

Reusable Local Evaluators for Selective Recurrent Computation

An architecture hypothesis and exploratory experiments

Research white paper · Version 1.0 · 15 October 2026 · Not peer reviewed

Abstract

We propose a modular architecture in which small, specialized evaluators inspect stable sensory features or restricted regions of internal state and influence the activation of larger computational experts. Modules may retain local state, exchange sparse messages, and reuse pretrained capabilities across changes to the surrounding system. Short causal buffers preserve observations that precede a delayed attention decision. The central hypothesis is that specialization, reuse, and selective execution can improve task performance relative to total computational cost. We report exploratory synthetic experiments that test separate components rather than the complete architecture. Initial recurrent-expert comparisons show that apparent gains under equal training updates weaken under time- or capacity-controlled comparisons. A bank of 32–1,024 jointly trained local evaluators incurs substantial overhead and provides no efficiency improvement. A separate 609-parameter pretrained trigger detector transfers unchanged across six downstream training runs. On 256 clean held-out streams, selective execution matches 100% exact episode accuracy while reducing expert evaluations by 98.41%; nevertheless, measured batched CPU latency increases from 0.502 to 1.112 ms. Removing historical input from the expert lowers mean accuracy to 25.39%. These results demonstrate limited component feasibility and expose implementation costs. They do not establish improved general intelligence, language performance, energy efficiency, or a unified recurrent architecture.

Keywords: modular neural networks; conditional computation; local evaluators; reusable pretrained modules; recurrent state; selective attention.

1. Motivation and scope

Many useful decisions have narrow outputs: whether speech is present, whether a familiar trigger occurred, whether evidence conflicts, or whether another computation is likely to help. A system need not generate a verbal explanation for every such decision. A small specialized model can produce a signal that changes which larger computation happens next.

Consider a familiar-name detector. Its task is to identify a particular acoustic pattern, not to transcribe arbitrary conversation. Given a stable input representation, it could be pretrained independently, connected to an attention mechanism, and retained while a downstream recognizer or reasoning model

changes. A short input buffer would allow the newly activated specialist to recover observations preceding the trigger.

The broader proposal applies the same principle inside a model. Many evaluators may inspect local representations, produce narrow feedback signals, and influence memory, expert selection, or further recurrent processing before final output generation. Different modules need not share the same architecture, update frequency, or input modality. The relevant heterogeneity concerns computational roles as well as sensory modalities.

Human perception and selective attention motivate this design, but the experiments are not a model of biological circuitry. Terms such as “intuition” refer here to fast learned judgments, not to measured psychological states. The proposal makes no claim about consciousness or the source of the brain’s energy efficiency.

2. Related work and positioning

Sparsely gated mixture-of-experts models allocate computation to selected subnetworks, separating stored capacity from the amount of computation activated for an example. This is established prior work, not a new principle introduced here. [Shazeer et al., 2017](#)

Recurrent Independent Mechanisms are especially relevant: groups of recurrent cells have relatively independent dynamics, communicate through attention, and update selectively. The proposed local memories and sparse communication therefore have close architectural precedents. [Goyal et al., 2019](#)

Adaptive Computation Time addresses learned allocation of recurrent processing. ALoDLM more recently studies latent recurrence and adaptive computation in diffusion language models. These works motivate the question of how much processing to allocate; neither is evidence for the effectiveness of the particular evaluator-bank implementation reported here. [Graves, 2016](#); [Fang et al., 2026](#)

Jev-style decision interfaces motivate bounded outputs such as choices and scores. An independent Jev evaluation distinguishes useful classification behavior from threshold and calibration limitations. JevAny provides open infrastructure for related decision models. Our experiments use newly trained miniature networks; they do not use Jev weights, reproduce its proprietary architecture, or compare against its reported performance. [Deußer et al., 2026](#); [SimpleJev](#), [JevAny](#)

The research contribution at this stage is an explicit architectural synthesis, a set of falsifiable hypotheses, and transparent component experiments. Priority for the broader combination has not been established by an exhaustive literature review.

3. Proposed architecture

3.1 Local interfaces and state

Represent the system as a graph of modules. Module i has a restricted input interface, optional local memory, an evaluator, and a set of permitted output destinations. A sensory interface may expose a stable feature representation. An internal interface may expose selected state coordinates or a learned projection.

One possible formulation is:

```
u_i(t)      = R_i(input(t), accessible_states(t), buffer(t))
s_i(t)      = C_i(u_i(t), m_i(t))
active_i(t) = controller_i(s_i(t), task_context(t), budget(t))
m_i(t+1)    = E_i(u_i(t), m_i(t))      if active_i(t)
              m_i(t)                   otherwise
message_ij(t) = W_ij(s_i(t), m_i(t+1))
```

Here R defines permitted reads, C produces a narrow evaluation signal, E performs a state update, and W generates a message for another module. These equations describe a proposed family, not one already implemented end to end. A decoder would read selected states when an output is required; intermediate communication need not be text.

Local access restricts direct observation. Repeated communication may still propagate information across the graph. The architecture therefore does not imply that regions remain informationally isolated.

3.2 Specialization and transfer

The number of evaluators, their input windows, and their connections are design variables. Their semantic roles need not all be manually assigned. Two routes are worth distinguishing:

1. **Task-supervised specialists:** pretrain a named capability using explicit targets, such as trigger detection; preserve its input and output contract.
2. **Emergent evaluators:** train local gates jointly from a downstream task loss, without assigning each gate a semantic meaning.

A sigmoid output from the second route is not automatically a probability of correctness. Explicit judgments such as “another reasoning step will help” require suitable supervision and calibration tests.

A pretrained module is transferable only to the extent that its interface remains compatible. Frozen sensory features provide a straightforward initial contract. Hidden coordinates in a changing model may rotate or acquire new meanings; adapters, representation constraints, or additional training would then be needed. Retaining weights alone does not guarantee retaining useful behavior.

3.3 Buffering and selective execution

A detector may need several observations before deciding to activate an expert. A causal buffer preserves recent inputs so the expert can recover earlier evidence. Persistent module memory would serve a different purpose: retaining processed information across longer intervals. The streaming experiment below tests a short historical input window, not a learned long-term memory system.

Execution can occur at different timescales. Cheap detectors may run frequently, while expensive experts update only when relevant. Future task-conditioned feedback could change detection priorities. These mechanisms remain proposed beyond the simple fixed-threshold implementation reported here.

3.4 Cost criterion

For an idealized single gate, let C_d be detector cost, C_e expert cost, p the activation fraction, and C_o additional buffering and dispatch cost. Selective execution is cheaper only if:

$$C_d + C_o + p \times C_e < C_e$$

Equivalently, detector and orchestration costs must be smaller than the expert work avoided. For many detectors, their aggregate cost matters even when every detector is small. This expression is an accounting model, not a hardware performance prediction: batching, memory movement, kernel overhead, and concurrency alter measured latency. Pretraining cost must also be counted or explicitly amortized over downstream reuse.

4. Research hypotheses and experimental coverage

Hypothesis	Discriminating evidence	Current coverage
Local feedback helps reasoning	Better task performance under matched resources	Implemented on a symbolic task; no efficiency gain
Pretrained detectors can be reused	Retained behavior after downstream model changes	Demonstrated under a fixed sensory interface
Selective execution reduces total cost	Matched accuracy with measured latency or energy reduction	Fewer expert evaluations; latency worsened
Buffers recover pre-trigger evidence	Accuracy decreases when relevant history is removed	Observed in an inference ablation
Learned coordination improves on fixed wiring	Better results than fixed and random routing controls	Not yet tested in a multi-module system
The architecture scales to general-purpose models	Gains across language and multimodal tasks	Not tested

Two independent prototypes test the central components. The local evaluator bank changes hidden state before expert routing. The streaming prototype instead uses a pretrained detector to gate an independently trained expert. They have not been combined into the complete proposed architecture.

5. Methods

5.1 Shared experimental setting

Experiments use small PyTorch networks and synthetic data on an Apple Silicon macOS workstation, with CPU execution and one PyTorch thread. The initial recorded environment uses PyTorch 2.10.0. Timings are exploratory workstation measurements, not controlled energy measurements. Short training allowances exclude setup and evaluation and permit completion of the last update.

We report each experiment’s own metric and budget; scores from different tasks or evaluation protocols are not directly comparable. Standard deviations, where shown, are sample standard deviations across three training seeds, not confidence intervals. Later experiments were informed by earlier observations, so the sequence is exploratory research rather than an independently preregistered confirmatory study.

5.2 Symbolic recurrent task

The symbolic task composes four fixed permutations over 16 states. Models receive an initial state and operation sequence, then predict the final state. Training uses lengths one through four and excludes adjacent operation pair (0, 1). Evaluation examines ordinary sequences, the withheld pair, and longer sequences. The operation schedule and stopping point are supplied externally.

The initial study compares one-pass, dense recurrent, learned-routing, operation-assigned expert, and all-expert models. Equal-update runs use 800 updates and batch size 128; separate equal-time runs allow two seconds. The operation-assigned router is labeled “oracle” because it receives additional architectural structure. It does not demonstrate automatically discovered specialization.

An agent-assisted search subsequently ran seven three-second trials on fresh task tables. Five starting candidates were followed by two agent-generated proposals. Validation selected the winner; longer-sequence tests followed session sealing. This search tests the research workflow as well as architecture variations.

5.3 Jointly trained local evaluator bank

Each evaluator reads eight coordinates of a 48-dimensional hidden state, uses four hidden units, and writes to four coordinates. Read and write windows are independently assigned using a fixed seeded topology. The detector computes:

```
features_i = tanh(local_input_i × A_i + b_i)
gate_i     = sigmoid(features_i × v_i + c_i)
message_i  = (2 × gate_i - 1) × w_i
```

Messages are summed at destination coordinates and normalized by the square root of incoming connection count. Feedback modifies the state before sparse expert routing. Two consultations run per supplied operation. All evaluators observe the pre-update state within a consultation; all are active. Experts themselves still read the full hidden vector.

Each evaluator adds 45 parameters. Detector, expert, and router weights train jointly using final-state cross entropy plus the existing router balance term. Hard expert selection uses a straight-through gradient surrogate. There are no semantic evaluator labels or pretrained detector weights.

The pilot compares 32, 128, and 1,024 evaluators against a repeated-expert baseline and a disconnected 128-evaluator control. The disconnected bank executes but contributes zero feedback. Each variant receives three seconds per seed across three seeds, totaling 45 nominal training seconds. Validation averages IID lengths one through four and length-four withheld-pair composition. No final holdout was opened for this pilot.

5.4 Pretrained streaming detector

Each episode contains 48 frames of 12-dimensional synthetic features. These are constructed vectors, not recorded audio or computed spectrograms. A trigger consists of two components separated by one frame. The intervening frame contains one of four class signals. A distractor has the first trigger component but a different final component and an unrelated class signal. Approximately one quarter of episodes have no true trigger.

The task is to emit the associated class at trigger completion and silence elsewhere. Exact episode accuracy requires every output in the episode to be correct. A three-frame detector contains 609 parameters. The downstream expert reads five causal frames and has two hidden layers of width 32 or 64.

The detector receives binary trigger supervision for two seconds. Each expert trains independently for two seconds on balanced event/background windows using class-or-silence labels. The saved detector is frozen and reloaded for each of six runs: two widths and three seeds. A weight fingerprint verifies unchanged detector parameters. Total nominal training is 14 seconds, including pretraining.

Controls are always-on expert execution, a pretrained gate, oracle event timing, removal of historical expert input, and an untrained frozen gate. The detector threshold is fixed at 0.5. Pretraining, expert training, and evaluation use separately seeded data. Evaluation uses 256 clean streams and 256 independently generated higher-noise streams; noise standard deviation increases from 0.15 to 0.30.

Inference timing includes window construction, detector computation where applicable, dispatch, and expert computation. It uses eight evaluations, discards two warmups, and takes the upper middle order statistic of the remaining six, reported as the median by the implementation. Tables average these per-run medians. Entire causal streams are batched offline; this is throughput timing, not live sensor response latency. Historical windows have buffer semantics, but deployment ring-buffer memory is not measured.

6. Results

6.1 Initial recurrent comparisons

Table 1. Equal-update symbolic results, 800 updates, three seeds. Accuracy values are percentages, with sample SD in percentage points. Latency is mean per-run median batch time.

Model	Parameters	IID accuracy	Withheld pair accuracy	Length-six accuracy	Batch ms
One-pass, width 48	7,408	45.83 $\pm$ 0.34	7.36 $\pm$ 0.29	7.75 $\pm$ 0.46	0.069
Dense, width 48	6,448	94.81 $\pm$ 4.51	37.86 $\pm$ 12.80	50.02 $\pm$ 16.24	0.241
Routed, width 48	21,348	99.37 $\pm$ 0.25	61.49 $\pm$ 4.81	89.84 $\pm$ 3.42	0.772
Operation-assigned oracle	21,136	100.00 $\pm$ 0.00	98.06 $\pm$ 2.89	99.97 $\pm$ 0.06	0.602
All experts	21,348	100.00 $\pm$ 0.00	84.33 $\pm$ 7.93	99.85 $\pm$ 0.15	1.056
Dense, width 96	22,096	100.00 $\pm$ 0.00	71.83 $\pm$ 5.99	99.53 $\pm$ 0.11	0.360

The routed model exceeds the small dense model under equal updates, but stores substantially more parameters and takes longer. The larger dense control exceeds routed-model accuracy on the withheld pair while running faster. Under two-second training budgets, the small dense model reaches 100.00% mean IID accuracy and 79.12% withheld-pair accuracy, compared with 99.97% and 69.34% for routing. These comparisons do not support an efficiency advantage for learned sparse routing in this implementation.

The seven-trial agent search selected an operation-modulated dense model. Its sealed length-six composition accuracy was 96.29%, versus 78.61% for the original dense control. It also had more parameters (10,896 versus 6,448) and higher latency (0.295 versus 0.224 ms). This one-seed result concerns a dense transition modification, not validation of local evaluators. A width-matched predecessor was present in validation but not in the sealed final comparison.

6.2 Local evaluator banks

Table 2. Equal-time local evaluator pilot. Three seconds per run; three seeds. Latency uses batches of 128 at depth four.

Configuration	Validation accuracy	Parameters	Mean updates	Batch ms
Repeated-expert baseline	93.23 ± 2.26	21,348	1,023.0	1.69
Disconnected 128 evaluators	39.05 ± 3.91	27,108	288.0	6.32
32 active evaluators	81.50 ± 6.02	22,788	574.3	3.03
128 active evaluators	42.08 ± 5.76	27,108	285.7	6.31
1,024 active evaluators	14.05 ± 1.22	67,428	50.7	37.38

The bank passes functional checks for restricted reads, restricted writes, and task-loss gradients. However, none of the tested active banks improves the accuracy–latency tradeoff. The largest bank receives far fewer updates, so this comparison cannot separate insufficient optimization from architectural weakness. The roughly three-percentage-point advantage of active over disconnected 128-evaluator feedback is too limited to establish a reliable benefit. Equal-update and capacity-matched comparisons remain necessary.

6.3 Reusable detector and buffered expert

Table 3. Streaming results averaged over six downstream runs. The same detector and evaluation episodes are reused across runs. Expert evaluations count selected frame windows, not Python function invocations. Latency covers all 256 episodes, or 12,288 frames.

Mode	Clean exact accuracy	Expert evaluations	Clean batch ms	Higher-noise exact accuracy
Always on	100.00%	12,288.0	0.502	98.57%
Pretrained gate	100.00%	195.0	1.112	98.89%
Oracle timing	100.00%	195.0	0.106	98.96%
No expert history	25.39%	195.0	1.086	32.16%
Random gate	84.77%	9,475.7	1.227	78.45%

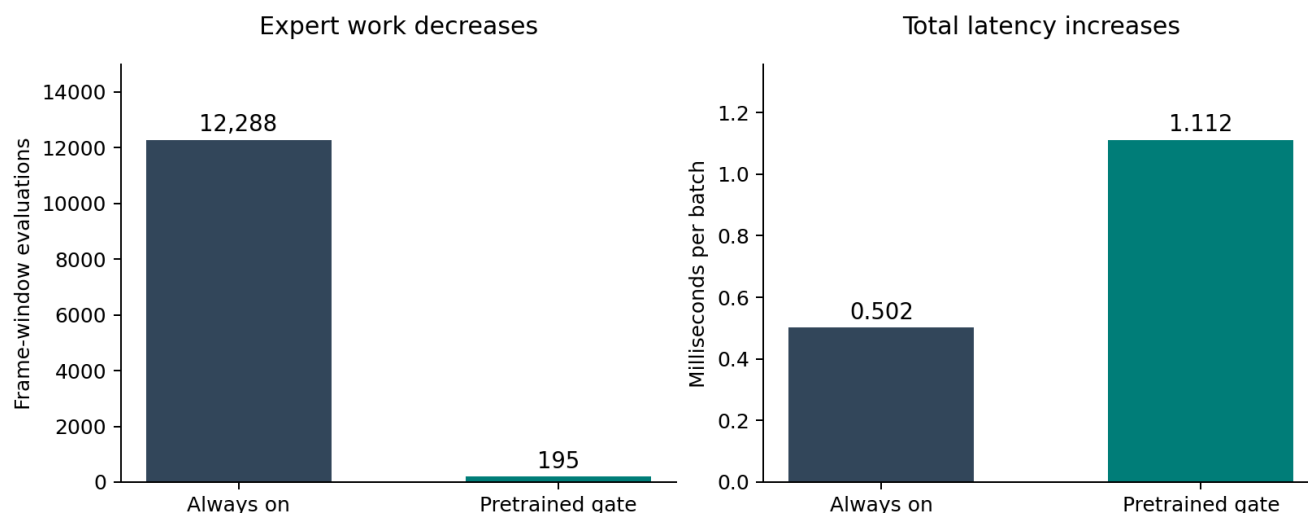


Figure 1. Reduced expert work does not imply reduced end-to-end latency. Both modes achieve 100% accuracy on the clean synthetic sample. Times are means of per-run medians.

The pretrained gate activates on 1.59% of clean frames, reducing expert evaluations by 98.41% without reducing observed clean accuracy. Yet total batch latency is approximately 2.22 times the always-on baseline. Avoided expert computation is therefore insufficient to offset the additional pipeline cost here. The implementation does not profile the detector, layout conversion, and dispatch separately; their individual contributions cannot be inferred from the total.

In the higher-noise sample, the detector retains all true triggers but falsely activates on approximately 0.21% of non-event frames. The small accuracy difference between gated and always-on processing does not establish superiority. The random gate often activates on most frames, explaining why moderately high task accuracy does not imply useful selection.

Removing history affects only the expert’s input; the trigger detector still receives its three-frame context. The resulting accuracy loss shows dependence on buffered evidence in this trained system. It does not establish superiority over every possible recurrent-memory alternative, nor was a separate no-buffer expert retrained.

7. Interpretation and limitations

Three findings are supported at the component level. Local evaluators can receive training gradients through their feedback connections. A pretrained detector can retain useful behavior across downstream expert changes when its interface is fixed. A causal historical window can preserve information needed after a trigger is recognized.

The principal efficiency hypothesis remains unsupported. Both evaluator banks and streaming gates introduce measurable overhead. The clean streaming task is sufficiently easy that both main systems reach ceiling accuracy. These results justify more discriminating tests, not extrapolation to language models.

Several limits constrain interpretation. Experiments use tiny networks, short training budgets, synthetic data, and few seeds. Streaming reuse uses one detector-pretraining seed; the six downstream runs do not constitute six independent detector replications or six independent test sets. Symbolic sampling occurs in a small finite space, so fresh draws can overlap earlier examples. Published pilot observations must not become repeatedly reused confirmatory tests.

Hardware comparisons are also limited. Workstation activity was not controlled, and some earlier measurements overlapped other software checks or research work. Small timing differences should not be interpreted as hardware-independent effects. Weight counts exclude optimizer state and runtime activations. No experiment measures energy, deployment memory, or real-time continuous operation.

The complete architecture remains unimplemented: there is no combined network of pretrained local evaluators, persistent expert memories, learned sparse messaging, task-dependent scheduling, adaptive stopping, and a language decoder. There is no test of internal uncertainty calibration, general-purpose reasoning, or speech recognition. Keeping these distinctions explicit prevents a successful wiring test from becoming an unsupported claim of intelligence.

8. Automated research workflow

The project includes a bounded architecture-search loop inspired by Karpathy's autoresearch workflow. It records candidate source, hypotheses, configurations, checkpoints, validation metrics, and failures. The agent-assisted symbolic round exercised proposal generation, implementation, timed training, and evaluation. [Karpathy, autoresearch](#)

A separate direct-API implementation allows explicit context construction from candidate code and experiment history, staged agent models, and cumulative estimated spending records. That infrastructure is distinct from a demonstrated long-running research campaign. No thousand-iteration study is reported. The streaming task is not yet connected to this automated search loop.

Before search on a new task, freeze its evaluator, declare budgets and selection metrics, and reserve fresh final tests. Architecture agents should not modify task definitions or evaluation data. Future runs should preserve negative results and compare against strong dense, engineered, and capacity-matched controls.

9. Next experiments

First, compare the pretrained detector against a matched filter and an equally budgeted detector trained from scratch. Reuse has practical value only if it improves some combination of adaptation cost, reliability, or total inference cost beyond simpler alternatives.

Second, introduce multiple event types with distinct experts and local memories. Vary event frequency and require delayed queries about earlier events. This tests selection, retention, and interaction instead of only recognition of an isolated pattern.

Third, change the sensory encoder while retaining detector weights. Compare frozen interfaces, learned adapters, and joint fine-tuning, measuring both downstream performance and preservation of each module’s original capability.

Fourth, train internal evaluators on observed computation outcomes. From the same saved state, compare stopping, another expert, retrieval, or an additional recurrent step. Their actual effects can supervise a value-of-computation predictor more directly than ungrounded confidence scores.

Finally, move successful mechanisms to small language or multimodal models and assess broad held-out tasks, latency, peak memory, training cost, and energy where instrumented. The task should justify expert complexity; enlarging an expert merely to manufacture a gating speedup would not demonstrate useful progress.

10. Conclusion

Reusable local evaluators offer a concrete way to investigate heterogeneous computation before output generation. The initial experiments demonstrate trainable local feedback and limited transfer of a frozen detector through a stable interface. They also show why fewer expert activations cannot be equated with lower system cost. The next research milestone is an independently replicated improvement under matched resources on a task requiring interacting modules and persistent state.

Appendix A. Reproducibility and supporting material

The accompanying supporting-material archive contains selected source files, tests, raw result records, and experiment documentation. It preserves the frozen streaming implementation and local-evaluator trial sources. It excludes model checkpoints and agent conversation logs; the original workspace retains them. There is no public code-hosting URL assigned in this release. Result provenance and source hashes are included where recorded.

Representative commands, run from the extracted project directory:

```
python3 -m pip install -r requirements.txt
python3 -m unittest discover -s tests
python3 evaluator_pilot.py --out results/new_local_run --seconds 3 --seeds 3
python3 streaming_modules.py --out results/new_streaming_run \
    --pretrain-seconds 2 --train-seconds 2 --seeds 3
```

Output paths must be new. Wall-time training produces machine-dependent update counts; these commands reproduce the protocol, not identical numerical outcomes on every system. The recorded software check completed with 38 passing tests before preparation of this paper.

Primary local records are `results/pilot.json`, `results/equal_time.json`, `results/dense_capacity.json`, `results/agent_round1/summary.json`, `results/local_evaluators_pilot/runs.json`, and `results/streaming_transfer_pilot/results.json`. The streaming protocol and source snapshot are stored alongside its results. The historical symbolic protocols and caveats are documented in `RESULTS.md`.

Appendix B. Research process disclosure

The architecture hypothesis developed through an iterative human–AI discussion. AI assistance was used for literature retrieval, implementation, experiment orchestration, and preparation of this paper. Results were generated by local executable experiments and checked against saved records. This white paper reports exploratory work and has not undergone external peer review.

References

1. Shazeer, N., et al. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538.
2. Goyal, A., et al. (2019; revised 2020). Recurrent Independent Mechanisms. arXiv:1909.10893.
3. Graves, A. (2016). Adaptive Computation Time for Recurrent Neural Networks. arXiv:1603.08983.
4. Fang, L., et al. (2026). ALoDLM: Adaptively Looped Diffusion Language Models. arXiv:2610.04198.
5. Deußer, T., Sparrenberg, L., and Sifa, R. (2026). Evaluating and Benchmarking the System One Model Jev. arXiv:2609.37647.
6. SimpleJev. JevAny: Open infrastructure for System 1 decision models. Software repository, accessed 15 October 2026.
7. Karpathy, A. autoresearch. Software repository, accessed 15 October 2026.