



Coarse-to-Fine Text Generation on a Laptop: A Native MLX Study of Next-Concept Modeling

Jan Music
Independent

Abstract

Next-Concept Modeling (NCM) generates a sequence through successive levels of discrete latent codes, predicting each level in parallel. We independently implement the method in MLX and study synthetic language and Sudoku on Apple Silicon and NVIDIA GPUs. On a finite synthetic grammar, a 5M-parameter generator achieves lower small-batch latency than a similarly sized autoregressive generator, with lower sentence validity (89.8% versus 98.6%). Two Sudoku training seeds use the same 6.44M-parameter generator recipe and 16.384M presentations each. Retained greedy evaluation aggregates report easy/medium/hard solve rates of 98.10/83.40/40.30% for seed 0 and 97.25/79.60/33.25% for seed 1, with 2,000 boards per difficulty. The hard-set difference is 7.05 percentage points; two seeds do not establish population training variability or its cause. Original NCM reports 99.50/96.50/86.10% with a 25.46M generator, 51.2M presentations and a different task setup. Our generator is 3.95 times smaller, excluding its additional codec; the comparison establishes neither task parity nor matched-cost efficiency. E20 completes its schedule without a recorded reset or truncation, but missing per-board predictions and remote provenance manifests limit independent audit. We characterize this phase as a partial implementation and small-model success, retain the substantial hard-Sudoku deficit, and define controlled text evaluation as the next research direction rather than a demonstrated capability.

Keywords: coarse-to-fine generation, residual vector quantisation, parallel decoding, Apple Silicon, MLX

Status. Working paper, version 0.11. Every number here is copied from a run in the code repository (`runs/`, summarised in `paper/data/results.json`) or quoted from the cited paper and marked as quoted. Experiments still running are marked *in progress* and never estimated. Appendix C lists the changes between versions.

1 Reading guide

This paper has two audiences. **If you are new to language models**, read Section 2 (the idea in plain language) and Section 3 (glossary), then the boxed summary at the start of each later section. **If you build models**, Sections 5–7 give the method with every formula explained, the Apple Silicon implementation and the experiments. Section 8 tells the story of the project run by run: what each experiment (E0, E1, ...) asked, what changed and what we learned.

2 The idea in plain language

2.1 How a chat model writes

A chat model such as ChatGPT or Qwen writes one *token* (a word or a piece of a word) at a time. To write the 100th token it runs the whole model once, reads what it has written so far and picks the next token. A 100-token answer therefore needs 100 full runs of the model, one after the other. At small batch sizes, autoregressive decoding often spends much of its time moving model weights from memory. The balance depends on the model, hardware, cache and implementation; these experiments measure latency, not memory-bandwidth utilisation.

2.2 How next-concept modeling writes

NCM (Kumbong et al. 2026) writes differently. It first produces a single, very rough “concept” for the whole text, then refines it in a few rounds. Each round adds roughly twice as much detail, and **every piece of detail in a round is produced at the same time**. Figure 1 shows a real example from this project: a 32-character sentence written in six rounds (levels). After level 0 the rough shape is there (“eue likes one oed egg.”); by level 2 the words are settled; later levels only change single letters.

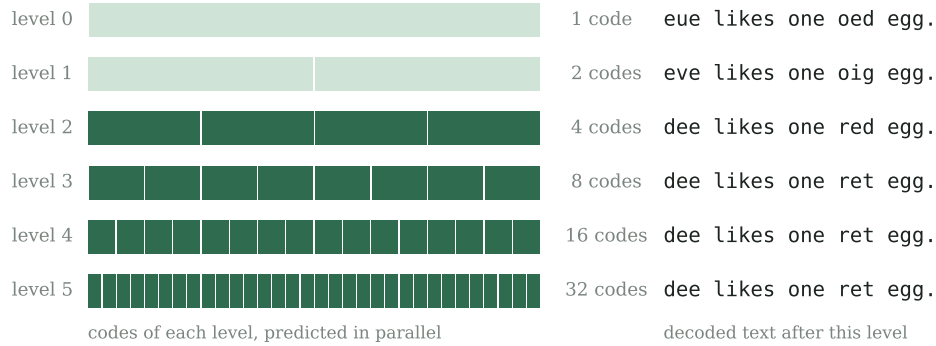


Figure 1: A real generation from experiment E1. Left: the codes of each level; all codes in one row come from one model run. Right: the text obtained by decoding everything up to that level. Six model runs write the sentence; a word-by-word model needs 32. The sample also shows a failure: level 3 turns “red” into “ret”, and nothing later repairs it.

The number of rounds is fixed by the model configuration (six in TinyWorld, twelve in the original paper’s Sudoku model), so NCM needs far fewer model runs than a word-by-word model. That is where its speed comes from.

2.3 Two parts: a codec and a generator

NCM has two learned parts. An analogy from image files helps.

- The **codec** is like an image format with progressive loading: a JPEG that first shows a blurry picture, then sharpens. It learns to turn a sentence into a stack of codes, from one coarse code to one code per letter, and to turn the codes back into the sentence. It is trained first, on its own, with reconstruction losses. Full-hierarchy reconstruction is measured separately from generation quality.
- The **generator** is the creative part. It learns to *invent* new code stacks one level at a time: level 0, then level 1 given level 0, and so on. The codec then turns the finished stack into text.

2.4 Why it matters, and the catch

Fewer, wider model runs suit a laptop GPU: one run reads the weights once and does a lot of arithmetic with them. The catch is that all codes of a level are chosen at the same moment, so neighbouring pieces cannot see each other’s choice. If two pieces of a level disagree, the result can be wrong in a way a word-by-word model would avoid. Much of this paper measures when that matters.

2.5 What we did

We rebuilt NCM from scratch for Apple’s MLX framework (Hannun et al. 2023), tested it on a small invented language and on Sudoku puzzles (the original paper’s showcase), and compared it with a character-level AR model of similar generator size and with a large chat model. The speed advantage holds on a laptop. On Sudoku, two training seeds solve 97.25–98.10% of easy boards but only 33.25–40.30% of hard boards. Original NCM reports 99.5% and 86.1%, respectively, under a different setup. A historical comparison matches generator presentations against our AR baseline; Section 8 records the experiments. NCM is a generation method used here for two tasks, not a Sudoku-specific expansion of the term or an instruction-following language model.

3 Glossary

Table 1: Terms used in this paper.

Term	Meaning
Token	The unit a model reads and writes. Here: one character (TinyWorld) or one Sudoku cell.
Autoregressive (AR) model	A model that writes one token at a time, each conditioned on all previous ones. Chat models are AR models.
Forward pass, NFE	One run of a model on its input. NFE (number of function evaluations) counts forward passes per generated sample.
Parameters	The learned numbers inside a model; “6.4M parameters” means 6.4 million.
Embedding, latent vector	A list of numbers (128–512 here) that represents a token or a position inside the model.
Vector quantisation (VQ)	Replacing a vector by the nearest entry of a fixed list, the <i>codebook</i> , so it can be stored as one integer, the <i>code</i> .
Codebook, code	The list of allowed vectors at one level (16 or 128 entries here) and the index of one entry.
Level	One round of refinement. Level k has l_k codes; level 0 is the coarsest.
Residual	What is left over: the difference between the true vector and the approximation built so far.
Codec (tokenizer, VQ-VAE)	Stage 1 of NCM: encoder, quantiser and decoder that turn text into codes and back.
Generator	Stage 2 of NCM: the transformer that predicts the codes of each level.
Transformer, attention, mask	The standard neural network for sequences. <i>Attention</i> lets each position read information from other positions; a <i>mask</i> says which positions may be read.
Teacher forcing	Training on the <i>true</i> previous outputs instead of the model’s own. Fast, but the model never practises recovering from its own mistakes.
Conditioning corruption	Replacing some true inputs with wrong ones during training so the model learns to cope with its own mistakes.
Cross-entropy (CE), nats	The training loss: how surprised the model is by the right answer, in natural-log units. Lower is better; $\ln 16 \approx 2.77$ nats is a uniform guess among 16 codes.

Term	Meaning
Perplexity	e^{CE} , the effective number of choices. Code perplexity 1 means a level always uses the same code and carries no information.
Temperature, greedy	Sampling randomness. Temperature 0 (greedy) always takes the most likely choice.
Batch, latency, p50	A batch is several samples generated together; latency is the time for one batch; p50 is the median over repeated runs.
KV cache	A store of past attention values that speeds up AR generation.
EMA	Exponential moving average: a running average that slowly forgets old values. Used to update codebooks.
Straight-through estimator	A trick to train through a step that has no gradient (like picking the nearest code) by treating it, for the gradient only, as the identity.
MLX, Metal	Apple’s machine-learning array framework, and the GPU interface it runs on.
fp32, bf16	32-bit and 16-bit (bfloat16) floating-point formats.
E0, E1, ...	Experiment numbers, in the order the experiments were run (Section 8).

4 Background

Residual vector quantisation. A vector quantiser maps a continuous vector to the nearest of V codebook entries (Oord et al. 2017). Residual quantisation applies several quantisers in sequence, each to what the previous ones left over (Lee et al. 2022; Zeghidour et al. 2021); it underlies modern neural audio codecs.

Next-scale prediction. VAR (Tian et al. 2024) combines residual quantisation with resolution: the first quantiser sees a 1×1 summary of an image, later ones 2×2 , 3×3 , up to full resolution, and a transformer generates one scale per step. NCM applies the idea to 1-D token sequences (Kumbong et al. 2026).

Diffusion language models refine positions of a sequence over T denoising steps (Sahoo et al. 2024). Full-sequence samplers process length L at each step; NCM’s early levels process fewer code positions, although prefix and decoder work also contribute. The NCM paper reports quality at or above strong diffusion language models with 10–31× faster generation under its H100 benchmark protocol (Kumbong et al. 2026).

5 Method

In short. The codec learns to describe a sequence with a stack of codes, coarse to fine. The generator learns to predict each level’s codes from the levels above it. Generation runs the generator once per level, then decodes.

We follow the NCM design (Kumbong et al. 2026) and its released Sudoku configuration; Table 2 lists where we differ. A running toy example uses a sequence of $L = 8$ tokens, levels $l = (1, 2, 4, 8)$ and vectors of width $C = 2$.

5.1 Codec: from tokens to a stack of codes

Encoder. Each token x_i and its position i are looked up in two tables, added and normalised:

$$z_i = \text{Norm} (W_{\text{tok}}[x_i] + W_{\text{pos}}[i]), \quad z \in \mathbb{R}^{L \times C}.$$

Reading it: W_{tok} has one learned row per possible token and W_{pos} one per position; z is the whole sequence as L vectors of C numbers. “Norm” rescales each vector (LayerNorm), or groups of channels across the sequence (GroupNorm, as upstream), without a learned scale, because the quantiser compares squared distances and a learned scale could grow without bound. In the toy example z is an 8×2 table of numbers.

Quantisation, level by level. The quantiser keeps a *residual* r , initially all of z , and for each level $k = 0, \dots, K - 1$ does four things:

$$\begin{aligned} & \underbrace{x_k = D_k r_k}_{1. \text{ shrink}}, \quad \underbrace{c_k = \arg \min_j \|x_k - E_k[j]\|^2}_{2. \text{ pick nearest codes}}, \\ & \underbrace{q_k = \phi_k (U_k E_k[c_k])}_{3. \text{ expand and refine}}, \quad \underbrace{r_{k+1} = r_k - \text{sg}(q_k)}_{4. \text{ subtract}}. \end{aligned}$$

Reading it:

1. **Shrink.** D_k averages the residual down to l_k positions (area pooling). Toy example: for level 1 ($l_1 = 2$), positions 1–4 are averaged into one vector and 5–8 into another.
2. **Pick.** For each of the l_k shrunk vectors, find the nearest of the V codebook entries $E_k[1..V]$; its index is the code. c_k is a list of l_k integers, for example (5, 12).
3. **Expand and refine.** Look the codes back up, stretch them to L positions by linear interpolation (U_k), and apply $\phi_k(h) = (1 - \rho)h + \rho \text{conv}_k(h)$: a small learned convolution blended half and half ($\rho = 0.5$) with its input, which smooths the blocky stretched result.
4. **Subtract.** Remove what this level explained, so the next level only describes what is left. sg (stop-gradient) means the subtraction is not differentiated.

The reconstruction after level k is the sum of everything explained so far:

$$\hat{z}_k = \sum_{j \leq k} q_j, \quad \hat{z} = \hat{z}_{K-1} \approx z.$$

Decoding $\hat{z}_k$ for $k < K - 1$ gives the “after level k ” texts of Figure 1.

Codebook updates. Codebooks are not trained by gradients. After each step every entry moves towards the average of the vectors assigned to it:

$$N_j \leftarrow \gamma N_j + (1 - \gamma) n_j, \quad m_j \leftarrow \gamma m_j + (1 - \gamma) \sum_{i: c_i=j} x_i, \quad E[j] = m_j / \tilde{N}_j.$$

Reading it: n_j counts how many vectors picked entry j in this batch, N_j is its running average with decay $\gamma = 0.99$, m_j the running sum of those vectors, and $\tilde{N}_j$ is N_j with a tiny amount added to every count (Laplace smoothing) so nothing is divided by zero. Entries nobody uses (N_j below a threshold) restart at a random input vector.

Losses. The decoder, one transformer block, reads $\tilde{z} = z + \text{sg}(\hat{z} - z)$. Numerically this is $\hat{z}$, but its gradient flows into z as if quantisation were the identity (the straight-through estimator), so the encoder learns. The training loss is

$$\mathcal{L}_{\text{decoder}} = \underbrace{\text{CE}(\text{Dec}(\tilde{z}), x)}_{\text{rebuild the tokens}} + \underbrace{\frac{1}{K} \sum_k \left(\beta \|\text{sg}(\hat{z}_k) - z\|^2 + \|\hat{z}_k - \text{sg}(z)\|^2 \right)}_{\text{VQ loss}}, \quad \beta = 0.25.$$

Reading it: the first term asks the decoder to rebuild the exact tokens. The *commitment* term (β) pulls the encoder output towards what the codes can express; the *fit* term trains the refinement convolutions ϕ_k so each partial sum $\hat{z}_k$ is as close to z as possible.

Fine-level dropout. Half of the time the decoder receives $\hat{z}_k$ for a random $k < K - 1$ instead of the full $\hat{z}$. It must then make sense of a coarse-only description, which is why every prefix of levels decodes to a plausible sequence: a form of nested dropout (Rippel et al. 2014).

Learned level 0. Averaging an entire Sudoku grid into one vector gives nearly the same vector for every grid, so level 0 carries no information (Section 7.2). Upstream replaces D_0 and U_0 by a strided convolution and a transposed convolution initialised to behave like pooling and repetition. Picking a code has no gradient, so the gradient reaches them through a straight-through estimator: q_0 is built from $x_0 + \text{sg}(E_0[c_0] - x_0)$, whose value is the code vector and whose gradient is that of x_0 .

5.2 Generator: predicting one level from the levels above

The generator is a transformer that reads one long row of vectors:

$$[\text{prefix } (P) \mid \text{level-0 queries } (l_0) \mid \text{input}_1 (l_1) \mid \dots \mid \text{input}_{K-1} (l_{K-1})], \quad \text{input}_k = D_k \hat{z}_{k-1}.$$

Reading it: the *prefix* holds the conditioning, for Sudoku the 81 cells of the puzzle (blanks as 0). Level 0 has nothing above it, so it reads learned query vectors. Every other level reads the reconstruction from all coarser levels, shrunk to its own length. Toy example: input_2 is $\hat{z}_1$ (levels 0 and 1 combined) averaged down to 4 positions. Head k turns each of the l_k positions of its block into a probability for each of the V codes of level k .

Training loss. With the true codes c_k of a training example,

$$\mathcal{L}_{\text{gen}} = \frac{\sum_k l_k \cdot \text{CE}_k}{\sum_k l_k}, \quad \text{CE}_k = -\frac{1}{l_k} \sum_{i=1}^{l_k} \log p_{\theta}(c_{k,i} \mid \text{input}_k, \text{prefix}).$$

Reading it: CE_k is the average surprise at the true codes of level k ; weighting by l_k makes every code count equally (the paper’s $\alpha = 0$ setting (Kumbong et al. 2026, App. F.2.3)). A uniform guess among 16 codes scores $\ln 16 \approx 2.77$.

Attention pattern. Positions of level k may read the prefix and the other positions of level k , nothing else (*block-diagonal* attention); coarser levels reach level k only through input_k . Two consequences follow. Training on the whole row at once computes exactly what level-by-level generation computes (tested, Section 6.2). And each level’s codes are chosen without seeing each other’s choices, the source of both the speed and errors like “red” $\rightarrow$ “ret”. RoPE (Su et al. 2021) positions restart at zero in every level, a learned level embedding tells levels apart, and queries and keys are RMS-normalised.

Conditioning corruption. In training the generator reads true coarser codes; in generation it reads its own, sometimes wrong, codes. Upstream narrows this gap by replacing a fraction p_k of the conditioning codes of level k with random codes of the same level, keeping the true codes as targets, with p_k falling linearly across levels (Kumbong et al. 2026, Sec. 3.4):

$$p_k = p_{\text{first}} + (p_{\text{last}} - p_{\text{first}}) \frac{k}{K-2}, \quad k = 0, \dots, K-2.$$

This is the coarse-to-fine analogue of scheduled sampling (Bengio et al. 2015). The paper notes that corruption “based on the model’s actual errors may better match inference conditions” but does not test it; experiment E9 tests it and finds no gain at our scale (Section 8).

5.3 Generation

For $k = 0, \dots, K-1$: run the generator on input_k and the prefix, sample all l_k codes of level k at once, add their contribution q_k to the reconstruction and shrink it into input_{k+1} . Finally decode $\hat{z}_{K-1}$ to tokens. Cost: K generator passes and one decoder pass, independent of how long each level is.

Table 2: Initial E6 settings versus released Original NCM Sudoku settings (Kumbong et al. 2026, Tables 5–6). Later runs change exposure and corruption; see Appendix A.

Component	Original NCM (Kumbong et al.)	MLX starting recipe (E6)
Tokens per board	89 (cells and separators)	81 (cells)
Levels $\times$ codes	12×16 , $l_{K-1} = 89$	12×16 , $l_{K-1} = 81$

Component	Original NCM (Kumbong et al.)	MLX starting recipe (E6)
Level-0 sampler	learned (one layer, as 89 is prime)	learned from E6, one layer
Codec training	10,000 updates $\times$ batch 1,024, lr $5e-4$	$3,000 \times 128$, lr $2e-4$
Generator	8×512 (25.5M), 50,000 updates $\times$ 1,024, lr $3e-4$	8×256 (6.4M), $4,000 \times 128$, lr $6e-4$
Conditioning corruption	yes, linear schedule	from E8 ($0.30 \rightarrow 0.05$)
Models	one per difficulty	one for all difficulties

6 Implementation on Apple Silicon

In short. Pure MLX and NumPy, no code copied from upstream. Two findings matter beyond this project: Apple GPUs compute fp32 matrix products at reduced precision, and block-diagonal attention is faster computed per level than through a mask.

6.1 Framework mapping

Codebooks and their running averages are frozen MLX arrays updated outside automatic differentiation; the shrink and stretch operators D_k , U_k are precomputed matrices; attention uses MLX’s fused `scaled_dot_product_attention`. Weights are stored as safetensors. The correspondence to upstream is documented component by component in `docs/research/upstream-map.md`.

6.2 Exactness and Metal precision

A test asserts that level-by-level generation produces the same logits as the training forward. On the Metal GPU the two disagreed by about 5×10^{-4} . Against a float64 NumPy reference, the same fp32 computation had an error of 1.4×10^{-2} on the GPU and 2×10^{-6} on the CPU: fp32 matrix products on Metal run at reduced internal precision, about 10^{-3} relative. Exactness tests therefore run on the CPU, and a separate test checks the GPU path within 5×10^{-3} . This matters for anyone writing equivalence tests for MLX models; sampling quality is evaluated separately, and these exactness checks do not establish the quality effect of precision.

6.3 Segmented block-diagonal attention

With a mask, attention over the full Sudoku row forms a 395×395 score matrix per head, most of it masked out. Computing attention separately for the prefix and for each level (keys and values: prefix plus that level) gives the same result with $P^2 + \sum_k l_k(P + l_k) \approx 48,000$ scores instead of 156,000. In training (width 256, batch 128, compiled) step time fell from 1.59 s to 1.18 s and peak memory from 18.7 GB to 16.4 GB, measured back to back. A test checks equality with the masked computation.

6.4 Compilation and precision

Compiling the training step with `mx.compile` cut step time by about $1.3\times$ ($1.22\text{ s} \rightarrow 0.92\text{ s}$ at width 256, batch 128). Casting the generator to bfloat16 gave a further $1.4\text{--}1.8\times$ in a first test but saved no memory, because the frozen fp32 codec promotes the generator’s inputs back to fp32; mixed precision is not used in the experiments reported here.

6.5 Interrupted training and reproducibility

Long runs need recoverable training state as well as model weights. Codec and generator snapshots now store optimizer moments and step, random state, shuffled-batch position, elapsed training time and model parameters; codec snapshots include VQ running averages and generator snapshots retain the best-validation model. Files are replaced atomically, with a preceding snapshot retained. A pause signal saves after the current step; a hard crash recovers from the last completed snapshot. Short CPU tests give identical saved state for uninterrupted

and resumed training, including the compiled generator path. This does not establish equality across hardware or backends. Rented-pod recovery also depends on copying snapshots outside the container before it disappears.

E17 exposed the distinction: closing its terminal killed training after the last saved step, 30,000 of 32,000. That legacy checkpoint contained weights alone. Its continuation retains the learning-rate and batch positions but resets optimizer moments and random state, so it must be reported as a weights-only recovery. The recovered model solves 71.25 / 31.75 / 3.70% of easy/medium/hard boards, close to E13, after 2,000 continuation steps taking 1,247.53 s. This is consistent with puzzle construction affecting transfer, but the reset prevents a controlled causal conclusion. The operational procedure is documented in `docs/RECOVERY.md`.

7 Experiments

In short. On a small invented language NCM is faster and somewhat less accurate than an AR model with similar generator size. On Sudoku our AR result is near the Original AR baseline; our NCM does not match Original NCM’s reported result.

Runs use an Apple M5 Pro (64 GB unified memory, macOS 26.5), an RTX 3080 or a rented A40, as labelled for each experiment, with MLX 0.32.3. Other GPU workloads often ran on the Mac; timings this affects are marked.

7.1 TinyWorld: an invented language

TinyWorld has 15,360 sentences of the form *name verb count adjective noun*, with number agreement (“one red cat”, “two red cats”), at most 32 characters. 12,000 sentences train the models; the rest are held out. A sentence is *valid* if it parses and agrees. The codec (6 levels, $l = 1, 2, 4, 8, 16, 32, 128$ codes each, 1.3M parameters) rebuilds 100% of held-out sentences exactly. The NCM generator (5.0M parameters) is compared with a character-level AR transformer of the same width and depth (4.8M parameters) with a KV cache.

Table 3: TinyWorld, NCM and AR with similar generator size (E1); NCM also retains its codec.

	MLX implementation	MLX AR baseline
Valid, temperature 1.0	89.8%	98.6%
Valid, temperature 0.7	95.8%	99.8%
Valid and not in the training set, temperature 1.0	16.3%	23.5%
Forward passes per sequence	6 + 1 decode	32
Median latency, batch 1 / 16 / 256	4.5 / 5.8 / 37.8 ms	30.5 / 22.9 / 49.8 ms

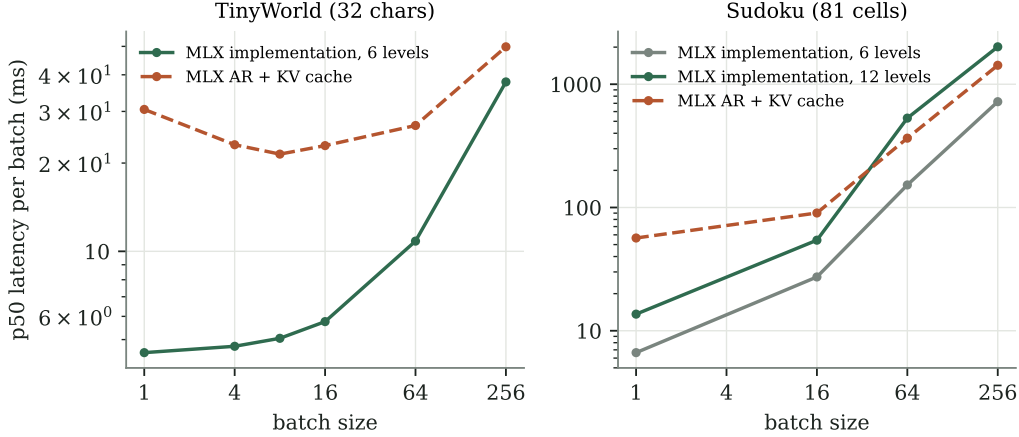


Figure 2: Median latency per batch for NCM and AR models. Left: TinyWorld. Right: Sudoku. Both axes logarithmic.

NCM is less accurate and faster. Its errors are local spelling slips; agreement, a sentence-wide rule, holds in 99.8% of well-formed samples. Forcing the true codes for the first k levels and sampling the rest (E2) shows that errors arise at every level, not only the finest. Letting level k also read coarser levels (block-causal attention, E3) raised validity at temperature 1 from 89.8% to 92.2% ($n = 1,024$, standard error about 0.9 points) at 1.5–1.9 $\times$ the latency.

7.2 Sudoku

Comparison labels. “MLX implementation” means the independently trained NCM model in this repository. “Original NCM” means Kumbong et al.’s published model and checkpoint results (Kumbong et al. 2026). The “MLX AR baseline” is trained here; the “Original AR baseline” is quoted from their study. The underlying method is NCM in both implementations.

Each board is 81 digits; the puzzle (blanks as 0) is the prefix and the solution is the target. Training uses 200,000 synthetic puzzles from randomised backtracking with 28–45 givens. Test sets are the 2,000 easy, medium and hard boards (40, 35 and 30 givens) published with the NCM paper. A board counts as *solved* if the output is a valid grid that keeps every given. Decoding is greedy.

Table 4: Sudoku, share of boards solved. Params counts generators only for NCM; the codec is additional. Paper rows are reference results from (Kumbong et al. 2026, Table 3). Budgets differ except for the explicitly matched E11 comparison.

Model	Params	Easy	Medium	Hard
MLX implementation, E4–E6	6.4M	0.0%	0.0%	0.0%
MLX implementation, E8 (+ corruption)	6.4M	6.2%	0.35%	0.0%
MLX implementation, E11 (+ 4 $\times$ data)	6.4M	55.2%	18.1%	1.6%
MLX implementation, E13 (+ stronger corruption)	6.4M	70.9%	31.2%	4.0%

Model	Params	Easy	Medium	Hard
MLX implementation, E14 (+ 16× data)	6.4M	87.4%	53.4%	11.2%
MLX implementation, E15	6.4M	96.25%	76.65%	31.90%
MLX implementation, E18 (base data)	6.4M	84.25%	49.10%	10.55%
MLX implementation, E19 pilot	6.4M	90.20%	59.75%	16.55%
MLX implementation, E19 funded (seed 0)	6.4M	98.10%	83.40%	40.30%
MLX implementation, E20 (seed 1)	6.4M	97.25%	79.60%	33.25%
MLX AR baseline	6.3M	13.8%	3.0%	0.4%
MLX AR baseline, E11's budget (2M sequences)	6.3M	20.5%	4.9%	0.5%
Qwen3.8 27B, 4-bit, no reasoning (50 boards each)	27B	0%	0%	0%
Original AR baseline (Kumbong et al.)	25M	14.6%	5.1%	1.0%
MDLM (reported by Kumbong et al.)	29M	92.0%	77.1%	30.2%
Original NCM (Kumbong et al.)	25M	99.5%	96.5%	86.1%

The first AR row is within a few percentage points of the Original AR baseline, but this numerical proximity does not establish a reproduction of its model or training recipe. Without conditioning corruption (E4–E6) our NCM solves nothing; with it (E8) it solves 6.2% of easy boards and fills 74.7% of blank cells correctly, more than the AR model's 71.0%, while still solving fewer boards. E11 solves 55.2% of easy boards; at the same 2.048M generator presentations, the AR model solves 20.5%, 4.9% and 0.5%. This matches generator data exposure, not total training cost, FLOPs or system size. Section 8 records the changes.

The shared E6 codec contains 13,816,704 gradient-trained values, 49,152 EMA-trained code-book values and 49,344 EMA accumulators: 13,915,200 saved scalar values in total. Its learned first-level down/up convolutions account for 10,617,344 trainable values. The 6,440,128 generator values are additional; generator-size comparisons exclude the codec. Older rows preserve historical rounding; E15 and the newly completed runs retain their two-decimal solved rates. Each test set has 2,000 boards. Most recipes use one training seed; E19-funded and E20 provide two seeds of the same scientific recipe. Repeated sampling seeds do not quantify training variability.

Historical synchronized timings use temperature 1.0 while Sudoku quality uses greedy temperature 0.0. Both samplers remain uncompiled; `mx.compile` is used only for generator training when configured. Latency therefore does not directly time the solved-board decoding setting. The

benchmark includes the output host copy, prefix work and final decoder, and records shared-device contention; cross-hardware timing ratios are not controlled comparisons.

Smaller-generator result. Easy-board accuracy is within 1.40 percentage points of Original NCM (98.10% versus 99.50%) with a 6.44M versus 25.46M generator: 3.95× smaller and 3.125× fewer generator presentations. This is a useful small-model result, not overall task parity: medium and hard remain 13.10 and 45.80 points behind. The separate codec is excluded from this size ratio, and the recipes and representation differ. E20 retains the same generator size and exposure but its easy score is 2.25 points below Original NCM; its medium/hard gaps are 16.90/52.85 points. The best-seed proximity on easy boards is not a reproducibility guarantee.

7.2.1 Completed E18 and E19 experiments

E18 tests stronger corruption on the base 200,000-puzzle corpus at E15’s 8.192M-presentation budget. Its 84.25 / 49.10 / 10.55% solve rates remain below both E14 and E15. Stronger corruption is therefore not a universal improvement, and this negative result remains part of the evidence.

The short E19 pilot uses the frozen E6 codec and 8M cached unique puzzles in addition to the base corpus. It completes 23,895 updates at batch 128 (3,058,560 presentations), solving 90.20 / 59.75 / 16.55%. E19-funded starts a fresh generator with the same corpus and a separate 128,000-update cosine schedule. At 16,384,000 presentations it solves 98.10 / 83.40 / 40.30%, improving on E15 by 1.85 / 6.75 / 8.40 percentage points. The exposure doubles while batch size, corpus and schedule differ. These comparisons establish outcomes of whole recipes, not a controlled effect of any one change.

Table 5: Completed training and checkpoint selection (training-loop seconds).

Run	Updates × batch	Presentations	Best step	Train seconds
E18	128,000 × 64	8,192,000	128,000	23,366.69
E19 pilot	23,895 × 128	3,058,560	23,895	10,225.84
E19 funded	128,000 × 128	16,384,000	127,000	54,419.81

The funded run completes its full schedule without a time-limit stop or optimizer reset; its final resumable snapshot is step 128,000, while held-out evaluation uses the best-validation step 127,000. Best weighted validation CE is 0.133244. The complete remote pipeline takes 55,700.91 s, including preprocessing, evaluation, benchmarking and transfer but excluding rental startup and the final cleanup wait. All 17 downloaded report/artifact files match their completion hashes. Source revision is 80dd445e5d17f54f53687a88459e011de4f9006f; the generator SHA-256 and full result extract are retained in the repository’s E19 report. Full-state snapshots preserve optimizer and random state; this uninterrupted run is not a test of cross-backend exact recovery.

Funded validation solves 70.90% of 1,000 boards. On easy/medium/hard tests, blank-cell accuracy is 99.69 / 97.12 / 86.99% and givens preservation is 100 / 100 / 99.60%; full-depth codec validation reconstruction is separately 100%. Wilson 95% intervals for the solved rates are 97.40–98.61 / 81.71–84.97 / 38.17–42.47%. These assume independent test boards, exclude training-seed variability and do not establish paired significance against prior runs. Per-board prediction archives were not retained. E20 supplies a second training seed below.

7.2.2 E20: a second training seed

E20 repeats the E19-funded scientific recipe with configured training seed 1 instead of 0. Both complete 128,000 updates at batch 128, using the frozen E6 codec, 200,000 base puzzles plus 8M cached unique puzzles, corruption 0.50→0.10, learning rate 0.0006 and 1,000-step warmup. Shared codec and corpus input-manifest hashes match. The configured seed affects initialization, minibatch order, dropout and corruption; internal generator streams use that seed plus one. These influences are not separately controlled.

The run name, output path and operational cap also change. E20’s cap is 960 rather than 1,440 minutes; neither run reaches its cap. E20 records no optimizer reset or truncation, and its final full-state snapshot and selected best checkpoint are both step 128,000. Training-loop time is 54,122.43 s; pipeline time is 55,543.62 s under the E19 timing boundary. Best weighted code CE

is 0.142558, compared with 0.133244 for seed 0. Separate physical A40 allocations and missing remote source inventories prevent a strict same-host causal attribution.

Table 6: Two-seed held-out aggregates. Each seed reports the same 2,000-board benchmark per difficulty; the mean is descriptive, not a pooled independent sample.

Quantity	Easy	Medium	Hard
Seed 0 solved boards	1,962	1,668	806
Seed 1 solved boards	1,945	1,592	665
Seed 1 solved rate	97.25%	79.60%	33.25%
Seed 1 minus seed 0	−0.85 pp	−3.80 pp	−7.05 pp
Two-seed mean	97.675%	81.500%	36.775%
Seed 1 Wilson 95% interval	96.44–97.88%	77.78–81.31%	31.22–35.35%

Both seeds retain strong easy-board performance and substantially weaker hard performance. E19-funded remains the best observed seed; neither two-seed means nor board-level Wilson intervals reliably estimate the population distribution of training outcomes. The intervals also exclude related-board dependence and repeated benchmark-guided recipe selection. Without per-board predictions, paired significance and agreement between the solved-board sets cannot be computed. No additional model evaluation is included in this version.

E20 validation solves 71.00%, compared with 70.90% for E19-funded, while hard blank-cell accuracy falls from 86.99% to 84.92%. Similar validation solving therefore does not imply similar external hard-set performance. The saved hard aggregates identify 670 valid grids, of which 665 preserve the givens: 1,330 failures are invalid grids and five additional failures are valid grids contradicting clues. Fourteen boards violate givens overall. Clue preservation alone cannot repair the dominant invalid-grid failure; the aggregates do not identify its internal cause.

Audit boundary. All 15 retained E20 completion files match recorded hashes; inference tensors equal the saved best-model tensors, and optimizer/training steps agree at 128,000. Generator SHA-256 is b6ddadb7278e7f1c5542eecef676840f39cb829e802d189b65f2dadd645c2b3d; the declared source is 4ad3646189197fe1ca76e45308848793fefe9a3f. These checks establish retained-byte integrity and internal consistency, not independent reproduction. No checkpoint-bound ordered scoring manifest, per-board predictions or remote source digest inventory was retained. Current local board files match the published hashes, but they were absent from the remote transfer-verification manifest. Actual remote evaluation identity, complete duplicate-free coverage and exact source bytes remain independently unverified. Reported outcomes are historical aggregates with these limits.

E20’s recorded synchronized benchmark uses temperature 1, three warmups and 20 iterations. Batch 1/16/64/256 p50 is 29.453/48.506/174.326/581.791 ms; p95 is 30.323/49.274/175.352/583.397 ms. The CUDA GPU-load log is empty because the wrapper used a Metal-only probe. Contention at benchmark time is unknown, and these timings neither measure greedy quality generation nor establish a seed-dependent speed effect.

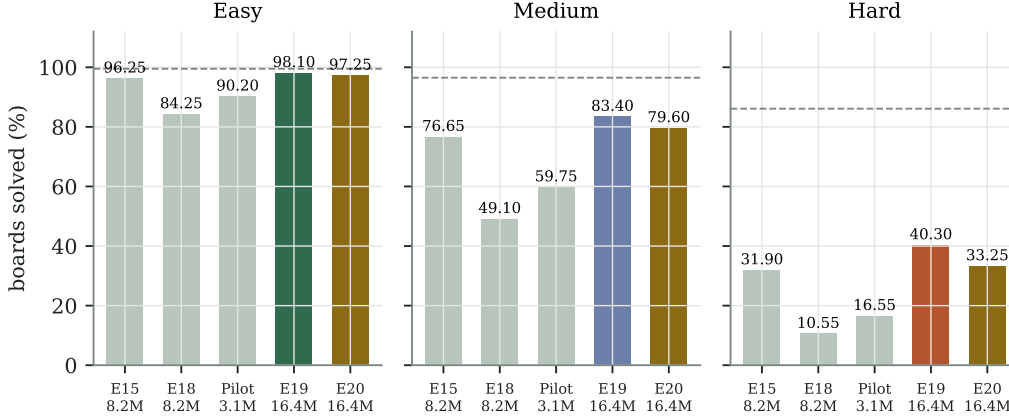


Figure 3: Completed MLX implementation recipes, with rounded generator presentations beneath each run; exact exposures are in the tables. Each bar reports 2,000 held-out boards; retained aggregate and provenance limitations apply. Dashed lines show Original NCM (Kumbong et al.), with its larger generator and 51.2M presentations; they are not matched-budget targets.

Funded synchronized generation at temperature 1.0 uses three warmups and 20 timed iterations, 12 generator forwards and completion through a host copy. Batch 1/16/64/256 p50 latency is 26.477 / 47.905 / 174.279 / 583.670 ms; p95 is 26.690 / 48.308 / 174.552 / 584.715 ms. This timing does not measure greedy quality decoding. Preflight records zero GPU utilization, zero allocated memory and no compute applications; an immediate pre-benchmark contention sample is not retained, so these are not certified idle-device limits. Neither cross-host A40 timings nor RTX 3080 comparisons isolate a hardware speed ratio.

The first funded allocation fails during shell preflight, before any training step, and is deleted. The failure is SIGPIPE from an early-closing version-check pipeline under `pipefail`; three regression cases reproduce it, and the corrected fresh attempt passes all 78 tests. Image startup timeouts recover before readiness. These are setup incidents, not extra training runs or reset-state continuations.

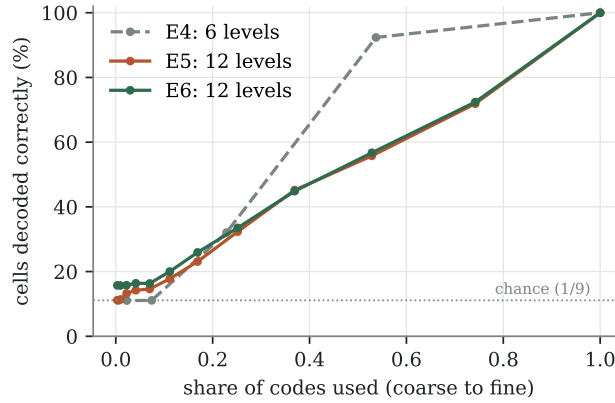


Figure 4: Cell accuracy when decoding from the first k levels of the codec, against the share of codes used. E4 stores almost all information in one level; E5 and E6 refine gradually.

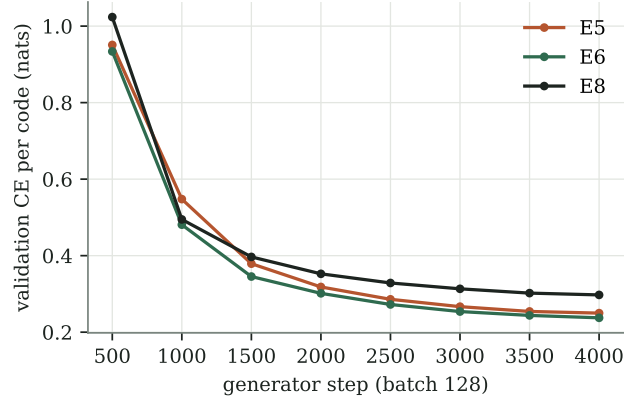


Figure 5: Validation cross-entropy per code during generator training.

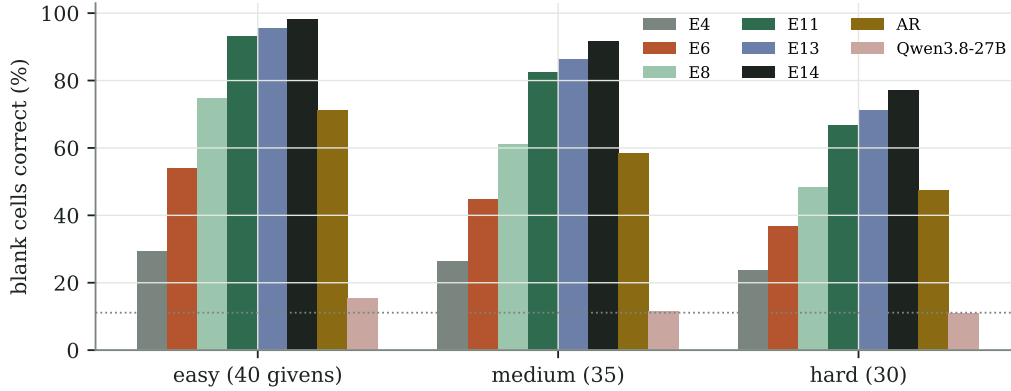


Figure 6: Blank cells filled with the reference digit, per test set. Chance is about 11%.

Why per-cell gains do not become solved boards. A board is about 300 codes. Teacher-forced accuracy per level in E6 ranges from 25% (level 0, which the puzzle barely determines) to 99.6% (the last level); generating greedily, an error at an early level changes the inputs of every later level. E8’s corruption training attacks exactly this compounding. Even 98% accuracy per cell, with independent errors, would solve only $0.98^{41} \approx 44\%$ of easy boards, which have 41 blanks. The paper’s generator was trained on about 51 million sequences; E6 saw 0.5 million.

7.3 A large chat model as reference

Qwen3.8 27B (4-bit, served locally) answered each board in one request at temperature 0 with reasoning disabled. It solved none of 150 boards; its blank-cell accuracy was 11–15%, near chance, and about half its replies were not a 9×9 grid. With reasoning enabled and up to 6,000 tokens per board it did not finish ten boards in 95 minutes on the shared GPU. This is a reference point, not a matched comparison: the model is about 4,000 times larger and was not trained on the task.

8 The journey: experiment by experiment

In short. The experiments examine architecture, training and data choices; some combine multiple changes. Experiment numbers (E0, E1, ...) identify runs; the full record with every number is in `docs/EXPERIMENTS.md`.

Table 7: Experiments in order. Sudoku results are for the easy test boards.

Run	Question	Change from the previous run	Result	What we learned
E0	Does the pipeline work end to end?	First version, 4,000 TinyWorld sentences	94.6% valid; overfits	The codec rebuilds 99.4% of sentences; models need more data
E1	NCM vs AR, fairly	12,000 sentences, best-validation checkpoints	NCM 89.8% vs AR 98.6% valid; 4–5× faster	Speed holds on a laptop; the accuracy gap is real
E2	Which level causes errors?	Force true codes for levels 0..k	Errors at every level	Not only the finest level is at fault
E3	Does reading coarser levels help?	Block-causal attention	92.2% valid, 1.5–1.9× slower	Small gain; kept as an option
E4	Does NCM solve Sudoku?	Puzzle as prefix, 6 levels	0% solved; AR 13.8%	One level holds all information: no real coarse-to-fine
E5	Does upstream’s level schedule fix it?	12 levels × 16 codes, lookup encoder	0%; blank cells 52% (was 29%)	Gradual refinement, but level 0 is dead
E6	Does a learned level 0 fix it?	Learned level-0 sampler, GroupNorm, lower codec lr	0%; blank cells 54%	Level 0 now used; small gains; diverges at lr 1e-3
E7	Upstream’s generator width	8 × 512	stopped early	Recipe search showed corruption was missing; GPU time moved to E8
E8	Does conditioning corruption help?	Uniform corruption 0.30 → 0.05	6.2% solved; blank cells 74.7%	The largest single gain: first solved boards; beats AR on blank cells; validation loss gets worse while generation gets better
E9	Does corruption from the model’s own samples help more?	Replacements drawn from the generator	3.0% solved; blank cells 74.7%	No: same blank cells, fewer solved boards, 1.56× training time

Run	Question	Change from the previous run	Result	What we learned
E10	Upstream’s width	8 × 512, lr 3e-4, 1.5M sequences, on the Mac	55.9% / 18.4% / 2.0%	Width alone does not help at this budget: matches the 256-wide E11
E11	More data	E8 on 2M sequences (4×), RTX 3080	55.2% easy, 18.1% medium, 1.6% hard solved	Data was the largest remaining lever; NCM now beats the same-size AR model
E12	Upstream’s width on a rented GPU	8 × 512, stronger corruption, 3.25M sequences, NVIDIA A40	74.6% / 35.2% / 6.0%	Width adds little; data still dominates
E13	Stronger corruption	0.50 → 0.10 at E11’s budget	70.9% / 31.2% / 4.0%	Corruption strength matters: +16 points on easy
E14	Even more data	E11 on 8.2M sequences (16× E8)	87.4% / 53.4% / 11.2%	Data keeps paying; still not saturated
E16	Fresh unique puzzles vs repetition	E13 on 2M greedy-unique puzzles seen once	57.6% / 21.8% / 2.2%	Worse than E13: how puzzles are built matters more than uniqueness
E17	Unique puzzles built like the test boards	E13 on random-removal puzzles that happen to be unique; weights-only recovery	71.25% / 31.75% / 3.70%	Close to E13; optimizer reset limits interpretation
E15	Both levers + unique-solution puzzles	E14 budget, corruption 0.50 → 0.10, +4M unique puzzles	96.2% / 76.6% / 31.9%	Within 3.3 points on easy; roughly one quarter of upstream’s generator size and one sixth of generator exposure
E18	Both levers, base data only	E14 budget, corruption 0.50 → 0.10	84.25% / 49.10% / 10.55%	Below E14 and E15; stronger corruption alone is not sufficient
E19 pilot	Larger-corpus short schedule	+8M unique puzzles; 3.059M presentations, batch 128, A40	90.20% / 59.75% / 16.55%	Below E15 at less exposure; not a matched comparison
E19 funded	Larger corpus and longer schedule	Fresh initialization; 16.384M presentations, batch 128, A40	98.10% / 83.40% / 40.30%	Best observed seed at twice E15 exposure; still below Original NCM
E20	Does the funded recipe repeat?	Fresh seed 1, same scientific settings and exposure	97.25% / 79.60% / 33.25%	Easy/medium gains persist; hard result varies by 7.05 points; audit limits remain

What the journey shows. The first Sudoku configuration failed for a reason no reconstruction test would reveal: its codec rebuilds boards perfectly yet places nearly all information in one level (the codec-refinement figure, E4), so the generator faces one large guess instead of a series of small ones. The fixes taken from upstream each helped a little and none was enough on its own. Reading the paper’s appendix in full showed the two largest gaps: conditioning corruption and a training budget about a hundred times larger. Corruption alone (E8) produced the first solved boards and lifted blank-cell accuracy by 20 points on every test set. Its validation loss was *worse* than without corruption, a reminder that teacher-forced loss does not measure what generation needs.

9 Discussion

What carries over to a laptop. Measured Apple Silicon runs show lower small-batch NCM latency than an AR model of similar generator size on the tested synthetic task. The advantage shrinks with batch size and code count; historical 12-level Sudoku generation is slower than AR from batch 64 on. These observations do not establish natural-text speed or isolate dispatch, memory traffic and kernel costs. The completed E19/E20 training runs use CUDA, not the Mac.

Partial success at the tested scale. A 6.44M generator learns substantial puzzle-conditioned structure: both funded-recipe seeds solve more than 97% of easy boards and 79% of medium boards. Both exceed E15’s reported rates, with twice its generator exposure and a changed overall recipe. This is sufficient evidence to continue studying the implementation. It is not an estimate of minimum required capacity, a scaling law or a demonstration of human-style logical reasoning. The additional codec prevents treating generator size as whole-system size.

Why hard Sudoku remains unresolved. The hard benchmark provides 30 clues, leaving 51 blanks, versus 40 clues and 41 blanks on easy boards. Every constraint and input clue must be satisfied simultaneously. Errors can be correlated, and a small change in average cell accuracy can accompany a larger change in exact-board accuracy. Neither clue count nor these aggregates reveals which logical dependencies the model fails to represent. The reference comparison also changes capacity, exposure, representation, corpus and difficulty-specific training. Those are candidate explanations, not experimentally established causes. Earlier width-512 trials do not justify promising that widening alone closes the gap.

How to challenge the interpretation. Before further long runs, retain checkpoint-bound input identities, per-board outputs and per-level diagnostics. Use a fresh untouched test set after fixing the selection protocol, since repeated examination of these benchmarks can guide recipe choices. Additional seeds characterize variation; separate random streams and controlled interventions are needed to identify whether initialization, order, dropout or corruption matters. True-code interventions on a diagnostic subset can test error propagation. Matched-data capacity and exposure ablations can test scale without conflating both. Checkpoint selection must use validation data rather than the external benchmark. These are proposed tests, not completed findings.

Decision for the next phase. The project can proceed to controlled text research without declaring the Sudoku reproduction complete. Keep the small generator as a reference while investigating independently converted Original NCM weights. Conversion parity would validate inference compatibility; it would not reproduce training or validate our smaller model. Begin text with a pinned permitted corpus, explicit tokenizer and padding boundaries, and document-separated splits. Report reconstruction and free-running generation separately, include readable failures and an autoregressive comparison with total-cost qualifications. Synthetic TinyStories would be an initial engineering benchmark; naturally occurring text and conditional tasks require further evidence. No instruction-following, summarization, translation or useful application integration follows from Sudoku scores.

10 Limitations

Most generator recipes use at most 6.4M parameters; width-512 ablations use about 25M. E19-funded and E20 provide only two training seeds of one recipe, not a reliable population variance estimate. The best hard score remains 45.80 points below Original NCM; the second seed remains 52.85 points below. No current-budget AR comparison, matched total-cost study or natural-text validation is available. Full-depth codec validation reconstruction does not

prove useful coarse representations under generated errors. Historical aggregate-only scoring, missing remote manifests, repeated benchmark-guided development and unknown benchmark contention limit independent audit and generalization. The original training implementation is not released; unstated recipe details remain uncertain. Existing measurements establish neither production reliability nor a benefit in a user application.

11 Related work

Parallel and coarse-to-fine decoding for text includes discrete diffusion (Sahoo et al. 2024) and the methods compared in (Kumbong et al. 2026). Residual quantisation is standard in audio codecs (Zeghidour et al. 2021) and image generation (Lee et al. 2022; Tian et al. 2024). Scheduled sampling (Bengio et al. 2015) addresses the train–test mismatch that conditioning corruption targets. Bakhaow/ncm-forge adds PyTorch training loops around upstream modules. The repository’s dated related-work comparison records other implementations; we make no priority claim for this MLX implementation.

12 Conclusion

This independent MLX implementation preserves exact CPU training/per-level inference equivalence and demonstrates coarse-to-fine generation on a finite synthetic language and Sudoku. The funded small-generator recipe reports 97.25–98.10% easy, 79.60–83.40% medium and 33.25–40.30% hard solving across two seeds at equal exposure. We regard this as partial implementation and small-model success, with substantial failures on hard boards and unresolved provenance and reproducibility limits. It does not reproduce Original NCM’s scores or establish matched-cost superiority. The observed seed difference warrants better diagnostics rather than attributing the deficit to a single component. The next research phase will test text generation and its quality/cost tradeoff; those capabilities remain unvalidated.

Code and reproducibility

Code, configurations and this paper: github.com/janmus1c/ncm-mlx. Every configuration is in `configs/`; measured runs are logged in `docs/EXPERIMENTS.md`. The benchmark guide explains budget matching, parameter accounting, synchronization and uncertainty.

```
git clone https://github.com/janmus1c/ncm-mlx && cd ncm-mlx
mise run setup                                # Python 3.12, MLX, tests
scripts/run_experiment.sh configs/sudoku-e6.json # train, evaluate, benchmark
.venv/bin/python paper/make_figures.py --plot   # rebuild figures from committed data
paper/build.sh                                  # rebuild this paper
```

AI assistance

OpenAI Codex assisted with implementation, experiments, analysis, figures, documentation and manuscript drafting and revision. It is not an author. This assistance does not constitute independent verification of the results or human review of the manuscript.

References

- Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. ‘Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks’. *Advances in Neural Information Processing Systems*.
- Hannun, Awni, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. *MLX: Efficient and Flexible Machine Learning on Apple Silicon*. <https://github.com/ml-explore/mlx>.
- Kumbong, Hermann, Ishan S. Khare, Iddah Mlauzi, et al. 2026. ‘Next-Concept Modeling: Fast, Parallel Coarse-to-Fine Text Generation’. *arXiv Preprint arXiv:2610.14162*.
- Lee, Doyup, Chiheon Kim, Saehoon Kim, Minsu Cho, and Wook-Shin Han. 2022. ‘Autoregressive Image Generation Using Residual Quantization’. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

- Oord, Aaron van den, Oriol Vinyals, and Koray Kavukcuoglu. 2017. ‘Neural Discrete Representation Learning’. *Advances in Neural Information Processing Systems*.
- Rippel, Oren, Michael Gelbart, and Ryan Adams. 2014. ‘Learning Ordered Representations with Nested Dropout’. *International Conference on Machine Learning*.
- Sahoo, Subham Sekhar, Marianne Arriola, Yair Schiff, et al. 2024. ‘Simple and Effective Masked Diffusion Language Models’. *Advances in Neural Information Processing Systems*.
- Su, Jianlin, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. ‘RoFormer: Enhanced Transformer with Rotary Position Embedding’. *arXiv Preprint arXiv:2104.09864*.
- Tian, Keyu, Yi Jiang, Zehuan Yuan, Bingyue Peng, and Liwei Wang. 2024. ‘Visual Autoregressive Modeling: Scalable Image Generation via Next-Scale Prediction’. *Advances in Neural Information Processing Systems*.
- Zeghidour, Neil, Alejandro Luebs, Ahmed Omran, Jan Skoglund, and Marco Tagliasacchi. 2021. ‘SoundStream: An End-to-End Neural Audio Codec’. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*.

Appendix A: Hyperparameters

Table 8: Main settings per experiment. All runs: AdamW, weight decay 0.01, cosine schedule with warmup, batch 128, commitment $\beta = 0.25$, fine-level dropout 0.5.

Run	Levels	Codes	Codec	Codec steps (lr)	Generator	Gen. steps (lr)	Other
E1	1,2,4,8,16,32	128	128 wide, 2+2 blocks	3,000 (1e-3)	6×256	3,000 (6e-4)	EMA 0.95; best step 1,500
E4	1,3,9,27,54,81,32		2 encoder blocks	3,000 (1e-3)	8×256	5,000 (6e-4)	prefix 81
E5	1,2,4,6,9,13,18,26,37,50,67,81		lookup encoder, 1 decoder block, 256 wide	3,000 (1e-3)	8×256	4,000 (6e-4)	EMA 0.99
E6	as E5	16	E5 + learned level 0, GroupNorm(8), dropout 0.3	3,000 (2e-4)	8×256	4,000 (6e-4)	
E8	as E5	16	E6 codec	—	8×256	4,000 (6e-4)	corruption 0.30 $\rightarrow$ 0.05, compiled
E9	as E5	16	E6 codec	—	8×256	4,000 (6e-4)	as E8, model-sampled

Later Sudoku runs reuse the frozen E6 codec and its 12-level, 16-code hierarchy. All use an 81-cell prefix, eight attention heads, configured generator seed 0 except E20 (seed 1), AdamW with weight decay 0.01, a cosine learning-rate schedule and compiled generator training. Generator dropout is 0.1 at width 256 and 0.2 at width 512; warmup is 200 steps at width 256 except E19/E20, which use 1,000; width 512 also uses 1,000. The codec is additional to the generator sizes below.

Table 9: Later Sudoku generator settings. Steps are completed steps for the reported results; corruption gives the first and last replacement probabilities.

Run	Generator	Steps	Batch	Learning rate	Corruption
E10	8×512	24,000	64	3e-4	0.30 $\rightarrow$ 0.05
E11	8×256	32,000	64	6e-4	0.30 $\rightarrow$ 0.05
E12	8×512	50,776	64	3e-4	0.50 $\rightarrow$ 0.10
E13	8×256	32,000	64	6e-4	0.50 $\rightarrow$ 0.10
E14	8×256	128,000	64	6e-4	0.30 $\rightarrow$ 0.05
E15	8×256	128,000	64	6e-4	0.50 $\rightarrow$ 0.10
E16	8×256	32,000	64	6e-4	0.50 $\rightarrow$ 0.10
E17	8×256	32,000	64	6e-4	0.50 $\rightarrow$ 0.10
E18	8×256	128,000	64	6e-4	0.50 $\rightarrow$ 0.10
E19 pilot	8×256	23,895	128	6e-4	0.50 $\rightarrow$ 0.10
E19 funded	8×256	128,000	128	6e-4	0.50 $\rightarrow$ 0.10
E20	8×256	128,000	128	6e-4	0.50 $\rightarrow$ 0.10

The base data are 200,000 training puzzles and 1,000 validation puzzles, with split seed 0 and 28–45 givens. E15 adds one million greedy-unique puzzles for each seed 201–204; E16 adds one million for each seed 205–206. E17 adds the unique subset of one million random-removal candidates for each seed 101–104; its final 2,000 steps are a weights-only continuation with reset optimizer and random state (Section 6.5). E12’s recorded run completed 50,776 steps; the repository’s current E12 configuration specifies 200,000, so running that configuration unchanged does not reproduce the reported exposure. E18 adds no unique-puzzle corpus. Both E19 runs and E20 add one million greedy-unique puzzles per seed 201–208; their separate schedules use full-state snapshots every 1,000 updates. The original million-step E19 configuration remains unstarted and is not either reported result. Full settings and checkpoint prerequisites are in the corresponding `configs/sudoku-e*.json` files and `docs/EXPERIMENTS.md`.

Appendix B: Environment

Apple M5 Pro, 64 GB unified memory, macOS 26.5; MLX 0.32.3, NumPy, Python 3.12.14 managed with `mise` and `uv`. Figures: `matplotlib`. Paper: `Pandoc` and `Typst`. E18: RTX 3080, Python 3.12.15; both E19 runs and E20: NVIDIA A40, Python 3.12.3, MLX 0.32.3 CUDA. Preflight and contention qualifications accompany each run in the experiment log.

Appendix C: Version history

Version	Date	Changes
0.1	2026-10-23	First draft: method, implementation, TinyWorld and Sudoku results E0–E6.
0.2	2026-10-23	Plain-language introduction, glossary, formulas explained with a running example, experiment journey (E0–E9), hyperparameter appendix, NeurIPS-style layout, HTML, EPUB, LaTeX, JATS and DOCX editions.
0.3	2026-10-23	E8 results: conditioning corruption gives the first solved boards.

Version	Date	Changes
0.4	2026-10-23	E11: four times the training data gives 55% of easy boards; first run on NVIDIA.
0.5	2026-10-24	E13 (stronger corruption) and E14 (16× data): 87% of easy boards.
0.6	2026-10-24	E15 combines data, stronger corruption and unique puzzles: 96% of easy boards. Layout checks on every build.
0.7	2026-10-24	Full-state pause/resume and recovery checks; E17's weights-only continuation solves 71.25% of easy boards, with the optimizer-reset qualification.
0.8	2026-10-24	Release audit: codec/system parameter accounting, budget and sampling qualifications, prior-art scope and simplified visual identity.
0.9	2026-10-24	Submission preparation: public-data figure rebuild command, qualified memory-bandwidth discussion, later Sudoku settings and AI assistance disclosure.
0.10	2026-10-25	Completed E18/E19 results, checkpoint provenance, A40 timings, uncertainty and explicit budget differences.
0.11	2026-10-27	Completed E20 seed replication; two-seed assessment, hard-board failure accounting, explicit provenance limits and a gated text-research direction.