TINY AUTOREGRESSIVE RECURSIVE MODELS

Paulius Rauba
University of Cambridge
pr501@cam.ac.uk

Claudio Fanconi
University of Cambridge
caf83@cam.ac.uk

Mihaela van der Schaar
University of Cambridge
mv472@cam.ac.uk

ABSTRACT

Tiny Recursive Models (TRMs) have recently demonstrated remarkable performance on ARC-AGI, showing that very small models can compete against large foundation models through a two-step refinement mechanism that updates an internal reasoning state z and the predicted output y. Naturally, such refinement is of interest for any predictor; it is therefore natural to wonder whether the TRM mechanism could be effectively re-adopted in autoregressive models. However, TRMs cannot be simply compared to standard models because they lack causal predictive structures and contain persistent latent states that make it difficult to isolate specific performance gains. In this paper, we propose the Autoregressive TRM and evaluate it on small autoregressive tasks. To understand its efficacy, we propose a suite of models that gradually transform a standard Transformer to a Tiny Autoregressive Recursive Model in a controlled setting that fixes the block design, token stream, and next-token objective. Across compute-matched experiments on character-level algorithmic tasks, we surprisingly find that there are some two-level refinement baselines that show strong performance. Contrary to expectations, we find no reliable performance gains from the full Autoregressive TRM architecture. These results offer potential promise for two-step refinement mechanisms more broadly but caution against investing in the *autoregressive* TRM-specific model as a fruitful research direction.

1 INTRODUCTION

A common way to improve performance in autoregressive Transformers is to "increase compute". Yet, under a fixed decoder-block template, the same budget of decoder-block evaluations can be allocated in non-equivalent ways: (i) *untied depth* (distinct layers), (ii) *tied recurrent depth* (reusing a shared block across steps), or (iii) *within-token refinement* (multiple internal updates before emitting next-token logits), as just a few examples. These choices match compute in block passes but differ in parameter sharing, state evolution, and how gradients supervise intermediate computation (see Fig. 1).

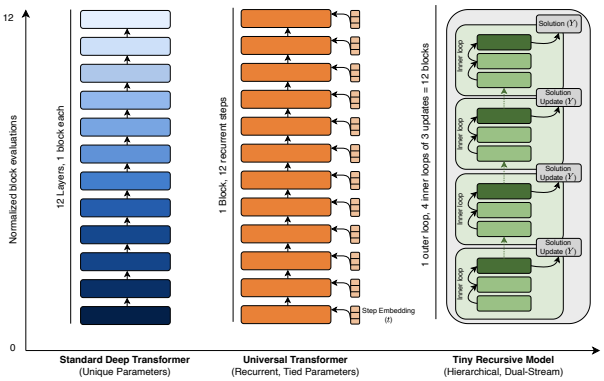

Figure 1: **Compute placement at fixed block-pass budget.** Three autoregressive decoders execute the same compute (12 decoder-block evaluations) but allocate it differently: **(left) Deep Transformer**: 12 untied layers, **(middle) Universal Transformer**: one shared block unrolled for 12 recurrent steps, with step embeddings, and **(right) Tiny Recursive Model**: hierarchical dual-stream refinement using multiple inner updates before each solution update. We investigate which allocation yields the best generalization per block evaluation.

Recent hierarchical refinement models motivate within-token refinement as latent multi-step computation, i.e. instead of emitting intermediate tokens, the model iteratively refines internal latents before producing logits Wang et al. (2025); Jolicoeur-Martineau (2025). The TRM model

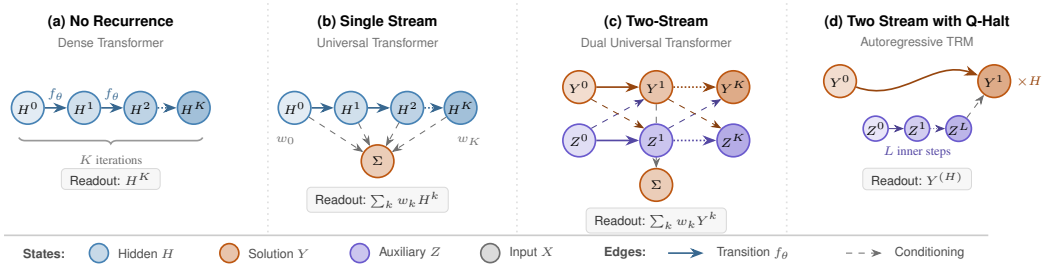

Figure 2: **Compute placement architectures.** (a) Single hidden-state stream with final-iterate readout. (b) Adaptive halting with weighted readout across all iterates. (c) Two-stream factorization: solution Y and auxiliary Z with cross-conditioning. (d) Nested hierarchy: L inner refinements of Z per outer update of Y, repeated H times.

shows particularly strong results due to its strong performance in the ARC-AGI challenge suite relative to standard LLMs[1]. Implicitly, this can be seen as a "*token-level reasoning*" hypothesis, where reasoning here is used loosely as simply another word for computation. Therefore, if this token-level reasoning hypothesis holds, then under matched compute, allocating iteration *within* each decoding step should improve generalization per block evaluation by enabling intra-step self-correction prior to committing a token.

Can we directly test this hypothesis in other settings, such as the autoregressive setting? To understand why and when it works, we would require isolating the specific mechanism that the TRM adopts under matched compute. We therefore take *compute placement* as the object of study and ask: *Given a fixed autoregressive decoder block template and a fixed next-token objective, how should iterative computation be allocated to maximize generalization per unit compute?* We answer this in a controlled autoregressive family that fixes the token stream and next-token loss, enforces standard causal masking and KV-cache semantics, uses an identical decoder block, and excludes routing and token-stream modifications. Within this family, architectures correspond to different unrollings of the same block template.

Findings. Across compute-matched experiments on character-level algorithmic tasks (addition, copying, reversing), untied depth and a *flat* two-stream recurrent baseline yield the strongest generalization per block evaluation. The autoregressive TRM model, contrary to expectations, provides no consistent benefit. In fact, in most experiments, the performance degrades sharply. This suggests that under autoregressive setups, maintaining an inner refinement loop does not yield benefits given equivalent compute. However, we also recognize these are early results on "tiny" models in smaller data regimes; and these could, in principle, change, in not-so-tiny autoregressive recursive models.

Contributions. (i) We formalize *compute placement* for autoregressive Transformers under a fixed block template and introduce a controlled ladder that isolates tying, step conditioning, halting/readout, and hierarchical refinement. (ii) We derive an autoregressive projection of TRM-style hierarchical refinement that preserves causal masking and removes cross-call latent carry which allow for compute-matched comparisons without token-stream changes. (iii) We empirically show that at matched block-pass budgets, untied depth and flat two-stream recurrence dominate, while token-internal hierarchical refinement is not a reliable route to improved autoregressive generalization.

2 Background

Preliminaries (autoregressive setting). Let $\mathcal{V}$ be a finite vocabulary and $\mathbf{x} = (x_1, \ldots, x_T) \in \mathcal{V}^T$ a length-T sequence. We model

$$p_\theta(\mathbf{x}) = \prod_{t=1}^{T} p_\theta(x_t \mid \mathbf{x}_{<t}), \qquad \mathbf{x}_{<t} = (x_1, \ldots, x_{t-1}),$$

[1]Despite its impressive results, two key differences in the setup are that the TRM was trained as a supervised learning model directly on available data; and used extensive data augmentation for ARC-AGI

and train by maximum likelihood,

$$\mathcal{L}(\theta) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid \mathbf{x}_{<t}).$$

At depth ℓ, hidden states are $H^{(\ell)} \in \mathbb{R}^{T\times d}$ with rows $h_t^{(\ell)} \in \mathbb{R}^d$, and $H^{(0)} = \mathrm{Embed}(\mathbf{x}) + \mathrm{Pos}(\mathbf{x})$ (optionally plus other additive signals). We use causal language modelling, enforcing causality with an attention mask that restricts each position to attend only to earlier tokens.

2.1 Transformers from a compositional point of view

Standard Transformers (Vaswani et al., 2017) implement $\mathbf{x}_{<t} \mapsto p_\theta(x_t \mid \mathbf{x}_{<t})$ by stacking L residual blocks. Each block $\ell \in \{0, \ldots, L-1\}$ applies $f_{\theta_\ell} : \mathbb{R}^{T\times d} \to \mathbb{R}^{T\times d}$ (attention–MLP), giving

$$H^{(\ell+1)} = f_{\theta_\ell}\left(H^{(\ell)}\right), \qquad H^{(L)} = f_{\theta_{L-1}} \circ \cdots \circ f_{\theta_0}\left(H^{(0)}\right).$$

Equivalently, a Transformer repeatedly applies a block template, composing L distinct transformations in sequence. This fixed-depth, acyclic graph is easy to optimize and parallelize, but hard-codes a uniform compute budget: each token receives exactly L block applications. Any additional iterative computation must therefore be realized indirectly (e.g., via extra tokens, a common technique for test-time scaling) or by modifying the architecture to allocate compute differently.

2.2 Universal Transformers

Universal Transformers (UTs) (Dehghani et al., 2018) replace untied depth with recurrent depth by repeatedly applying a single block f_θ:

$$H^{(k+1)} = f_\theta(H^{(k)}), \qquad k = 0, 1, \ldots, K-1,$$

with $H^{(0)}$ as above. For fixed K, this is a depth-K Transformer with full parameter tying, increasing computational steps without increasing parameters and interpreting recurrence as iterative refinement of a shared latent state. Because tying makes refinement steps ambiguous, UTs inject a step encoding so f_θ can condition on k.

UTs are typically combined with Adaptive Computation Time (ACT) (Graves, 2016), which adds a halting mechanism and forms outputs via a weighted accumulation of intermediate states with a ponder-style penalty. In principle this enables non-uniform compute across tokens. In practice, refinement is still coupled to a global recurrent mechanism (tokens share the same update sequence until halting), and predictions come from an aggregate over iterates rather than a final refined state. Thus, while UTs show recurrent depth can be effective, they do not resolve our finer question: under a fixed autoregressive decoder block, where does iteration improve generalization per unit compute?

2.3 Tiny Recursive Models: An extension of UT

Tiny Recursive Models (TRMs) (Jolicoeur-Martineau, 2025) have drawn attention because a 7M-parameter TRM reportedly generalizes well on ARC-AGI (e.g., 45% on ARC-AGI-1 and 8% on ARC-AGI-2 with heavy augmentation and test-time compute), suggesting recursive depth can support structured reasoning with small parameter budgets.

TRMs can be seen as extending UTs by keeping a tied refinement operator but changing *what* is refined and *how* it is read out. They decompose state into a fixed input stream $X = \mathrm{Embed}(\mathbf{x}) + \mathrm{Pos}(\mathbf{x})$, a solution stream Y, and an auxiliary reasoning stream Z. With a shared block f_θ, Z is refined while conditioning on X and the current solution, and Y is updated through the refined Z:

$$\begin{aligned} Z^{(k,i+1)} &= f_\theta\big(X + Y^{(k)} + Z^{(k,i)}\big), \qquad i = 0, \ldots, n-1, \\ Y^{(k+1)} &= f_\theta\big(Y^{(k)} + Z^{(k,n)}\big). \end{aligned}$$

This induces a hierarchical schedule: multiple inner refinements of Z precede each update of Y (repeated over an outer loop). TRM halting also differs from ACT (Graves, 2016): instead of a convex combination of iterates with a ponder penalty, TRMs use a binary halt head on the solution stream and output the terminal iterate $Y^{(K)}$.

We use TRMs here for a narrower purpose: they provide a clean way to place iterative computation while keeping the block and objective fixed. However, in the original implementation, TRMs are introduced as supervised learning solvers, whereas autoregressive decoding must emit next-token distributions at every step. Emitting explicit "thinking" tokens would change the token stream and thus the learning problem, so instead we adapt TRM refinement to operate *within* a masked decoder step: for a fixed prefix, we iteratively refine the hidden state producing next-token logits without extra tokens or routing (Sec. 3). Figure 2 summarizes the architectures.

3 AUTOREGRESSIVE TOKEN-LEVEL REFINEMENT

We return to the central question: *under matched compute, how should iterative computation be allocated within an autoregressive decoder?* Existing approaches increase untied depth, reuse a block as recurrent depth (Dehghani et al., 2018), or add within-step refinement intended to resemble "token-level reasoning" (Wang et al., 2025; Chen et al., 2025). However, these proposals are rarely directly comparable: they often change the token stream (e.g., scratch/pause tokens) (Wei et al., 2022; Nye et al., 2021; Zelikman et al., 2022), add conditional capacity via routing (Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2022), or alter the inference interface (e.g., solver-style refinement) (Bai et al., 2019), confounding where supervision acts. Methods that "think" by emitting extra tokens may help (Wei et al., 2022; Nye et al., 2021; Zelikman et al., 2022), but they also change the modelling problem by changing the generated sequence. Here we fix the token stream and next-token objective, and ask whether extra computation can be organized *internally* under strict autoregressive semantics.

Within this controlled scope, TRMs are of interest because they represent an extreme compute placement: refining a structured latent state multiple times *within* a single decoding step. If token-internal refinement yields gains in our controlled autoregressive family (Wang et al., 2025; Chen et al., 2025), the autoregressive TRM model ought to show empirical performance gains. The mismatch is that TRMs are proposed as supervised solvers with bidirectional attention and cross-call state carry. This section adapts TRM-style refinement to autoregressive decoding and uses it to study compute placement in autoregressive Transformers.

3.1 COMPUTE PLACEMENT AND WHY TRM-STYLE IS THE RIGHT STRESS TEST

Autoregressive compute can be increased in non-equivalent ways: as untied depth (distinct blocks), tied recurrent depth (reapplying a shared block) (Dehghani et al., 2018), or within-step refinement (multiple internal updates before logits) (Wang et al., 2025; Jolicoeur-Martineau, 2025). These mechanisms can match block evaluations per token, but they differ in the iterated state and in how predictions are read out.

This motivates a pragmatic taxonomy used throughout the paper. *Single-stream* variants differ by weight tying, step disambiguation, and whether intermediate iterates contribute via a weighted readout (Dehghani et al., 2018; Graves, 2016; Banino et al., 2021). *Multi-stream* variants (Wang et al., 2025; Jolicoeur-Martineau, 2025) decompose the latent state into asymmetric components (e.g., solution and auxiliary streams). This is a form of nested scheduling in which the auxiliary stream undergoes multiple inner refinements before each solution update[2]. A final axis is the readout: aggregate over iterates or use a terminal iterate (Graves, 2016; Banino et al., 2021; Jolicoeur-Martineau, 2025).

3.2 TWO CHALLENGES TO OVERCOME

TRMs are not directly compatible with strict autoregressive decoding for two reasons. ► **Challenge 1 (causality).** Solver-style TRMs use bidirectional attention, so an update at position i can depend on future positions $j > i$. We enforce strict autoregressive semantics by applying a causal mask. ► **Challenge 2 (cross-call carry).** TRMs maintain a persistent latent carry that is reset only upon halting. This can leak information across prefixes: computation performed under earlier contexts can

[2]This contrasts with alternative nesting mechanisms that focus on parameter-level nesting (Rauba & van der Schaar, 2026; Behrouz et al., 2025)

influence later logits beyond the explicit conditioning set $\mathbf{x}_{<t}$. We remove this by re-initializing the TRM latent streams on every forward pass, so logits at step t depend only on $\mathbf{x}_{<t}$ within that call.

These modifications isolate token-internal refinement within a fixed compute budget. For a given block-pass budget, the nested schedule simply partitions block evaluations across inner Z refinements and outer Y updates. Consequently, any difference from single-stream recurrence cannot be attributed to future leakage, cross-call state, routing, or token-stream changes (Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2022; Wei et al., 2022; Nye et al., 2021; Zelikman et al., 2022). In our compute-placement ladder (Sec. 4), this autoregressive TRM-style projection is the extreme token-internal rung; if it fails to improve generalization at equal compute, it provides a strong negative test for within-step refinement in this setting.

4 Mechanistic decomposition of iterative computation

We study *compute placement* in autoregressive Transformers: modifying how computation is allocated while holding fixed tokenization, the next-token objective, causal attention/KV-cache semantics, and the decoder-block template (attention–MLP with residual/norm), with no routing and no token-stream modifications. This controlled setting isolates where iteration matters.

Model	Tie	Step	ACT	2S	Nest	Q-halt
Dense Transformer	✗	✗	✗	✗	✗	✗
Iterative Transformer	✓	✗	✗	✗	✗	✗
Iterative Step Transformer	✓	✓	✗	✗	✗	✗
Universal Transformer	✓	✓	✓	✗	✗	✗
Dual UT	✓	✓	✓	✓	✗	✗
Dual Nested UT	✓	✓	✓	✓	✓	✗
Autoregressive TRM	✓	✓	✓/✗	✓	✓	✓/✗

Table 1: **Progressive deltas (single-column ladder).** Each row adds exactly one mechanism relative to the previous row. Tie=weight tying; Step=step embeddings; ACT=Graves ACT-style halting; 2S=two-stream (y, z); Nest=hierarchical $H \times L$ loop; Q-halt=TRM-style binary halting. **Note:** We have used terms that best describe the underlying model architecture; for instance, Autoregressive TRM is best described for what it is doing, while the non-autoregressive version is called TRM.

We define three criteria that we think a study with high internal validity should therefore satisfy. ► **Controlled family:** fixed token stream, objective, masking, and block structure, so variants differ only in iteration placement; ► **Axis isolation:** vary one mechanism at a time (e.g., tying, halting, readout), enabling causal attribution; ► **Compute normalization:** match compute by counting block evaluations rather than parameters.

Many existing approaches vary multiple axes at once: standard Transformers use untied depth, single-stream state, and final-state readout; UTs add weight tying and step embeddings; ACT and PonderNet add adaptive halting with weighted accumulation; TRMs add state decomposition, hierarchical iteration, and binary halting with terminal readout; and Deep Equilibrium Models replace explicit unrolling with implicit fixed-point solvers, changing both iteration and training dynamics. These bundled changes make mechanistic attribution difficult.

How to interpret Table 1. The table defines the controlled *compute-placement ladder*: each row adds exactly one mechanism while holding fixed the token stream and next-token objective, causal masking/KV-cache semantics, and the decoder-block template f_θ. Columns indicate: ►TIE: reuse the same parameters across the C block passes (tied recurrence) vs. distinct θ_ℓ (untied depth); ►STEP: inject a step signal e_k so a tied block can condition on iteration index; ►ACT: Graves-style adaptive computation time, i.e. halting with weighted readout over iterates $H^{(k)}$ (run in a compute-matched regime when required); ►2S: replace single-stream H with two-stream (Y, Z) conditioned on the fixed input stream $X = \mathrm{Embed}(\mathbf{x}) + \mathrm{Pos}(\mathbf{x})$; ►NEST: refine Z for L inner steps before each Y update (repeated H times), moving compute *within* a decoding step without changing the token stream; ►Q-HALT: replace ACT-style accumulation with TRM-style binary halting and terminal-iterate readout (logits from the final solution iterate). Under compute normalization (Sec. B), all rows execute the same number of block evaluations per forward pass, so differences can be attributed to the single added mechanism.

5 Evaluation

Evaluation methodology. We evaluate a controlled ladder of seven autoregressive architectures that differ only in *compute placement*, while holding fixed the token stream, next-token objective, causal

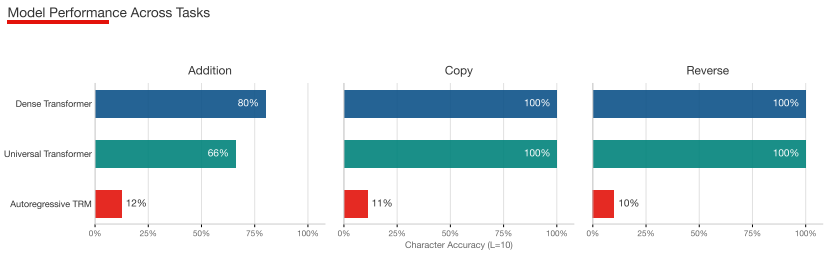

Figure 3: **Model performance across tasks.** Character accuracy (length 10) on Addition, Copy, and Reverse for three architectures: Dense Transformer, Universal Transformer (UT), and autoregressive TRM. Copy and Reverse are solved by the Dense Transformer and UT (100% accuracy), while Addition remains more difficult and separates these two models (80% vs. 66%). The autoregressive TRM performs poorly on all three tasks, reaching only 12%, 11%, and 10% accuracy, respectively.

masking/KV-cache semantics, and an identical decoder-block template (Pre-Layer-Normalization causal self-attention + GELU MLP with residuals), with no routing and no token-stream modifications. Compute is normalized by directly matching the number of decoder-block evaluations per forward pass ("block passes") to a fixed budget C, configuring each model's unrolling so that Dense uses C untied layers, tied/step-aware variants reuse a shared block for C steps (optionally with step embeddings), ACT-style UT variants are run in a full-compute mode (no early halting) to keep realized block passes fixed, and two-stream / nested variants subdivide the C passes across reasoning/solution updates (and inner refinements when nested). All models are trained under identical optimization and data conditions (procedurally generated character-level algorithmic sequences; next-token cross-entropy with loss on outputs only), and are evaluated on Copy, Reverse, and Addition with out-of-distribution length scaling using greedy decoding. Our evaluation metrics include sequence exact match, character accuracy, and position-wise accuracy by output quartiles to localize where failures accumulate. We do not compute the sensitivity of the autoregressive samplers to input perturbations (Rauba et al., 2025) because we only care about exactly predicting the single ground-truth answer given the input sequence.

5.1 Performance under equal compute

> **RQ1**. Under matched block-pass compute, which compute placements yield the strongest performance, and which tasks most strongly differentiate these choices?

Result Figure 3 compares three primary architectures on Addition, Copy, and Reverse. The Dense Transformer and UT both achieve perfect performance on Copy and Reverse, indicating that these two tasks are easy for these models at the tested sequence length. Addition is more demanding: the Dense Transformer reaches 80% character accuracy, while the UT reaches 66%. By contrast, the autoregressive TRM performs poorly on every task, with accuracy near 10% throughout, which is close to chance-level behavior at the character level[3].

Interpretation. The main pattern is that task demands interact strongly with architectural constraints. Copy and Reverse mainly test whether the model can preserve or reorder information already present in the input. Under the present training and evaluation setup, both the Dense Transformer and UT have sufficient capacity to do this reliably. Addition has a different structure: correct output requires intermediate state updates that remain consistent across positions, because local mistakes can propagate through the sequence via carry-like dependencies. This makes Addition a more sensitive test of whether the architecture can sustain a stable internal computation over multiple steps. The gap between the Dense Transformer and UT therefore shows that these two models, although both strong on Copy and Reverse, differ in how effectively they support that computation. The autoregressive TRM fails even on Copy and Reverse, which suggests that its difficulty is more basic: in this setting, it does not learn a robust input-output mapping for any of the three tasks.

[3]The associated github repo can be found at https://github.com/pauliusrauba/autoregressive-TRM

Takeaway. Copy and Reverse distinguish the autoregressive TRM from the other two models, while Addition further separates the Dense Transformer from the UT. The Dense Transformer performs best overall, the UT is strong but weaker on Addition, and the autoregressive TRM does not generalize successfully on any task shown.

5.2 Error concentration and learning bottlenecks

RQ2. Where do errors concentrate in autoregressive decoding, and when does training succeed (or fail) to resolve the final dependency under different compute placements?

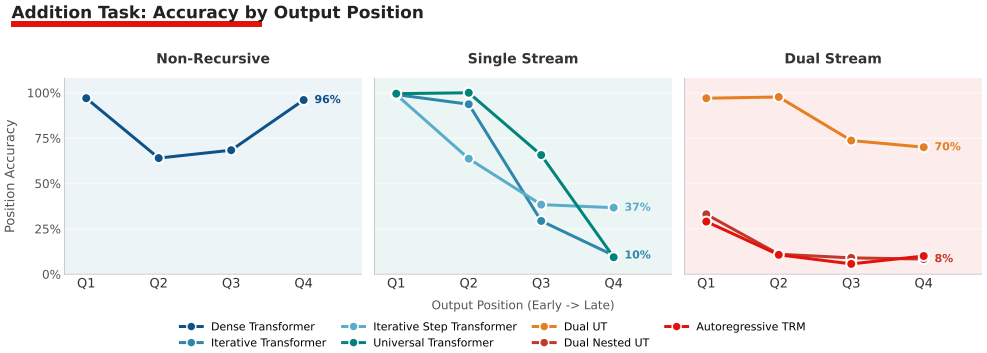

Figure 4: **Addition: accuracy stability by output position.** Character accuracy by output quartile. Dense remains high and nearly flat across positions, while several single-stream recurrent models exhibit sharp late-position collapse (Q4 $\approx$ 8–10%). Dual UT largely avoids this collapse.

Results. Figure 4 shows that addition errors are strongly *position-structured*. Dense is high and nearly flat across all quartiles, indicating stable decoding performance. In contrast, single-stream recurrent variants degrade sharply at late positions: Iterative, Iterative Step, and UT fall to roughly $\approx 10\%$, $\approx 9\%$, and $\approx 8\%$ in Q4, despite substantially higher accuracy earlier in the output. Dual UT largely avoids this late-position collapse, remaining comparatively stable across positions.

Figure 5 shows the corresponding *training-time* picture. Only Dense and Dual UT reliably break through the final-character bottleneck, with last-character accuracy rising sharply and reaching high values by the end of training. All other architectures remain essentially flat over the full training horizon: single-stream recurrent variants (Iterative, Iterative Step, UT) do not break through, and nested/terminal variants (Dual Nested UT and the Autoregressive TRM) also stay near chance.

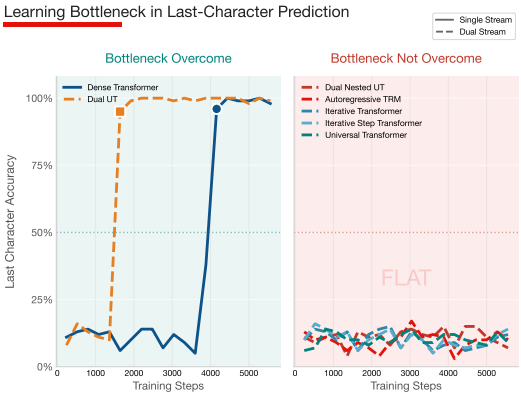

Figure 5: **Addition: training bottleneck on the final character.** Last-character accuracy over training. Only Dense and Dual UT reliably overcome the bottleneck; the remaining compute placements stay flat near chance.

Interpretation. Together, these plots point to addition requiring to learn and maintain a globally consistent carry-like summary across the decoding trajectory. Early output positions can often be predicted using local regularities (e.g., digitwise patterns) that do not require full consistency, so several models appear competent in Q1–Q3. The final part of the output is different: it is *maximally sensitive* to upstream inconsistencies, so small representational errors that are tolerable earlier accumulate and surface as a sharp drop in Q4. We interpret this in that architectures that remain flat on last-character accuracy never acquire the core global dependency, even if they learn partial heuristics that help earlier positions.

Takeaway. On addition, errors concentrate late because the final dependency is least tolerant to upstream inconsistency. Whether training overcomes this bottleneck depends strongly on compute placement: Dense and flat two-stream recurrence (Dual UT) cross it, while single-stream recurrence and nested/terminal refinement typically do not.

5.3 Learning dynamics

RQ3. Do different placements of the same compute budget merely shift final accuracy, or do they change *how* solutions are learned over training?

Result. Figure 6 shows qualitatively different learning trajectories under compute matching. Dense exhibits a long plateau followed by a late, sharp jump to high accuracy. Single-stream tied variants (Iterative, Iterative Step, UT) improve more gradually and plateau well below Dense. Dual UT accelerates earlier and reaches a higher plateau than single-stream recurrence. In contrast, nested/terminal variants (Dual Nested UT, Autoregressive TRM) remain low throughout, indicating failure to enter the high-accuracy regime. We find similar results in our further experiments, i.e. Dense models improve early positions and only later on late positions, whereas Dual UT's late positions rise more slowly and level off lower. However, we notice that the exact behavior is very sensitive to the hyperparameter and training setup used.

Interpretation. These dynamics are consistent with a shift from partial heuristics to an internal computation that resolves the global dependency in addition. Early positions admit local regularities, so models can improve without learning globally consistent carry propagation; high accuracy requires maintaining a carry-like summary that remains aligned with the readout across decoding. Untied depth may ease learning via stage-wise specialization, and flat two-stream recurrence may help by separating auxiliary computation from the solution interface. By contrast, nested refinement with terminal readout weakens credit assignment to early inner-loop steps, creating an optimization barrier that training often fails to cross.

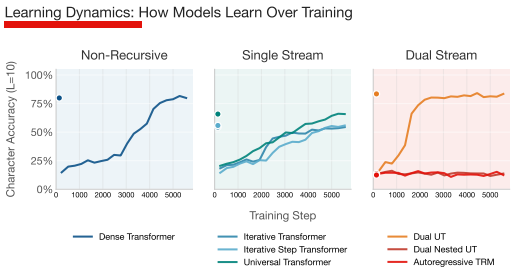

Figure 6: **Learning dynamics under compute matching (addition).** Held-out character accuracy (length L=10) over training steps. Dense shows a late, abrupt transition; single-stream tying improves gradually but plateaus; Dual UT accelerates earlier; nested/terminal variants remain low.

Takeaway. Under matched compute, compute placement changes learning dynamics: Dense and (to a lesser extent) Dual UT cross the final-dependency barrier, whereas single-stream tying often stalls and nested/terminal refinement frequently fails to learn it at all.

6 Discussion

Implications for "latent reasoning" claims in autoregressive models. Our results suggest that, under a strict next-token objective with causal masking, dual stream architectures *can* result in superior generalization, but the autoregressive TRM model does not exhibit such behavior. We posit that this might be attributable to either credit-assignment mechanisms intenral within different architectures or improved representation learning within a compute budget that can be exploited early on in the training. Importantly, we find that such "dual stream" autoregressive settings can indeed hold promise, but, at least in the small data algorithmic regime, the autoregressive TRM model does not seem to be a worthwhile research pursuit. We would be interested to see fruitful work which extends this idea further to not-so-tiny autoregressive settings or more complicated settings that require higher levels abstraction as a part of the solution space.

Impact Statement This work has an impact on at least several areas of machine learning and artificial intelligence, none of which we think are important to explicitly highlight.

Acknowledgements We thank reviewers for their feedback and comments. Canon Medical Systems Corporation funds CF's studentship. This work was supported by Azure sponsorship credits granted by Microsoft's AI for Good Research Lab.

REFERENCES

Bai, S., Kolter, J. Z., and Koltun, V. Deep equilibrium models. *Advances in neural information processing systems*, 32, 2019.

Bai, S., Koltun, V., and Kolter, J. Z. Multiscale deep equilibrium models. *Advances in neural information processing systems*, 33:5238–5250, 2020.

Banino, A., Balaguer, J., and Blundell, C. Pondernet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021.

Behrouz, A., Razaviyayn, M., Zhong, P., and Mirrokni, V. Nested learning: The illusion of deep learning architectures. *arXiv preprint arXiv:2512.24695*, 2025.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Chen, Y., Shang, J., Zhang, Z., Xie, Y., Sheng, J., Liu, T., Wang, S., Sun, Y., Wu, H., and Wang, H. Inner thinking transformer: Leveraging dynamic depth scaling to foster adaptive internal thinking. *arXiv preprint arXiv:2502.13842*, 2025.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, Ł. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

Elbayad, M., Gu, J., Grave, E., and Auli, M. Depth-adaptive transformer. *arXiv preprint arXiv:1910.10073*, 2019.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

Graves, A. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Gu, J., Wang, C., and Zhao, J. Levenshtein transformer. *Advances in neural information processing systems*, 32, 2019.

Hahn, M. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Jacobs, M., Fel, T., Hakim, R., Brondetta, A., Ba, D., and Keller, T. A. Block-recurrent dynamics in vision transformers. *arXiv preprint arXiv:2512.19941*, 2025.

Jain, A. and Linares, R. Tiny recursive control: Iterative reasoning for efficient optimal control. In *AIAA SCITECH 2026 Forum*, pp. 2380, 2026.

Jolicoeur-Martineau, A. Less is more: Recursive reasoning with tiny networks. *arXiv preprint arXiv:2510.04871*, 2025.

Kaiser, Ł. and Sutskever, I. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

Liao, Q. and Poggio, T. Simple recursive model: Simplified, single-state reasoning with skip connections. 2026.

Liu, W., Zhou, P., Wang, Z., Zhao, Z., Deng, H., and Ju, Q. Fastbert: a self-distilling bert with adaptive inference time. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pp. 6035–6044, 2020.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

Padayachy, K., Richman, R., and Wüthrich, M. V. Tab-trm: Tiny recursive model for insurance pricing on tabular data. *arXiv preprint arXiv:2601.07675*, 2026.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Rauba, P. and van der Schaar, M. Deep hierarchical learning with nested subspace networks. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ymUOPsbxLi.

Rauba, P., Seedat, N., Kacprzyk, K., and van der Schaar, M. Self-healing machine learning: A framework for autonomous adaptation in real-world environments. *Advances in Neural Information Processing Systems*, 37:42225–42267, 2024.

Rauba, P., Wei, Q., and Van Der Schaar, M. Statistical hypothesis testing for auditing robustness in language models. *arXiv preprint arXiv:2506.07947*, 2025.

Reed, S. and De Freitas, N. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Stern, M., Chan, W., Kiros, J., and Uszkoreit, J. Insertion transformer: Flexible sequence generation via insertion operations. In *International Conference on Machine Learning*, pp. 5976–5985. PMLR, 2019.

Sukhbaatar, S., Weston, J., Fergus, R., et al. End-to-end memory networks. *Advances in neural information processing systems*, 28, 2015.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wang, G., Li, J., Sun, Y., Chen, X., Liu, C., Wu, Y., Lu, M., Song, S., and Yadkori, Y. A. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025.

Wang, W. and Reid, F. Tiny recursive reasoning with mamba-2 attention hybrid. *arXiv preprint arXiv:2602.12078*, 2026.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Weston, J., Chopra, S., and Bordes, A. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

Yoo, S., Malla, S., Choi, C., Lu, W. D., and Choi, J. H. Adept: Adaptive dynamic early-exit process for transformers. *arXiv preprint arXiv:2601.03700*, 2026.

Zelikman, E., Wu, Y., and Goodman, N. D. Star: Self-taught reasoner. In *Proceedings of the NIPS*, volume 22, 2022.

A Extended Related Work

Summary. Prior work on improving autoregressive Transformers can be understood as exploring alternative ways to allocate additional computation, but typically does so by coupling multiple design choices. Fixed-depth Transformers assign uniform compute per token and scale primarily through parameters, data, and training compute (Vaswani et al., 2017; Brown et al., 2020; Hoffmann et al., 2022), yet both theory and empirics show that certain algorithmic or multi-step dependencies fundamentally require depth or iterative computation rather than width alone (Hahn, 2020; Pérez et al., 2021; Wang et al., 2025), motivating approaches such as Chain-of-Thought that externalize iteration into the token stream (Wei et al., 2022). Other work internalizes iteration by reusing parameters across steps, including Universal Transformers (Dehghani et al., 2018) and adaptive halting mechanisms such as ACT and PonderNet (Graves, 2016; Banino et al., 2021), as well as related inner-thinking or early-exit methods (Elbayad et al., 2019; Xin et al., 2020; Chen et al., 2025), though these often conflate compute allocation with halting behavior, parameter efficiency, or inference-time latency. Hierarchical and multi-timescale models further decompose latent state and update schedules, separating fast refinement from slower solution updates (Wang et al., 2025; Jolicoeur-Martineau, 2025), continuing a broader tradition of iterative latent computation in memory-augmented and algorithmic models (Graves et al., 2014; Sukhbaatar et al., 2015; Kaiser & Sutskever, 2015), in contrast to rationale-based methods that render iteration explicit in generated text (Nye et al., 2021; Zelikman et al., 2022). Orthogonal approaches approximate implicit depth via fixed-point computation (Bai et al., 2019; 2020), allocate conditional capacity through routing in mixture-of-experts models (Shazeer et al., 2017; Fedus et al., 2022), or alter the token stream through non-monotonic or multi-pass decoding (Stern et al., 2019; Gu et al., 2019). Finally, some work offers refinement not via latent level processes but by iterative hypothesis generation and refinement of solutions (Rauba et al., 2024). In contrast, our work holds the autoregressive setting fixed—no routing and no token-stream modification—and isolates compute placement by directly comparing depth, recurrence, hierarchy, halting, and readout under matched block-evaluation budgets to identify where internal iteration most effectively improves generalization.

Fixed-depth Transformers and language modeling. A large body of work builds on the fixed-depth Transformer architecture (Vaswani et al., 2017), which underlies most modern LLMs and has historically improved via scale in parameters, data, and compute (Devlin et al., 2019; Radford et al., 2019;?; Brown et al., 2020; Kaplan et al., 2020; Hoffmann et al., 2022). Mechanistically, fixed depth assigns identical compute to each position, so "easy" and "hard" tokens receive the same number of block applications. This matters because constant-depth attention has formal limits for certain iterative computations: some algorithmic dependencies require depth (or a simulation of depth through additional generation steps) rather than width alone (Hahn, 2020; Pérez et al., 2021). Empirically, even when deeper stacks help, gains can diminish on tasks that naturally require multi-step computation (Wang et al., 2025). A common workaround is to externalize iteration into the token stream—e.g., Chain-of-Thought—thereby increasing decoding length and committing intermediate steps as text (Wei et al., 2022). In contrast, we keep the token stream fixed and study how different internal unrollings of the same decoder block change performance per unit compute.

Recurrent depth and adaptive computation. A separate line of work increases effective depth by reusing parameters across steps, turning depth into a recurrent refinement process. Universal Transformers tie layer parameters and repeatedly apply a shared block, typically augmented with step embeddings to disambiguate iterations (Dehghani et al., 2018). Building on this, Adaptive Computation Time (ACT) introduces a learned halting unit that can stop early and often forms outputs via a weighted accumulation of intermediate states, together with a ponder-style regularizer (Graves, 2016). PonderNet reframes halting as a probabilistic process, aiming for improved training behavior while preserving the idea of variable computation per input (Banino et al., 2021). More recent "inner thinking" style approaches also allocate extra internal steps selectively, but typically co-vary compute allocation with additional architectural or training choices (Chen et al., 2025). In parallel, early-exit and depth-adaptive inference methods (e.g., DeeBERT / FastBERT) stop computation once confidence is high, primarily targeting latency rather than improved generalization (Elbayad et al., 2019; Xin et al., 2020; Liu et al., 2020). Weight sharing is also used for parameter efficiency in settings such as ALBERT (Lan et al., 2019); however, parameter efficiency and compute placement are distinct, since recurrence changes the computation graph even when parameter count is held fixed.

Our ladder separates these effects by varying tying, step disambiguation, halting/readout, and state structure under equalized block-pass budgets.

Hierarchical and multi-step latent reasoning. Beyond flat recurrence, several approaches introduce multiple timescales or state decompositions, separating fast refinement from slower solution updates. The Hierarchical Reasoning Model (HRM) illustrates the utility of multi-rate computation for structured reasoning, suggesting that "where iteration happens" can matter as much as "how much iteration happens" (Wang et al., 2025). Tiny Recursive Models (TRM-style) similarly implement an inner loop that refines a "reasoning" state and an outer loop that updates a "solution" state, achieving strong generalization on ARC-style settings despite small parameter counts (Jolicoeur-Martineau, 2025). These ideas connect to a longer tradition of explicitly multi-step neural computation, including multi-hop memory updates (Weston et al., 2014; Sukhbaatar et al., 2015), iterative read/write controllers over external memory (Graves et al., 2014), and tied-parameter procedure learning for algorithmic tasks (Kaiser & Sutskever, 2015; Reed & De Freitas, 2015). A key distinction is whether iteration is latent or textual: scratchpads and rationale-driven methods push multi-step computation into generated text (Wei et al., 2022; Nye et al., 2021; Zelikman et al., 2022), whereas HRM/TRM-style models aim to keep iteration latent, enabling internal self-correction without emitting intermediate tokens. Our work sits in this latent-iteration family, but focuses on controlled comparisons: single-stream vs. dual-stream state, flat vs. hierarchical schedules, and final-state vs. weighted-accumulation readout, all under matched compute.

TRM-related work Recently, since the publication of the TRM paper, a number of subsequent follow-up papers have appeared. These include applying TRMs for control problems (Jain & Linares, 2026), studies raising the "block-recurrent hypothesis" (Jacobs et al., 2025), analyses of TRMs on tabular data (Padayachy et al., 2026) or applying architectural changes to TRMs (Wang & Reid, 2026; Liao & Poggio, 2026).

Implicit depth and conditional computation. A different strategy is to avoid explicit unrolling and instead solve for an equilibrium. Deep Equilibrium Models compute a fixed point of a transformation and backpropagate via implicit differentiation, effectively representing infinite depth governed by solver dynamics (Bai et al., 2019); multiscale variants extend this idea across timescales (Bai et al., 2020). While these models share the theme of refinement-to-convergence, the iteration is defined by numerical solvers rather than an explicit architectural schedule, which complicates axis-by-axis attribution. Orthogonally, Mixture-of-Experts increases conditional capacity through routing rather than sequential iteration (Shazeer et al., 2017; Lepikhin et al., 2020; Fedus et al., 2022); we explicitly exclude routing to avoid conflating compute placement with conditional capacity. Finally, non-monotonic generation strategies refine sequences over multiple passes by inserting or editing tokens (Stern et al., 2019; Gu et al., 2019), and dynamic early-exit generation variants further modify decoding trajectories (Yoo et al., 2026). These approaches demonstrate that iterative refinement can be effective, but they change the token sequence or decoding semantics relative to strict left-to-right generation. In contrast, our controlled setting holds the autoregressive family fixed—no routing, no token-stream modifications, explicit unrolling—and evaluates which placements of a fixed block-evaluation budget meaningfully improve generalization.

B Experimental Framework

Compute normalization. We measure compute in *block passes*, i.e., the number of evaluations of the decoder block executed in a single forward pass. This is the natural unit in our controlled family: it captures the dominant cost, matches the unrolling viewpoint of Section 4, and—unlike parameter count—does not confound compute with parameterization. In particular, increasing untied depth increases both the number of block evaluations and the number of distinct parameter sets, whereas increasing tied recurrence increases only the number of evaluations.

Accordingly, we fix a target budget C and configure each model so that every forward pass performs exactly C block evaluations. For dense (untied) depth and tied single-stream variants, one macro step consists of one block pass, and therefore $C = n$. For 2-phase variants, each macro step performs two block passes (a reasoning update followed by a solution update), giving $C = 2n$. For nested variants, each macro step executes H outer cycles, and within each outer cycle performs L inner refinements

and one solution update, so that $C = n \cdot H \cdot (L + 1)$. For example, at $C = 12$, a dense model uses $n = 12$, a 2-phase model uses $n = 6$, and a nested configuration can use $n = 4$ with $(H, L) = (1, 2)$.

This normalization yields the intended causal comparison: holding C fixed, architectures differ only in the *placement* of the C block evaluations—untied depth versus tied recurrence versus within-token refinement—rather than in the number of evaluations performed.

B.1 A CONTROLLED LADDER OF COMPUTE PLACEMENT

We evaluate seven architectures that form a controlled ladder. The ladder is constructed so that consecutive models differ by exactly one axis, while remaining in the same autoregressive family (tokenization, masking, objective, and block definition held fixed). Concretely, all variants are expressed as alternative unrollings of repeated applications of the same decoder-block template, differing only in (i) whether parameters are tied across applications, (ii) whether applications are step-conditioned, (iii) whether the compute budget is governed by a halting rule and whether intermediate iterates are exposed at readout, and (iv) whether the latent state is structured and updated via a flat or hierarchical schedule.

We begin with the *Dense Transformer*, the standard decoder-only baseline, which composes n distinct blocks sequentially and forms predictions from the final depth representation. We then isolate *weight sharing* by tying all blocks, yielding the *Tied Transformer* in which the same block is applied recurrently:

$$H^{(k+1)} = f_\theta\big(H^{(k)}\big), \qquad k = 0, \ldots, K-1.$$

Next, we isolate *step disambiguation* by injecting step embeddings, giving the *Tied Step-Aware Transformer*:

$$H^{(k+1)} = f_\theta\big(H^{(k)} + e_k\big), \qquad k = 0, \ldots, K-1.$$

From this point, we introduce *adaptive compute* via *Universal Transformers* with ACT (Graves, 2016). ACT augments the recurrent unrolling with a learned halting rule and, in its standard form, exposes intermediate iterates through a weighted accumulation at readout. Because our goal is compute-equal comparison, we use ACT in a controlled regime when needed: we disable the ponder penalty and force full computation so that the realized number of block evaluations is fixed to C, ensuring that halting does not change the compute budget.

We then isolate the effect of *state structure* by moving from a single stream to a two-stream update, producing the *2-phase Universal Transformer*. Writing $X = \mathrm{Embed}(\mathbf{x}) + \mathrm{Pos}(\mathbf{x})$, one macro step updates a reasoning stream Z conditioned on (X, Y, Z) and then updates a solution stream Y conditioned on the refined reasoning state:

$$Z' = f_\theta(X + Y + Z), \qquad Y' = f_\theta(Y + Z').$$

This transformation places additional computation *within* the autoregressive step without altering the token stream and without introducing routing. We then isolate *iteration hierarchy* by nesting multiple refinements of the reasoning stream prior to each solution update (the *Nested 2-phase UT*): before updating Y, we apply L successive applications of f_θ to the reasoning stream Z (each conditioned on X and the current Y), and only then perform the solution update. Finally, we isolate the effect of *readout and termination* by replacing ACT-style weighted accumulation with binary halting and final-iterate readout (the *Tiny Recursive Model*, autoregressive TRM-style). In this variant, the computation halts when $\sigma(q) > 0.5$, and the terminal solution iterate $Y^{(K)}$ is used to produce next-token logits, yielding a controlled instantiation of token-internal refinement beyond recurrent depth (Section 2.3).

Throughout this ladder, each architecture is obtained by adding exactly one compute-placement mechanism relative to the previous one, so that differences in performance can be causally attributed to the corresponding axis under the compute normalization of Section B.

B.2 TRAINING PROTOCOL

All models are trained under identical optimization and data conditions so that observed differences can be attributed to compute placement rather than to training confounds. We optimize with AdamW at learning rate 3×10^{-4}, batch size 64, for a fixed number of gradient steps (1,000–5,000 across

sweeps). Training sequences are generated on-the-fly from procedural generators, yielding an effectively unbounded stream of examples; this removes memorization as a viable strategy and makes generalization the primary signal through which additional compute can help.

Two implementation controls are essential for internal validity within our compute-normalized comparisons. For ACT-based models, we optionally set the ponder cost to zero and enforce full computation so that each forward pass executes the same fixed budget of C block passes as the non-halting baselines, ensuring that adaptive halting does not change realized compute. Separately, when evaluating extrapolation to increased test-time compute, we optionally disable step embeddings so that increasing the number of recurrent applications does not introduce an out-of-support step signal that could confound the effect of extra iterations.

B.3 Tasks and evaluation

We evaluate on three character-level algorithmic tasks chosen to induce qualitatively different causal dependencies under left-to-right decoding: **Copy** (predominantly local identity), **Reverse** (long-range dependence), and **Addition** (structured multi-step dependence via carry propagation). Each task is defined by a training length L_{train} and evaluated out of distribution by scaling length by fixed factors, i.e., we test at $L_{\text{eval}} = \alpha\, L_{\text{train}}$ with $\alpha \in \{1, 1.5, 2, 3, 5\}$, thereby directly probing length generalization. Decoding is greedy, and we evaluate 100 samples per condition. We report (i) sequence-level exact match, (ii) character-level accuracy, and (iii) position-wise accuracy via a quartile partition of the output positions, which localizes where errors accumulate along the generated sequence.

B.4 Controlled variables

To satisfy the controlled-family desideratum of Section 4, we hold fixed the token stream, objective, attention semantics, and block template across all variants, so that only the *placement* of block evaluations differs. Concretely, all models use the same character-level vocabulary of 14 tokens (digits 0–9, $|$, $+$, $=$, and newline) and are trained with next-token cross-entropy under prompt masking (loss on output tokens only). The decoder block is identical in every model: Pre-LayerNorm causal self-attention followed by a GELU MLP with $4d$ expansion and residual connections, evaluated under standard causal masking compatible with KV caching. We introduce neither routing (no Mixture-of-Experts / Mixture-of-Depths) nor token-stream modifications (no pause/think tokens and no auxiliary output streams). Under these constraints, any difference in performance arises from how the same block evaluations are scheduled and exposed, enabling causal attribution in Section 5 to specific compute-placement mechanisms.