.close()
        plt.figure(figsize=(8,4));
        for name,data in results.items(): plt.plot(gens, data['avg_ratio'], label=f'{name} cost', lw=2)
        plt.ylabel('Mean Surprise Ratio (S_dot/Bound)'); plt.title(f'{model_name.upper()} vs {obs_strength.capitalize()} Observer (m={m_val}): Ratio Evolution')
        plt.xlabel('Generation'); plt.legend(); plt.grid(True, alpha=0.5); plt.ylim(0,1.1); plt.savefig(f'images/evo_{model_name}_m{m_val}_ratio.png', dpi=150); plt.close()

print("\n--- Generating Section 7: IC vs. Surprise Experiment ---")
k_range=np.arange(1,16); surprise_vs_weak, surprise_vs_strong=[],[]
for k in tqdm(k_range, desc="IC vs Surprise"):
    gen_weak=XORGenerator(k=int(k), p=0.1); obs_weak_run=FiniteMemoryObserver(m=2); surprise_vs_weak.append(run_simulation(gen_weak, obs_weak_run, 2000)['achieved_rate'])
    gen_strong=XORGenerator(k=int(k), p=0.1); obs_strong_run=FiniteMemoryObserver(m=8); surprise_vs_strong.append(run_simulation(gen_strong, obs_strong_run, 2000)['achieved_rate'])
plt.figure(figsize=(8,5)); plt.plot(k_range, surprise_vs_weak, 'o-', label='vs. Weak Observer (m=2)', lw=2); plt.plot(k_range, surprise_vs_strong, 's-', label='vs. Strong Observer (m=8)', lw=2)
plt.xlabel("Generator Computational Complexity (IC Proxy 'k')"); plt.ylabel("
```

```
    Achieved Surprise Rate (S_dot)"); plt.title("Observer Capability Forces Higher
    Generator Complexity")
plt.legend(); plt.grid(True, linestyle='--', alpha=0.7); plt.ylim(0,1.1); plt.
    xticks(np.arange(1,16,2)); plt.tight_layout(); plt.savefig('images/
    figure_IC_vs_Surprise.png', dpi=150); plt.close()

print("\n--- All simulations complete. Zipping results. ---")
shutil.make_archive('simulation_images', 'zip', 'images')
print("\nDone! You can now download 'simulation_images.zip' from the Colab file
    browser on the left.")
```

**Listing 1:** Complete Python script for simulation and figure generation.