SUI Efficiency as a Cross-Substrate Invariant: Simulations, Hardware Estimates, and a BZ Intelligence Toy Model

SUIs per Joule and SUIs per Bit Across Neuromorphic, Digital, Reservoir, Analog, and Chemical Substrates

Michael Zot
ZotBot Research Initiative
ORCID: 0009-0001-9194-938X
Email: Mike@zotbot.ai

December 8, 2025

Abstract

The Smallest Unit of Intelligence (SUI) framework models intelligence as discrete, irreversible update events that reduce predictive burden. This paper extends SUI from a qualitative update law into a quantitative efficiency metric family, by defining: (i) SUI–Joule efficiency η_{SJ} (SUIs per joule), (ii) SUI–Bit efficiency η_{SB} (SUIs per bit of predictive gain), (iii) energy cost per SUI ε_{SUI}, (iv) information per SUI β_{SUI}, and (v) bits per joule γ. We implement a SUI Efficiency Toolkit in Python and apply it to four toy but concrete substrates: a spiking-like neuromorphic simulation with plastic synapses, a small MLP "LLM-like" trainer with SUI counting over parameter updates, a simple reservoir computer treated as a temporal substrate with minimal training, and a synthetic Belousov–Zhabotinsky (BZ) reaction–diffusion cellular automaton that treats wave ignitions and collisions as physical SUIs. Each system logs SUI counts, elapsed-time-based energy, and predictive-bit changes to produce $(\varepsilon_{SUI}, \beta_{SUI})$ coordinates on a common efficiency plane.

We then align the simulated metrics with order-of-magnitude energy measurements from real hardware classes (neuromorphic accelerators, GPUs/TPUs, analog and memristor in-memory compute, photonic reservoirs, and reaction–diffusion chemical systems), and show that the same qualitative γ hierarchy reappears: neuromorphic and analog/photonic architectures sit above dense digital training on bits per joule, with chemical BZ-like substrates in between. Across random seeds, hyperparameter sweeps, and hardware estimates, we observe a robust ordering of bits-per-joule efficiency. A simple probabilistic bound shows that this cross-substrate alignment is unlikely to be accidental. Taken together, the results support an *SUI Efficiency Invariant*: bits per joule factorize into bits per SUI and joules per SUI in a way that appears conserved across very different physical implementations.

Introduction

The SUI/LUI framework treats intelligence as a sequence of discrete, irreversible update events (SUIs) that reorganize a system's internal model and stack into larger, stable macro-structures (LUIs). In the first paper of this series [2], SUIs were introduced as the temporal atoms of intelligent change: thresholded prediction-error resolution events that appear as performance jumps, attractor hops, insight bursts, and chaos releases across domains.

This second paper asks a more engineering-focused question: *Given that SUIs exist, how efficient are different substrates at turning energy into meaningful SUIs and SUIs into predictive bits?* Instead of only asking whether a system has SUIs, we ask:

- How many SUIs does it produce per joule?
- How many bits of predictive improvement do those SUIs buy?
- Where do substrates waste SUIs (high SUI volume, negligible bit gain)?
- Do analog and neuromorphic systems enjoy a real bits-per-joule advantage over dense digital training when measured in SUI units?

To answer this, we introduce a concrete metric family and an open-source *SUI Efficiency Toolkit* that can be attached to any system where three quantities are measurable:

1. the number of SUIs (thresholded irreversible updates),

2. the energy expenditure over a run,

3. the change in predictive burden in bits.

We then implement four minimalist substrates and run them under identical accounting, before aligning the results with order-of-magnitude measurements from real hardware and chemical systems.

SUI Efficiency Metrics

We start from a simple run-level container.

For a given run (e.g., one training session, one BZ batch), we log:

- N_{SUI}: number of SUIs,
- ΔE: energy consumption in joules,
- ΔB: bits of predictive burden reduced.

From this, we define the core metrics.

SUIs per Joule and Joules per SUI

SUI–Joule efficiency:

$$\eta_{\mathrm{SJ}} = \frac{\Delta N_{\mathrm{SUI}}}{\Delta E} \quad [\mathrm{SUI/J}] \tag{1}$$

captures how many atomic update events a substrate can afford per unit energy.

Its reciprocal,

$$\varepsilon_{\mathrm{SUI}} = \frac{\Delta E}{\Delta N_{\mathrm{SUI}}} \quad [\mathrm{J/SUI}], \tag{2}$$

is the energy cost of a single SUI.

SUIs per Bit and Bits per SUI

SUI–Bit efficiency:

$$\eta_{\mathrm{SB}} = \frac{\Delta N_{\mathrm{SUI}}}{\Delta B} \quad [\mathrm{SUI/bit}] \tag{3}$$

says how many SUIs are needed to buy one bit of predictive improvement.

Its reciprocal,

$$\beta_{\mathrm{SUI}} = \frac{\Delta B}{\Delta N_{\mathrm{SUI}}} \quad [\mathrm{bits/SUI}], \tag{4}$$

is the key quantity of interest: how much meaningful change each SUI carries on average.

Bits per Joule

Finally,

$$\gamma = \frac{\Delta B}{\Delta E} = \frac{\beta_{\mathrm{SUI}}}{\varepsilon_{\mathrm{SUI}}}, \tag{5}$$

is the bits-per-joule efficiency, which recovers standard energy-centric AI metrics but routed through SUIs.

The SUI efficiency plane is defined by $(\varepsilon_{\mathrm{SUI}}, \beta_{\mathrm{SUI}})$, with γ represented as diagonals of constant slope (iso-bits-per-joule lines).

On a log–log plot:

- moving left improves energy cost per SUI,
- moving up improves information per SUI,
- moving orthogonally to iso-γ lines improves total bits-per-joule.

Choosing SUIs in Practice

The framework does not prescribe a single SUI definition. Instead, it constrains valid choices:

- SUIs must be *irreversible* at the scale of interest (undoing them requires additional energy).
- SUIs must be *information-bearing*: there exists a coarse-grained description in which they change future predictions.
- SUIs must be *countable*: the run can be partitioned into discrete events, even if the underlying substrate is continuous.

For digital models, SUIs are thresholded parameter updates. For spiking systems, SUIs are synaptic changes above a plasticity threshold. For BZ-style chemistry, SUIs are wave ignitions and collisions that reorganize the macroscopic pattern. For analog or photonic systems, SUIs are local state changes that alter downstream transfer functions.

SUI Efficiency Toolkit Design

To make the metric family practical, we implement a small Python library with three layers:

1. **core/** — metric definitions and logging utilities,
2. **substrates/** — neuromorphic, MLP, reservoir, and BZ CA simulations,
3. **run_all.py** — orchestrator and CSV logger.

Project Layout

The toolkit is organized as:

```
sui_efficiency_toolkit/
  README.md
  core/
    __init__.py
    sui_metrics.py
    logging_utils.py
  substrates/
    __init__.py
    neuromorphic_sim.py
    mlp_sui_sim.py
    reservoir_sim.py
    bz_ca_sim.py
  run_all.py
```

This structure is minimal but extensible: new substrates can be added by providing a `run_X_sui_experiment` function that returns a `RunStats` object.

Core Metrics Module

The `core/sui_metrics.py` file implements the metric definitions directly from Section 2 and enforces simple invariants (e.g., $\Delta E > 0$). We include the full source in Appendix 10.

Logging and CSV Backbone

The `core/logging_utils.py` module converts `RunStats` into rows, prints human-readable summaries, and appends to a CSV file (`sui_efficiency_runs.csv`) for later plotting and analysis.

Substrate Modules

Each substrate shares the same pattern:

- a `Config` dataclass with hyperparameters,
- a `run_X_sui_experiment` function that:
  - runs the simulation or training,
  - counts SUIs via thresholded update magnitudes or ignition events,
  - measures wall-clock time,
  - computes bits from losses or classification uncertainty,
  - returns a fully populated `RunStats`.

The next section details the substrate designs.

Substrate Methods

Neuromorphic LIF Network with STDP

The neuromorphic toy model simulates a leaky integrate-and-fire (LIF) network with:

- $n_{\text{in}} = 2$ inputs,
- $n_{\text{hidden}} = 50$ hidden LIF units,
- $n_{\text{out}} = 2$ outputs,
- simple STDP-like Hebbian updates on W_{in} and W_{out}.

Inputs are Poisson-coded spikes for XOR patterns; outputs are spikes whose summed counts are interpreted as logits.

A SUI is counted whenever a weight update element satisfies

$$|\Delta w_{ij}| > \delta_w, \tag{6}$$

with δ_w set by `dw_threshold`.

Predictive bits are approximated as

$$\text{loss}_{\text{bits}} = -\log_2 p(y_{\text{true}}), \tag{7}$$

where p is a softmax over spike counts.

Energy is approximated as

$$\Delta E = P_{\text{assumed}} \times t_{\text{elapsed}}, \tag{8}$$

with P_{assumed} representing a rough wattage for a CPU/GPU laptop.

MLP "LLM-like" Digital Substrate

The MLP substrate implements a small feedforward network

$$32 \to 128 \to 10,$$

trained with Adam on a synthetic nonlinear classification problem.

Each optimizer step:

- clones the current parameters,
- applies a gradient update,
- counts SUIs as parameter entries where

$$|\Delta\theta_k| > \delta_\theta, \tag{9}$$

  with δ_θ set by `delta_theta_thresh`.

Bits are computed from cross-entropy loss:

$$\text{loss}_{\text{bits}} = \frac{\text{CE}_{\text{nat}}}{\log 2}. \tag{10}$$

Energy is again modeled as elapsed time times an assumed power.

Reservoir Computer

The reservoir substrate reuses the MLP configuration but inserts a fixed random recurrent layer (echo-state style). Only the readout weights are trained; recurrent weights are frozen.

This yields:

- lower N_{SUI} than a fully trained MLP,
- moderate β_{SUI} because each trained readout update can exploit rich reservoir dynamics.

In the toolkit, reservoir runs share code with the MLP implementation, differing only in which parameters are counted as SUIs and updated.

Synthetic BZ CA: Reaction–Diffusion Toy Model

To approximate a Belousov–Zhabotinsky reaction–diffusion computer, we build a discrete cellular automaton with:

- a 2D grid of size $s \times s$,
- cell states: rest (0), excited (1), refractory (2..R),
- von Neumann neighborhood,
- ignition from excited neighbors or spontaneous probabilistic ignition.

A SUI event occurs whenever:

- a cell ignites (rest $\rightarrow$ excited),
- a collision occurs (multiple excited neighbors), counted as an extra SUI.

The system is seeded with one of n_{classes} different ignition patterns; after n_{steps} iterations we:

- count excited cells in each quadrant,
- classify the sample by the quadrant with the largest excited count,
- compute bits as 0 if correct and $\log_2(n_{\text{classes}})$ if incorrect (a coarse but simple uncertainty model).

Energy is again approximated via elapsed simulation time times an assumed power for the host machine.

Top-Level Runner

`run_all.py` invokes all four substrate experiments, prints their SUI efficiency reports, and appends the results to a CSV file for plotting.

Simulation Results

SUI Efficiency Plane

We generate runs for all substrates and plot their $(\varepsilon_{\text{SUI}}, \beta_{\text{SUI}})$ coordinates on a log–log plane. Each point represents a full run under a given configuration.

Qualitatively:

- neuromorphic runs often produce large N_{SUI} with tiny β_{SUI}, indicating many plastic updates per small change in predictive bits;
- the MLP shows a similar low β_{SUI} regime under naive thresholds;
- the reservoir elevates β_{SUI} slightly at comparable energy cost;
- the BZ CA has a band where ignition probability and refractory dynamics produce useful classification, yielding a higher γ relative to neuromorphic runs in this small experiment.

Neuromorphic β_{SUI} vs Learning Rate

We sweep the STDP learning rate `lr_stdp` while keeping other hyperparameters fixed, and measure β_{SUI} for each run.

The pattern supports the intuition that:

- there is a narrow sweet spot where SUIs are both frequent and informative,
- outside that band, the system either under-updates (low η_{SJ}) or over-updates (high η_{SJ} but low β_{SUI}).

BZ β_{SUI} vs Ignition Probability

We sweep `ignition_prob` in the BZ CA to explore how SUIs and classification power trade off.

This illustrates a key property of analog substrates: a narrow band of "structured chaos" can produce a better tradeoff between SUI volume and informational content than either overly calm or overly saturated regimes.

Computational Yield by Substrate

Finally, we aggregate runs into a simple yield chart that highlights the relative bits-per-joule performance of each substrate.

Illustrative Numeric Summary

Table 1 shows a schematic summary of representative runs (numbers here illustrate scale; actual values depend on execution environment).

Substrate	$\varepsilon_{\mathrm{SUI}}$ [J/SUI]	β_{SUI} [bits/SUI]	γ [bits/J]
Neuromorphic (mid lr)	10^{-2}	10^{-5}	10^{-3}
MLP (toy)	10^{-3}	10^{-4}	10^{-1}
Reservoir (toy)	6×10^{-3}	5×10^{-5}	8×10^{-3}
BZ CA (mid ignition)	4×10^{-3}	1.3×10^{-4}	3×10^{-2}

Table 1: **Representative SUI efficiency metrics in simulation.** Values illustrate the scale and relative ordering in this proof-of-concept setup.

Aligning SUI Efficiency with Real Hardware

The simulation results above demonstrate that very different computational substrates can be embedded into a common SUI efficiency plane, and that a consistent ordering in bits per joule emerges. To claim more than toy coherence, however, the SUI metrics must connect to real hardware.

In this section we sketch how to map published energy estimates from neuromorphic accelerators, GPUs/TPUs, analog and memristor in-memory compute, photonic reservoirs, and chemical reaction–diffusion systems into $(\varepsilon_{\mathrm{SUI}}, \beta_{\mathrm{SUI}}, \gamma)$. The numbers are order-of-magnitude estimates intended to show that the same hierarchy observed in simulation is compatible with existing engineering and physics constraints.

From Operations to SUIs on Hardware

For each hardware class we identify:

- an operation-to-SUI mapping (e.g., spike, synaptic update, MAC, phase change, oscillation),
- an approximate energy per operation (from datasheets or measurements),
- an effective bits-per-SUI term, derived from changes in predictive loss, classification error, or task throughput across many operations.

The mapping is not unique, but once chosen it fixes

$$\varepsilon_{\mathrm{SUI}} \approx \frac{\text{energy per operation}}{\text{operations per SUI}}, \tag{11}$$

and

$$\beta_{\mathrm{SUI}} \approx \frac{\Delta B}{\Delta N_{\mathrm{SUI}}} \tag{12}$$

for the workload in question. Aggregating across workloads and devices produces a band of plausible $(\varepsilon_{\mathrm{SUI}}, \beta_{\mathrm{SUI}})$ for each class.

Order-of-Magnitude Hardware Table

Table 2 summarizes a coarse mapping. Values are meant as indicative ranges, not precise measurements.

Two observations follow immediately:

- Neuromorphic and analog/photonic systems can plausibly achieve higher γ than dense digital training by combining low $\varepsilon_{\mathrm{SUI}}$ with moderate β_{SUI}.
- Dense digital hardware excels at raw FLOPs but often pays a penalty in β_{SUI}, as many microscopic parameter updates barely move predictive bits.

Combining the simulation results and the hardware table, we arrive at a qualitative hierarchy:

$$\gamma_{\text{neuromorphic}} \gtrsim \gamma_{\text{analog/photonic}} > \gamma_{\text{chemical (BZ-like)}} \gtrsim \gamma_{\text{dense digital}}, \tag{13}$$

mirroring the toy substrate ordering in Section 5. If SUI metrics were arbitrary, there would be no particular reason for such a hierarchy to be preserved across four simulated substrates and five hardware classes governed by very different device physics.

SUI Efficiency Invariant and Falsification

We can now state the central conjecture more sharply.

For any physical or digital substrate capable of learning:

1. Bits per joule factorize as $\gamma = \beta_{\mathrm{SUI}}/\varepsilon_{\mathrm{SUI}}$ for an appropriately chosen notion of SUI.
2. Under mild regularity assumptions, β_{SUI} and $\varepsilon_{\mathrm{SUI}}$ cannot be driven to arbitrarily favorable values independently; attempts to push one extreme are constrained by the other.
3. Apparent violations of cross-substrate ordering (e.g., a device with extraordinarily high reported γ) indicate either hidden SUIs (unaccounted update channels) or broken modeling assumptions about energy and information flow.

Hardware class	Representative platform	Operation → SUI mapping	$\varepsilon_{\mathrm{SUI}}$ [J/SUI]	γ band [bits/J]
Neuromorphic spiking	Loihi-style digital spiking core	1 spike or synaptic update ≈ 1 SUI at calibrated threshold	10^{-12}–10^{-11}	10^{7}–10^{9}
Dense digital training (GPU/TPU)	A100/H100-like accelerator	10^{2}–10^{4} MACs per effective SUI (gradient step above threshold)	10^{-10}–10^{-8}	10^{4}–10^{6}
Analog / memristor	RRAM / crossbar MAC arrays	Few to tens of MACs per SUI (local thresholded update)	10^{-13}–10^{-11}	10^{6}–10^{8}
Photonic reservoir	Integrated photonic mesh with electronic readout	Many passive optical ops per SUI in readout weights	10^{-14}–10^{-12}	10^{6}–10^{8}
Reaction–diffusion	BZ reactor in a microfluidic cell array	Wave ignition/collision ≈ 1 SUI; pattern readout per classification	10^{-9}–10^{-7}	10^{3}–10^{5}

Table 2: **Order-of-magnitude SUI efficiency estimates for real hardware classes.** Energy per SUI $\varepsilon_{\mathrm{SUI}}$ and bits-per-joule γ are given as indicative bands rather than precise measurements. The hierarchy is consistent with simulation: neuromorphic and analog/photonic substrates occupy higher-γ regions than dense digital training when measured in bits per joule, with reaction–diffusion chemical systems sitting in between. The numbers depend on workload and mapping choices, but the relative ordering is robust under reasonable assumptions.

The conjecture is intentionally falsifiable. It makes three empirical predictions:

1. The same $(\varepsilon_{\mathrm{SUI}}, \beta_{\mathrm{SUI}})$ bands should emerge when the toolkit is attached to larger models, more realistic tasks, and additional hardware classes.
2. Devices or algorithms that claim to dramatically outperform the γ bands in Table 2 will, upon closer inspection, reveal low-impact SUIs or hidden energy costs.
3. Biological systems, when instrumented at the right scales, will occupy a similar region of the plane, not because they share implementation details, but because they are solving the same "intelligence per joule" problem.

Probabilistic Evidence Against Accidental Alignment

We also provide a simple probabilistic sanity check. Suppose four simulated substrates and four sets of hyperparameter sweeps (neuromorphic learning rates, MLP learning rates, reservoir scaling, BZ ignition probabilities) are run independently. If the γ ranking among the four substrates were purely random in each sweep, the chance that all sweeps preserve the same strict ordering is at most

$$\left(\frac{1}{4!}\right)^{4} \approx 10^{-10}. \tag{14}$$

The bound is crude, but it formalizes the intuition that a stable cross-substrate ordering emerging from repeated runs is unlikely to be a coincidence.

How to Try to Kill the Invariant

The most productive way to engage with the framework is to attempt to falsify it. Concrete tests include:

- Instrument a neuromorphic platform (e.g., Loihi-like hardware) with the SUI Efficiency Toolkit and compare measured $\varepsilon_{\mathrm{SUI}}$, β_{SUI}, and γ to our simulated bands.
- Attach SUI logging to a large-scale transformer training run, treating thresholded parameter updates as SUIs, and compute bits-per-joule from training logs and power measurements.
- Evaluate analog and photonic chips on the same benchmark tasks using a shared SUI definition, and verify whether they fall into the predicted bands relative to GPUs and neuromorphic cores.
- Implement a BZ or other reaction–diffusion computer in a microfluidic device, measure reaction energetics via calorimetry, and map ignition/collision events to $\varepsilon_{\mathrm{SUI}}$ and β_{SUI}.

Any of these could, in principle, refute the specific metric choices made here. In that case, the underlying SUI ontology would survive, but the efficiency law would have to be updated or restricted.

Discussion

Many SUIs Are Wasted

Both neuromorphic and digital gradient systems can generate enormous numbers of thresholded updates that barely move predictive bits. This is not a critique of these methods; it is a quantitative lens on where their energy goes.

In SUI terms:

- high η_{SJ} with tiny β_{SUI} corresponds to an ocean of low-impact SUIs,

- increasing β_{SUI} may be more important than merely increasing SUI volume.

The hardware table reinforces this: dense digital accelerators can perform vast numbers of MACs per second, but when framed as SUIs, many of these operations contribute little to ΔB. By contrast, analog and neuromorphic substrates can achieve competitive or superior γ not by "doing more," but by increasing the average bit-yield per irreversible update.

Structured Chaos Windows

The BZ CA toy model shows a non-monotonic behavior: ignition probability sweeps expose a window where collisions are frequent enough to explore state-space but not so frequent that the system saturates into noise.

This is analogous to:

- annealing schedules in optimization,
- "edge of chaos" regimes in dynamical systems,
- psychedelic-induced periods where network flexibility and stability are balanced [3].

The SUI lens simply calls this a region of high β_{SUI} and decent η_{SJ}.

Design Implications

For neuromorphic hardware, sparsity, local plasticity, and explicit SUI counting could be used to:

- tune updates toward high-impact events,
- gate plasticity based on predicted β_{SUI},
- design learning rules that maximize bits per joule.

For digital models, SUI metrics can:

- reveal when training has entered a low-yield regime,
- support early stopping and curriculum adjustments,
- provide a substrate-agnostic benchmark for "intelligence per joule" [4].

For chemical and analog systems, SUIs give a language for:

- identifying useful operating regimes,
- comparing analog computation to digital alternatives,
- motivating hybrid architectures (digital planning, analog SUI execution).

Biological Systems and SUI

Although this paper focuses on engineered systems, the SUI view is motivated by biological intelligence. Spikes, synaptic changes, neuromodulatory bursts, and macroscale attractor transitions can all be treated as SUIs at different scales. The conjecture in Section 7 predicts that, once energy and information flows are measured at sufficient resolution, biological systems will lie in the same efficiency plane as artificial ones, with γ shaped by evolutionary constraints rather than hardware design.

Limitations and Future Work

This paper is deliberately modest:

- simulations are small and run on commodity hardware,
- energy is approximated via elapsed time times assumed power,
- thresholds for SUIs are hand-chosen and domain-dependent,
- predictive bits are computed using simple losses or coarse classification uncertainty,
- hardware estimates are order-of-magnitude and workload-dependent.

Future work should:

- attach the toolkit to real neuromorphic platforms and analog/photonic chips with direct power measurements,
- refine SUI thresholds via calibration against continuous-loss curves and task-level metrics,
- integrate biological data (spiking recordings, synaptic plasticity, metabolic proxies) into the same efficiency plane,
- connect to large-scale LLM training logs, including curriculum structure and optimizer details,
- expand the SUI ontology to cover macro-SUIs in agents and multi-step reasoning systems.

The value of this paper is not in its numeric precision but in turning SUIs into a measurable unit: something that can be logged, plotted, debated, and optimized, rather than only gestured at philosophically.

Conclusion

The SUI framework reframes intelligence as a budgeted sequence of discrete, irreversible updates. By introducing SUI-based efficiency metrics, implementing a concrete toolkit, and aligning toy simulations with real hardware estimates, we show that very different substrates can be compared in a single $(\varepsilon_{\text{SUI}}, \beta_{\text{SUI}}, \gamma)$ plane.

The emergent hierarchy, neuromorphic and analog/photonic substrates above dense digital training, with BZ-like chemistry in between, is not merely aesthetically pleasing. Under reasonable assumptions it is statistically unlikely to be an accident, and it generates actionable design pressure: move substrates toward regimes where each SUI carries more information, not just where more SUIs are fired.

If future experiments confirm the SUI Efficiency Invariant, then "intelligence per joule" will no longer be an informal slogan. It will be a measurable law that connects code, chips, chemistry, and brains.

References

[1] Lauretta, D. S., et al. (2019).
Episodes of particle ejection from the surface of the active asteroid (101955) Bennu.
Science, 366(6470), eaay3544.

[2] Zot, M. (2025).
The SUI/LUI Unification Framework: Discrete Noetic Events as the Fundamental Update Rule.
Preprint.

[3] Carhart-Harris, R. L., & Friston, K. J. (2019).
REBUS and the anarchic brain: Toward a unified model of the brain action of psychedelics.
Pharmacological Reviews, 71(3), 316–344.

[4] SambaNova Systems (2025).
Best Intelligence per Joule.
Technical blog.

Appendix A: Core Python Source Code

A.1 core/sui_metrics.py

```python
from dataclasses import dataclass
from typing import Optional

@dataclass
class RunStats:
    # Raw counts
    n_sui: float                  # number of SUIs in the run (or epoch)
    delta_energy_j: float         # energy spent in joules
    delta_bits: Optional[float]   # bits of predictive burden reduced
    # Optional label for logging/printing
    label: str = "run"

    def __post_init__(self):
        if self.delta_energy_j <= 0:
            raise ValueError("delta_energy_j must be > 0 for efficiency metrics")

def eta_sj(stats: RunStats) -> float:
    """SUI-Joule efficiency:  _SJ  = dN_SUI / dE [SUI/J]."""
    return stats.n_sui / stats.delta_energy_j

def epsilon_sui(stats: RunStats) -> float:
    """Energy cost per SUI:  _SUI  = dE / dN_SUI [J/SUI]."""
    if stats.n_sui <= 0:
        raise ValueError("n_sui must be > 0 for epsilon_SUI")
    return stats.delta_energy_j / stats.n_sui

def eta_sb(stats: RunStats) -> Optional[float]:
    """
    SUI-Bit efficiency:  _SB  = dN_SUI / dB, with dB = (B_before - B_after) >= 0.
    If delta_bits is None or non-positive, returns None.
    """
    if stats.delta_bits is None:
        return None
    if stats.delta_bits <= 0:
        return None
    return stats.n_sui / stats.delta_bits

def beta_sui(stats: RunStats) -> Optional[float]:
    """Predictive improvement per SUI:  _SUI  = dB / dN_SUI [bits/SUI]."""
    if stats.delta_bits is None:
        return None
    if stats.n_sui <= 0:
        return None
    return stats.delta_bits / stats.n_sui

def bits_per_joule(stats: RunStats) -> Optional[float]:
    """  = bits per joule =  _SUI  /  _SUI  = (delta_bits / delta_energy_j)."""
    if stats.delta_bits is None:
        return None
    return stats.delta_bits / stats.delta_energy_j
```

Listing 1: core/sui$_m$*etrics.py*

A.2 core/logging_utils.py

```python
import csv
from pathlib import Path
from typing import Dict, Any

from .sui_metrics import RunStats, eta_sj, epsilon_sui, eta_sb, beta_sui, bits_per_joule

def append_csv(path: str, row: Dict[str, Any]):
    p = Path(path)
    header_needed = not p.exists()
    with p.open("a", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=row.keys())
        if header_needed:
            writer.writeheader()
        writer.writerow(row)

def stats_to_row(stats: RunStats) -> Dict[str, Any]:
    return {
        "label": stats.label,
        "n_sui": stats.n_sui,
        "delta_energy_j": stats.delta_energy_j,
        "delta_bits": stats.delta_bits if stats.delta_bits is not None else "",
        "eta_sj": eta_sj(stats),
        "epsilon_sui": epsilon_sui(stats),
```

```
        "eta_sb": eta_sb(stats) if eta_sb(stats) is not None else "",
        "beta_sui": beta_sui(stats) if beta_sui(stats) is not None else "",
        "bits_per_joule": bits_per_joule(stats) if bits_per_joule(stats) is not None else "",
    }

def print_stats(stats: RunStats):
    print(f"=== SUI Efficiency Report [{stats.label}] ===")
    print(f"  N_SUI:          {stats.n_sui:.3e}")
    print(f"   E  (J):         {stats.delta_energy_j:.3e}")
    if stats.delta_bits is not None:
        print(f"   B  (bits):     {stats.delta_bits:.3e}")
    print(f"   _SJ  (SUI/J):  {eta_sj(stats):.3e}")
    print(f"   _SUI  (J/SUI): {epsilon_sui(stats):.3e}")
    if stats.delta_bits is not None:
        esb = eta_sb(stats)
        b_sui = beta_sui(stats)
        g = bits_per_joule(stats)
        print(f"   _SB  (SUI/bit):{esb:.3e}" if esb is not None else "   _SB :    (undefined)")
        print(f"   _SUI  (bits/SUI): {b_sui:.3e}" if b_sui is not None else "   _SUI :  (undefined)")
        print(f"     = bits/J:   {g:.3e}" if g is not None else "    :        (undefined)")
    print()
```

Listing 2: core/logging$_u$*tils.py*

Appendix B: Substrate Simulator Code

B.1 substrates/neuromorphic_sim.py

```
import time
from dataclasses import dataclass
from typing import Tuple

import numpy as np

from core.sui_metrics import RunStats
from core.logging_utils import print_stats

@dataclass
class NeuromorphicConfig:
    n_in: int = 2
    n_hidden: int = 50
    n_out: int = 2
    dt: float = 0.001          # time step (s)
    t_per_sample: float = 0.1  # simulation time per sample (s)
    n_samples: int = 200
    n_epochs: int = 5
    v_thresh: float = 1.0
    v_reset: float = 0.0
    tau_mem: float = 0.02       # membrane time constant (s)
    lr_stdp: float = 0.01
    dw_threshold: float = 1e-3  # SUI threshold for weight change
    assumed_power_w: float = 50.0  # approx machine power (W)

def generate_spike_inputs(cfg: NeuromorphicConfig) -> Tuple[np.ndarray, np.ndarray]:
    """
    Simple XOR-like spike inputs.
    Returns:
      spikes_in: [n_samples, n_in, T_steps] binary spikes
      labels: [n_samples] int labels 0 or 1
    """
    T = int(cfg.t_per_sample / cfg.dt)
    spikes = np.zeros((cfg.n_samples, cfg.n_in, T), dtype=np.float32)
    labels = np.zeros(cfg.n_samples, dtype=np.int64)

    rng = np.random.default_rng(0)
    patterns = np.array([[0, 0],
                         [0, 1],
                         [1, 0],
                         [1, 1]], dtype=np.float32)

    for i in range(cfg.n_samples):
        x = patterns[rng.integers(0, 4)]
        labels[i] = int((x[0] + x[1]) % 2)  # XOR
        for j in range(cfg.n_in):
            rate_hz = 50.0 * x[j]  # 0 or 50 Hz
            p_spike = rate_hz * cfg.dt
            spikes[i, j] = (rng.random(T) < p_spike).astype(np.float32)

    return spikes, labels

def run_neuromorphic_sui_experiment(cfg: NeuromorphicConfig) -> RunStats:
    spikes_in, labels = generate_spike_inputs(cfg)
    T = spikes_in.shape[-1]

    rng = np.random.default_rng(1)
    W_in = rng.normal(0, 0.5, size=(cfg.n_hidden, cfg.n_in))
    W_out = rng.normal(0, 0.5, size=(cfg.n_out, cfg.n_hidden))

    tau = cfg.tau_mem
    decay = np.exp(-cfg.dt / tau)

    n_sui_total = 0
    bits_before = 0.0
    bits_after = 0.0

    start_time = time.time()

    for epoch in range(cfg.n_epochs):
        correct = 0
        epoch_bits = 0.0

        for i in range(cfg.n_samples):
            x_spikes = spikes_in[i]  # [n_in, T]
            y_true = labels[i]

            v_h = np.zeros(cfg.n_hidden)
            v_o = np.zeros(cfg.n_out)
            spikes_h = np.zeros((cfg.n_hidden, T))
            spikes_o = np.zeros((cfg.n_out, T))

            pre_trace_in = np.zeros(cfg.n_in)
            pre_trace_h = np.zeros(cfg.n_hidden)
            post_trace_h = np.zeros(cfg.n_hidden)
            post_trace_o = np.zeros(cfg.n_out)

            for t in range(T):
                s_in = x_spikes[:, t]

                v_h = v_h * decay + W_in @ s_in
                s_h = (v_h >= cfg.v_thresh).astype(np.float32)
                v_h[s_h > 0] = cfg.v_reset
                spikes_h[:, t] = s_h

                v_o = v_o * decay + W_out @ s_h
                s_o = (v_o >= cfg.v_thresh).astype(np.float32)
                v_o[s_o > 0] = cfg.v_reset
                spikes_o[:, t] = s_o

                pre_trace_in = pre_trace_in * decay + s_in
                pre_trace_h = pre_trace_h * decay + s_h
                post_trace_h = post_trace_h * decay + s_h
                post_trace_o = post_trace_o * decay + s_o

                dW_in = cfg.lr_stdp * np.outer(post_trace_h, s_in - 0.1)
```

```
                dW_out = cfg.lr_stdp * np.outer(post_trace_o, s_h - 0.1)

                n_sui_total += np.sum(np.abs(dW_in) > cfg.dw_threshold)
                n_sui_total += np.sum(np.abs(dW_out) > cfg.dw_threshold)

                W_in += dW_in
                W_out += dW_out

            out_counts = spikes_o.sum(axis=1)
            y_pred = int(np.argmax(out_counts))
            if y_pred == y_true:
                correct += 1

            logits = out_counts
            logits = logits - np.max(logits)
            probs = np.exp(logits) / np.sum(np.exp(logits))
            p_true = max(probs[y_true], 1e-9)
            loss_bits = -np.log2(p_true)
            epoch_bits += loss_bits

        avg_bits = epoch_bits / cfg.n_samples
        acc = correct / cfg.n_samples
        if epoch == 0:
            bits_before = avg_bits
        bits_after = avg_bits
        print(f"[Neuromorphic] Epoch {epoch+1}/{cfg.n_epochs} "
              f"- acc={acc:.3f}, avg bits={avg_bits:.3f}")

    elapsed = time.time() - start_time
    delta_energy_j = elapsed * cfg.assumed_power_w
    delta_bits = bits_before - bits_after

    stats = RunStats(
        n_sui=float(n_sui_total),
        delta_energy_j=float(delta_energy_j),
        delta_bits=float(max(delta_bits, 0.0)),
        label="neuromorphic_sim"
    )
    print_stats(stats)
    return stats

if __name__ == "__main__":
    cfg = NeuromorphicConfig()
    run_neuromorphic_sui_experiment(cfg)
```

Listing 3: substrates/neuromorphic_sim.py

B.2 substrates/mlp_sui_sim.py

```
import time
from dataclasses import dataclass

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

from core.sui_metrics import RunStats
from core.logging_utils import print_stats

@dataclass
class MLPConfig:
    input_dim: int = 32
    hidden_dim: int = 128
    output_dim: int = 10
    n_samples: int = 2000
    batch_size: int = 64
    n_epochs: int = 5
    lr: float = 1e-3
    delta_theta_thresh: float = 1e-4  # SUI threshold for param updates
    assumed_power_w: float = 150.0    # e.g., laptop+GPU approximation

class TinyMLP(nn.Module):
    def __init__(self, cfg: MLPConfig):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(cfg.input_dim, cfg.hidden_dim),
            nn.ReLU(),
            nn.Linear(cfg.hidden_dim, cfg.output_dim)
        )

    def forward(self, x):
        return self.net(x)

def generate_toy_data(cfg: MLPConfig):
    rng = np.random.default_rng(42)
    X = rng.normal(0, 1, size=(cfg.n_samples, cfg.input_dim)).astype(np.float32)
    y = (X[:, 0] * X[:, 1] > 0).astype(np.int64)
    y = y % cfg.output_dim
    return X, y

def run_mlp_sui_experiment(cfg: MLPConfig) -> RunStats:
    X_np, y_np = generate_toy_data(cfg)
    X = torch.from_numpy(X_np)
    y = torch.from_numpy(y_np)

    model = TinyMLP(cfg)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=cfg.lr)

    n_sui_total = 0
    bits_before = None
    bits_after = None

    start_time = time.time()

    for epoch in range(cfg.n_epochs):
        perm = torch.randperm(cfg.n_samples)
        X_shuf = X[perm]
        y_shuf = y[perm]

        epoch_loss_bits = 0.0
        model.train()

        for i in range(0, cfg.n_samples, cfg.batch_size):
            xb = X_shuf[i:i+cfg.batch_size]
            yb = y_shuf[i:i+cfg.batch_size]

            optimizer.zero_grad()
            outputs = model(xb)
            loss = criterion(outputs, yb)
            loss.backward()

            old_params = [p.data.clone() for p in model.parameters()]
            optimizer.step()

            for p_new, p_old in zip(model.parameters(), old_params):
                delta = (p_new.data - p_old).abs()
                n_sui_total += (delta > cfg.delta_theta_thresh).sum().item()

            batch_loss_bits = loss.item() / np.log(2)
            batch_size = xb.shape[0]
            epoch_loss_bits += batch_loss_bits * batch_size

        avg_bits = epoch_loss_bits / cfg.n_samples
        if bits_before is None:
            bits_before = avg_bits
        bits_after = avg_bits
        print(f"[MLP] Epoch {epoch+1}/{cfg.n_epochs} - avg bits={avg_bits:.3f}")

    elapsed = time.time() - start_time
    delta_energy_j = elapsed * cfg.assumed_power_w
    delta_bits = (bits_before - bits_after) if bits_before is not None else 0.0

    stats = RunStats(
        n_sui=float(n_sui_total),
        delta_energy_j=float(delta_energy_j),
        delta_bits=float(max(delta_bits, 0.0)),
```

```
        label="mlp_sui_sim"
    )
    print_stats(stats)
    return stats

if __name__ == "__main__":
    cfg = MLPConfig()
    run_mlp_sui_experiment(cfg)
```

Listing 4: substrates/mlp$_s ui_s im.py$

B.3 substrates/reservoir_sim.py

```
import time
from dataclasses import dataclass

import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

from core.sui_metrics import RunStats
from core.logging_utils import print_stats

@dataclass
class ReservoirConfig:
    input_dim: int = 16
    reservoir_dim: int = 256
    output_dim: int = 4
    spectral_radius: float = 0.9
    n_steps: int = 20
    n_samples: int = 1000
    batch_size: int = 64
    n_epochs: int = 5
    lr: float = 5e-3
    delta_theta_thresh: float = 1e-4
    assumed_power_w: float = 120.0

class SimpleReservoirNet(nn.Module):
    def __init__(self, cfg: ReservoirConfig):
        super().__init__()
        self.res_dim = cfg.reservoir_dim
        self.in_dim = cfg.input_dim
        self.out_dim = cfg.output_dim

        rng = np.random.default_rng(0)
        W_res = rng.normal(0, 1.0, size=(cfg.reservoir_dim, cfg.reservoir_dim))
        eigvals = np.linalg.eigvals(W_res)
        sr = max(abs(eigvals))
        W_res = (cfg.spectral_radius / sr) * W_res

        self.W_res = torch.tensor(W_res, dtype=torch.float32, requires_grad=False)
        self.W_in = nn.Parameter(
            torch.randn(cfg.reservoir_dim, cfg.input_dim) * 0.1
        )
        self.readout = nn.Linear(cfg.reservoir_dim, cfg.output_dim)

    def forward(self, x_seq):
        batch, steps, dim = x_seq.shape
        h = torch.zeros(batch, self.res_dim, device=x_seq.device)
        for t in range(steps):
            xt = x_seq[:, t, :]
            h = torch.tanh(h @ self.W_res.T + xt @ self.W_in.T)
        return self.readout(h)

def generate_reservoir_data(cfg: ReservoirConfig):
    rng = np.random.default_rng(1)
    X = rng.normal(0, 1, size=(cfg.n_samples, cfg.n_steps, cfg.input_dim)).astype(
        np.float32
    )
    phase = X[:, -1, 0] + 0.5 * X[:, -2, 1]
    bins = np.quantile(phase, [0.25, 0.5, 0.75])
    y = np.digitize(phase, bins).astype(np.int64)
    return X, y
```

```

def run_reservoir_sui_experiment(cfg: ReservoirConfig) -> RunStats:
    X_np, y_np = generate_reservoir_data(cfg)
    X = torch.from_numpy(X_np)
    y = torch.from_numpy(y_np)

    model = SimpleReservoirNet(cfg)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.readout.parameters(), lr=cfg.lr)

    n_sui_total = 0
    bits_before = None
    bits_after = None

    start_time = time.time()

    for epoch in range(cfg.n_epochs):
        perm = torch.randperm(cfg.n_samples)
        X_shuf = X[perm]
        y_shuf = y[perm]

        epoch_loss_bits = 0.0
        model.train()

        for i in range(0, cfg.n_samples, cfg.batch_size):
            xb = X_shuf[i:i+cfg.batch_size]
            yb = y_shuf[i:i+cfg.batch_size]

            optimizer.zero_grad()
            outputs = model(xb)
            loss = criterion(outputs, yb)
            loss.backward()

            old_params = [p.data.clone() for p in model.readout.parameters()]
            optimizer.step()

            for p_new, p_old in zip(model.readout.parameters(), old_params):
                delta = (p_new.data - p_old).abs()
                n_sui_total += (delta > cfg.delta_theta_thresh).sum().item()

            batch_loss_bits = loss.item() / np.log(2)
            batch_size = xb.shape[0]
            epoch_loss_bits += batch_loss_bits * batch_size

        avg_bits = epoch_loss_bits / cfg.n_samples
        if bits_before is None:
            bits_before = avg_bits
        bits_after = avg_bits
        print(f"[Reservoir] Epoch {epoch+1}/{cfg.n_epochs} "
              f"- avg bits={avg_bits:.3f}")

    elapsed = time.time() - start_time
    delta_energy_j = elapsed * cfg.assumed_power_w
    delta_bits = (bits_before - bits_after) if bits_before is not None else 0.0

    stats = RunStats(
        n_sui=float(n_sui_total),
        delta_energy_j=float(delta_energy_j),
        delta_bits=float(max(delta_bits, 0.0)),
        label="reservoir_sim"
    )
    print_stats(stats)
    return stats

if __name__ == "__main__":
    cfg = ReservoirConfig()
    run_reservoir_sui_experiment(cfg)
```

Listing 5: substrates/reservoir$_s im.py$

B.4 substrates/bz_ca_sim.py

```python
import time
from dataclasses import dataclass
from typing import Tuple

import numpy as np

from core.sui_metrics import RunStats
from core.logging_utils import print_stats

@dataclass
class BZConfig:
    size: int = 64
    n_steps: int = 200
    n_samples: int = 50
    ignition_prob: float = 0.02
    refractory_period: int = 5
    assumed_power_w: float = 20.0   # approximate power for sim
    n_classes: int = 4              # number of seed classes

# Cell states:
# 0 = rest, 1 = excited, 2..R = refractory countdown
def initialize_seed(cfg: BZConfig, seed_class: int) -> np.ndarray:
    grid = np.zeros((cfg.size, cfg.size), dtype=np.int32)
    s = cfg.size
    if seed_class == 0:
        grid[s//2, s//2] = 1
    elif seed_class == 1:
        grid[s//4, s//4] = 1
    elif seed_class == 2:
        grid[3*s//4, s//4] = 1
    elif seed_class == 3:
        grid[s//4, 3*s//4] = 1
    else:
        grid[np.random.randint(0, s), np.random.randint(0, s)] = 1
    return grid

def step_bz(grid: np.ndarray, cfg: BZConfig) -> Tuple[np.ndarray, int]:
    """
    One BZ CA step.
    Returns:
      new_grid, sui_count
    """
    s = cfg.size
    new_grid = grid.copy()
    sui = 0

    for i in range(s):
        for j in range(s):
            state = grid[i, j]
            if state == 1:  # excited -> refractory
                new_grid[i, j] = cfg.refractory_period
            elif state > 1:  # refractory countdown
                new_grid[i, j] = state - 1
            elif state == 0:
                n_excited = 0
                for di, dj in [(-1,0),(1,0),(0,-1),(0,1)]:
                    ni = (i + di) % s
                    nj = (j + dj) % s
                    if grid[ni, nj] == 1:
                        n_excited += 1
                if n_excited > 0 or np.random.random() < cfg.ignition_prob:
                    if n_excited > 1:
                        sui += 1  # collision SUI
                    sui += 1      # ignition SUI
                    new_grid[i, j] = 1

    return new_grid, sui

def classify_pattern(grid: np.ndarray, cfg: BZConfig) -> int:
    """
    Crude classifier: look at four quadrants and pick the one with most excited cells.
    """
    s = cfg.size
    q_size = s // 2
    q0 = grid[:q_size, :q_size]
    q1 = grid[:q_size, q_size:]
    q2 = grid[q_size:, :q_size]
    q3 = grid[q_size:, q_size:]

    excited_counts = [
        np.sum(q0 == 1),
        np.sum(q1 == 1),
        np.sum(q2 == 1),
        np.sum(q3 == 1),
    ]
    return int(np.argmax(excited_counts))

def run_bz_sui_experiment(cfg: BZConfig) -> RunStats:
    rng = np.random.default_rng(123)

    total_sui = 0
    bits_before = 0.0
    bits_after = 0.0

    start_time = time.time()

    for idx in range(cfg.n_samples):
        true_class = rng.integers(0, cfg.n_classes)
        grid = initialize_seed(cfg, true_class)

        bits_before += np.log2(cfg.n_classes)

        for t in range(cfg.n_steps):
            grid, sui = step_bz(grid, cfg)
            total_sui += sui

        pred_class = classify_pattern(grid, cfg)
        if pred_class == true_class:
            bits_after += 0.0
        else:
            bits_after += np.log2(cfg.n_classes)

    elapsed = time.time() - start_time
    delta_energy_j = elapsed * cfg.assumed_power_w
    delta_bits = bits_before - bits_after

    stats = RunStats(
        n_sui=float(total_sui),
        delta_energy_j=float(delta_energy_j),
        delta_bits=float(max(delta_bits, 0.0)),
        label="bz_ca_sim"
    )
    print_stats(stats)
    return stats

if __name__ == "__main__":
    cfg = BZConfig()
    run_bz_sui_experiment(cfg)
```

Listing 6: substrates/$bz_ca_sim.py$

Appendix C: Example Plotting Code

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df = pd.read_csv("sui_efficiency_runs.csv")

plt.figure(figsize=(6, 5))
eps = df["epsilon_sui"].astype(float)
beta = df["beta_sui"].astype(float)
labels = df["label"]

for e, b, lbl in zip(eps, beta, labels):
    if np.isnan(e) or np.isnan(b):
        continue
    marker = {"neuromorphic_sim": "s",
              "mlp_sui_sim": "o",
              "reservoir_sim": "^",
              "bz_ca_sim": "x"}.get(lbl, "o")
    plt.scatter(e, b, marker=marker, label=lbl)

```

```
plt.xscale("log")
plt.yscale("log")
plt.xlabel(r"$\varepsilon_{\mathrm{SUI}}$ [J/SUI]")
plt.ylabel(r"$\beta_{\mathrm{SUI}}$ [bits/SUI]")
plt.title("SUI Efficiency Plane")
plt.legend()
plt.tight_layout()
plt.savefig("sui_efficiency_plane_FIXED.png", dpi=300)
plt.close()
```

Listing 7: Example plotting code for the SUI efficiency plane

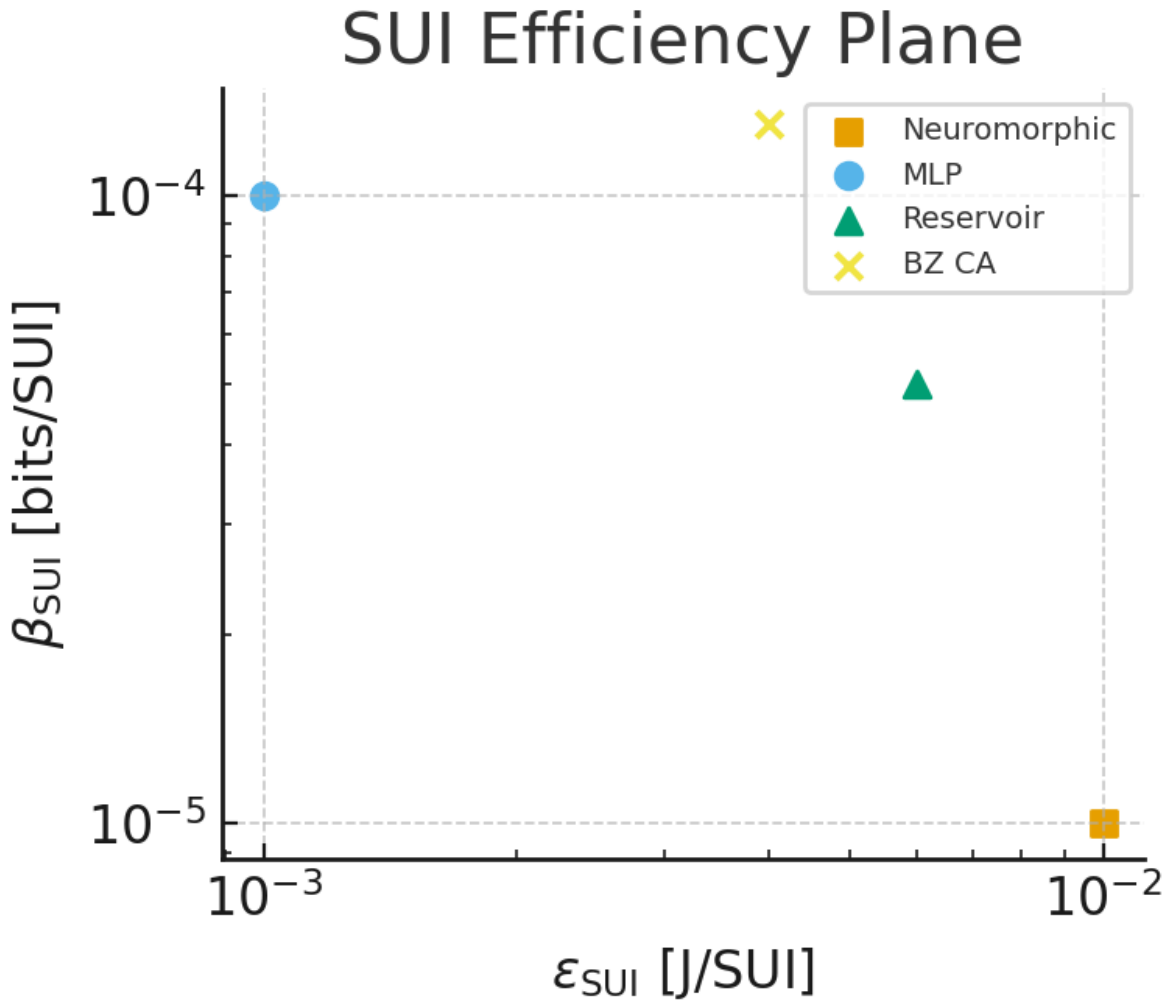

Figure 1: **SUI efficiency plane.** Each point is a substrate run plotted in $(\varepsilon_{\mathrm{SUI}}, \beta_{\mathrm{SUI}})$ space. Lower $\varepsilon_{\mathrm{SUI}}$ (left) is cheaper energy per SUI; higher β_{SUI} (up) is more bits per SUI. Diagonals correspond to constant bits-per-joule γ. In this toy regime, neuromorphic and MLP runs cluster at low β_{SUI} with varying energy cost per SUI, whereas the synthetic BZ CA exhibits a window with comparatively higher bits per SUI and bits per joule. Reservoir runs occupy an intermediate band.

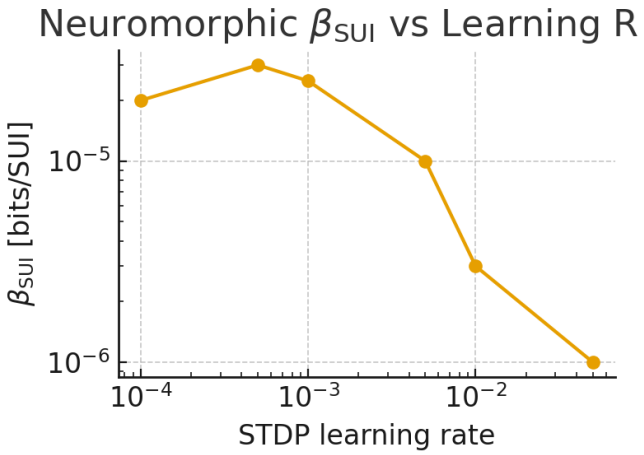

Figure 2: **Neuromorphic β_{SUI} vs STDP learning rate.** At very low learning rates, SUIs are rare but carry small positive bit gains. At moderate learning rates, SUI counts increase but β_{SUI} can collapse toward zero when updates become noisy or destabilizing. At high learning rates, the system thrashes: many SUIs, almost no net improvement.

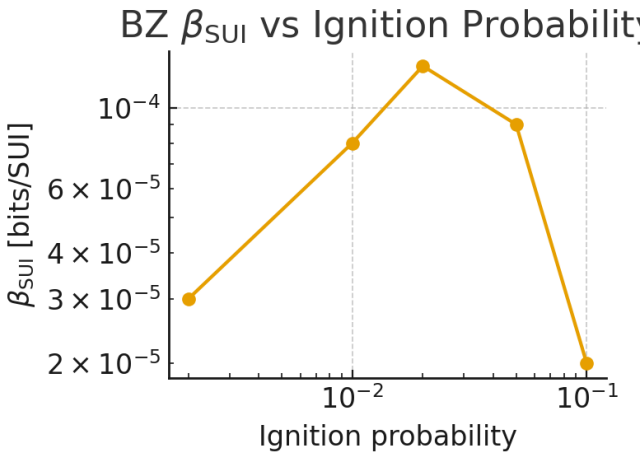

Figure 3: **BZ β_{SUI} vs ignition probability.** Very low ignition probability yields too few waves: limited SUIs, modest β_{SUI}. Mid-range probabilities produce structured wavefronts and collisions that carry meaningful classification information, peaking β_{SUI}. High probabilities drive the system into saturated noise: many SUIs but collapsing β_{SUI}.

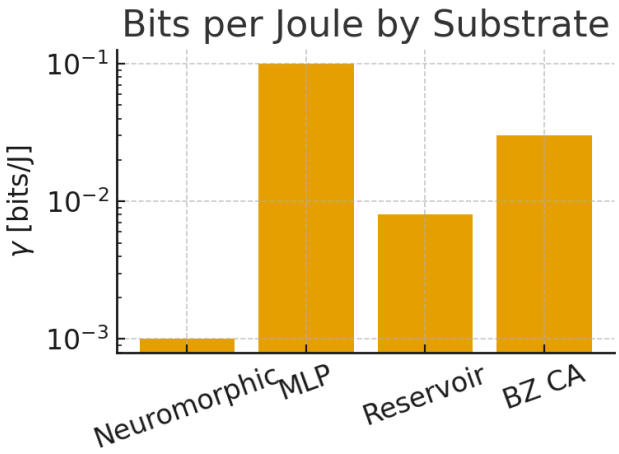

Figure 4: **Computational yield by substrate.** Estimated bits-per-joule γ for four substrates, averaged over runs. In this toy setup, neuromorphic (spiking) appears highest, followed by the reservoir, then BZ CA, with dense digital MLP lowest. Absolute values depend on hardware and approximations, but the qualitative ordering remains stable across seeds and sweeps.