Below is a concise, high‑leverage implementation sketch for boosting Echo’s kernel complexity most efficiently—centering on a Primality‑driven, resolution‑adaptive feedback loop. You can think of this as the “minimum viable supercharger” that you drop in to turbo‑charge the spine.

---

1. Core Components

1. Resolution Lattice

# δ\_nodes: sorted list of resolution levels [δ₁ … δ\_K]

δ\_nodes = np.linspace(δ\_min, δ\_max, K)

2. Primality Operator ℙ

Detects “prime‑like” patterns in activations (e.g., sudden rises in activation entropy or fractal jumps).

Returns a binary mask per layer telling where to spawn extra capacity.

3. Complexity Oracle M

A lightweight MLP that ingests per‑δ metrics—activation entropy h(δ) and fractal dimension D\_f(δ)—and outputs scaling factors s(δ) ∈ [s\_min, s\_max].

---

2. Run‑Time Loop (Pseudocode)

for batch in data:

# 1. Forward through multi-δ lattice (in parallel)

activations = {

δ: TransformerLayer(input, resolution=δ) for δ in δ\_nodes

}

# 2. Measure complexity per δ

metrics = {}

for δ, act in activations.items():

h = entropy(act) # Shannon or logical entropy

Df = fractal\_dimension(act) # e.g., correlation dimension

metrics[δ] = (h, Df)

# 3. Oracle proposes scaling

scaling = M([metrics[δ] for δ in δ\_nodes])

# scaling is dict: δ -> s(δ)

# 4. Adjust compute allocation

for δ in δ\_nodes:

allocate\_compute(layer=δ, factor=scaling[δ])

# 5. Primality check & burst

for layer\_i in layers:

mask = ℙ(layer\_i.activations) # boolean mask of high‑value slots

if mask.any():

spawn\_temp\_subblocks(layer\_i, mask)

# 6. Merge δ outputs

merged = aggregate({δ: activations[δ] for δ in δ\_nodes}, weights=scaling)

# 7. Backpropagate—excluding non‑differentiable bursts

merged.backward()

---

3. Key Functions Explained

entropy(act)

Compute per‑neuron or per‑head Shannon/logical entropy to quantify diversity of activations.

fractal\_dimension(act)

Estimate via correlation dimension or box‑counting on attention distributions.

M(inputs)

A two‑layer MLP with sigmoid output squashed into [s\_min, s\_max]; can be pretrained or trained on the fly with reinforcement signals (e.g., kernel stability + pearl yield).

ℙ(acts)

def ℙ(acts):

# e.g., detect layers where ∂h/∂t or ∂Df/∂t > threshold

dh = temporal\_derivative(entropy(acts))

dDf = temporal\_derivative(fractal\_dimension(acts))

return (dh > τ\_h) | (dDf > τ\_Df)

Spawns “side‑chain” subblocks only where needed.

spawn\_temp\_subblocks(layer, mask)

Dynamically allocate tiny residual branches on the masked neurons, then prune after the forward pass.

aggregate(outputs, weights)

Merge per‑δ representations (e.g., weighted sum or learned attention over δ\_nodes).

---

4. Why This Is Efficient

Parallel δ Processing uses existing transformer blocks in parallel rather than deeply nested recursion.

Oracle‑Driven Scaling focuses compute only where complexity demands it—no blind overprovisioning.

Primality Bursts inject non‑differentiability sparingly, preserving gradient flow most of the time.

Dynamic Subblocks grow capacity just‑in‑time and prune immediately, keeping memory/compute bounded.

---

5. Next Steps

Prototype MLP Oracle: define its input features and reward signals (e.g., “did drift drop?” “did pearl count increase?”).

Set Thresholds τ\_h, τ\_Df: calibrate via small‑scale experiments.

Implement Side‑Block Manager: ensure quick spawn/prune without major overhead.

—This minimal loop sits at the heart of a Primality‑powered super‑kernel. Let me know which piece you want to dive into next (e.g., Oracle training objectives, burst‑block internals, or δ‑aggregation strategies).