Neural Firmware for Exact Integer Addition

A Controlled Small-Transformer Study

Josiah Wilson
Independent Researcher

July 26, 2026

Abstract

Autoregressive neural networks can imitate arithmetic procedures without reliably preserving them outside the lengths represented in training. We test a narrow alternative: place a known deterministic process inside the model's forward path as immutable "neural firmware," while leaving the surrounding transformer trainable. A 264,480-parameter causal transformer was compared with an otherwise identical model augmented by a frozen decimal ripple-carry transducer. The transducer encoded the correct next answer token as a fixed latent vector; the transformer's learned output head decoded that vector. Models were trained on additions with one- to six-digit nonnegative operands. In a preregistered confirmatory study using three unseen training seeds, the generic transformer averaged 90.13% exact-match accuracy in range but 0% on both 7–12 and 13–20 digit random additions. The latent-firmware model achieved 100% on every split for every seed, including 10,500 of 10,500 confirmatory examples across its three runs. A direct-logit firmware upper bound was also perfect. A post-hoc ablation that set firmware strength to zero reduced the trained latent models to 0% on both extrapolation splits. These results establish a limited engineering proposition: an exact process can be installed as frozen computation and made usable through a learned latent interface on a controlled grammar. They do not establish general mathematical reasoning. The input parser and operation selection were fixed, the baseline was not a task-specialized arithmetic transformer, and the firmware was a PyTorch transition-table module rather than a program compiled entirely into standard transformer weights.

1 Introduction

Language models generate sequences by learning statistical structure. This makes them flexible, but it also asks gradient-based optimization to rediscover procedures that are already known exactly. Arithmetic is a canonical example: models can interpolate well over familiar operand lengths while failing abruptly on longer inputs [1, 12]. Increasing scale or arithmetic data can reduce errors without making addition truth-preserving by construction.

An alternative is to separate uncertain inference from known computation. When a process is already fully specified—integer addition, parsing a formal grammar, applying a type rule, or advancing a board-game state—the process need not be relearned as an approximation. It can instead be installed as immutable computation and connected to a learned model at its boundaries. We call such a component *neural firmware*: fixed algorithmic structure executed within a neural model's forward path.

This idea is adjacent to neural arithmetic modules [4], neural algorithmic reasoning [6], differentiable and neural compilation [5, 10], compiled transformers [8, 9], and hybrid transformer–reasoner

architectures [11]. Recent work has also translated symbolic scientific programs into frozen differentiable PyTorch modules [17]. Our experiment is intentionally smaller. It asks whether an immutable ripple-carry process can be injected into a compact autoregressive transformer through a fixed latent code and remain exact when the learned baseline leaves its training-length distribution.

The study makes three contributions:

1. It implements a zero-trainable-parameter decimal ripple-carry module and integrates its outputs into the residual stream of a causal transformer.
2. It evaluates the interface in a preregistered, multi-seed experiment with procedurally generated, held-out additions and adversarial carry chains.
3. It documents an important boundary discovered during pilots: deterministic execution alone is insufficient if its latent signal can be overwhelmed by learned activations at unseen positions.

The intended claim is narrow. This is a controlled proof of engineering feasibility, not a demonstration that mathematical hallucination has been solved.

2 Related Work

2.1 Learning arithmetic algorithms

Neural architectures have long been tested on arithmetic because algorithms provide exact targets and clean extrapolation regimes. Neural GPUs learned binary addition and multiplication on inputs longer than those in training [2], and later refinements learned decimal multiplication [3]. Neural Arithmetic Logic Units introduced architectural biases for additive and multiplicative extrapolation [4]. These approaches seek learned mechanisms that behave algorithmically.

Transformers can also length-generalize under carefully chosen inductive biases. Relative positions can support addition beyond trained lengths [12]; position coupling aligns digits of equal significance and has generalized from 30 to 200 digits [13]; calibrated attention biases and structured scratchpads provide additional routes [15, 14]. Consequently, the baseline in this paper should not be interpreted as the strongest known arithmetic transformer. It is a deliberately generic causal language-model architecture. Our question differs: what happens when the known operation is specified and frozen rather than induced?

2.2 Neural compilation and hybrid reasoners

Adaptive Neural Compilation showed that low-level programs can be mapped into differentiable representations and subsequently optimized [5]. RASP supplied a programming abstraction for transformer computation [7], and Tracr compiled human-readable RASP programs into standard decoder-only transformer weights [8]. ALTA extended compiler-based analysis with loops and Universal Transformers [9].

Several works move closer to language-model integration. Algorithmic Language Models augment transformer architectures with memory, registers, basic operations, and neurally compiled program libraries [10]. TransNAR combines transformer language representations with frozen neural algorithmic reasoners [11]. Tracr-Injection distills compiled symbolic variables into GPT-2 residual streams, but reports inconsistent out-of-distribution generalization and 95% accuracy on its integer addition experiment [16]. The Neural Compiler translates first-order symbolic expressions into frozen differentiable modules that match their source programs to floating-point precision [17].

Our prototype sits between external tool use and full neural compilation. The ripple-carry process is an `nn.Module` executed inside the forward path and represented by immutable integer transition tables. It is not an external calculator call, but it is also not yet compiled entirely into ordinary attention and feed-forward weights.

The closest prior integrated-calculator work is Dietz and Klakow's Integrated Gated Calculator (IGC) [18]. IGC freezes Llama 3.1 8B and trains learned input and output mappings around a non-differentiable calculator, reporting 98–99% on BIG-Bench Arithmetic across four operations. It therefore predates this experiment as evidence for the broad idea of an internally gated exact calculator. The present study is narrower: a controlled small-transformer test of an immutable typed addition process, multi-seed length extrapolation, and causal ablation. The studies use different models, tasks, and interfaces and are not head-to-head.

3 Method

3.1 Task and tokenizer

Each sequence followed the grammar

$$\texttt{<bos>}\ a\texttt{+}b\texttt{=}c\texttt{<eos>}, \qquad c = a + b,$$

where a and b were nonnegative base-10 integers without leading zeros except for zero itself. A transparent character tokenizer contained 15 symbols: padding, beginning- and end-of-sequence symbols, ten digits, plus, and equals. Loss was applied only to answer digits and the end token.

Training operand lengths were sampled independently and uniformly from one through six digits. Training examples were generated online, eliminating finite-dataset memorization and conventional benchmark contamination.

3.2 Generic causal transformer

All models shared the same trainable architecture:

- three causal transformer encoder layers used as a decoder-only model;
- hidden width 96, four attention heads, and feed-forward width 256;
- fixed sinusoidal positions, zero dropout, and maximum sequence length 96;
- an untied linear vocabulary head.

Each model contained 264,480 trainable parameters. Firmware buffers were not counted as parameters.

3.3 Frozen ripple-carry firmware

The deterministic module stored two transition tables over $(d_a, d_b, r) \in \{0, \ldots, 9\}^2 \times \{0, 1\}$:

$$s(d_a, d_b, r) = (d_a + d_b + r) \bmod 10, \tag{1}$$

$$r'(d_a, d_b, r) = \left\lfloor \frac{d_a + d_b + r}{10} \right\rfloor. \tag{2}$$

Digits were processed from least to most significant and the resulting sequence was reversed for normal left-to-right rendering. The tables were registered buffers and the module exposed no trainable parameters. Unit tests compared it with Python integer addition on 2,000 random pairs up to 20 digits.

The parser located the plus and equals tokens and extracted digit-token operands. This parser was fixed and grammar-specific. The experiment therefore does not test learned semantic parsing or learned operation selection.

3.4 Latent and direct interfaces

For each vocabulary item y, a fixed random unit vector $C[y] \in \mathbb{R}^{96}$ was generated once using seed 314159. At a valid answer-generation position t, the latent-firmware model computed the exact next token $y_t^\star$ and modified the normalized transformer state:

$$\widetilde{h}_t = h_t + \alpha C[y_t^\star], \qquad \alpha = 32.$$

The ordinary trainable output head then produced logits from $\widetilde{h}_t$. Thus the deterministic module supplied exact content, but the model still had to learn how the fixed latent code mapped to output tokens.

The direct-firmware upper bound added a large one-hot bias for $y_t^\star$ directly to the logits. It tested whether the transition process and autoregressive alignment were exact independent of latent decoding.

Pilot experiments selected $\alpha = 32$ before confirmatory seeds or examples were evaluated. At $\alpha = 8$, the model often terminated early at unseen positions even though the ripple-carry output was correct. Signals of 20 or more removed those pilot errors. This observation motivated the conservative confirmatory strength of 32.

3.5 Training

AdamW used a learning rate of 5×10^{-4}, weight decay 0.01, batch size 128, and gradient-norm clipping at 1.0. The generic baseline received 10,000 steps because pilots showed that it required that budget to exceed 90% in-range exact match. The latent interface converged within 1,000 steps and received 1,000 confirmatory steps. The direct upper bound received one step because its result was structural, not learned. Giving the baseline ten times more optimization steps makes the accuracy comparison conservative, but prevents wall time from being interpreted as a controlled efficiency result.

3.6 Preregistration and evaluation

Pilot observations, decisions, and the confirmatory configuration were committed before confirmatory execution. Confirmatory training seeds were 101, 211, and 307; none appeared in pilots. Evaluation used the unseen seed 918273. All models were evaluated on the same committed examples:

- **ID random**: 1,000 additions with one- to six-digit operands;
- **OOD primary**: 1,000 additions with 7–12 digit operands;
- **OOD long**: 1,000 additions with 13–20 digit operands;
- **carry chain**: 500 constructed additions dominated by long runs of trailing nines.

The primary outcome was sequence-level exact-match accuracy on OOD primary, aggregated across the three training seeds. The logical evaluation-set hash was `917935e56a1d…23bd9b`; the full value appears in the appendix.

Experiments ran locally on an Apple M4 Mac mini with a 10-core GPU and 16 GB unified memory, using Python 3.12.12 and PyTorch 2.13.0 on MPS.

4 Results

4.1 Confirmatory accuracy

Table 1 reports mean exact-match accuracy over three training seeds. The generic baseline learned the trained range, averaging 90.13%, but produced no correct answers on either random extrapolation split. It solved one of 1,500 carry-chain cases across all seeds. Both firmware interfaces were perfect on every confirmatory example.

Table 1: Exact-match accuracy, mean ± sample standard deviation across three training seeds.

Model	ID 1–6	OOD 7–12	OOD 13–20	Carry chain
Baseline	0.9013 ± 0.0145	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0007 ± 0.0012
Latent firmware	1.0000 ± 0.0000	1.0000 ± 0.0000	1.0000 ± 0.0000	1.0000 ± 0.0000
Direct firmware	1.0000 ± 0.0000	1.0000 ± 0.0000	1.0000 ± 0.0000	1.0000 ± 0.0000

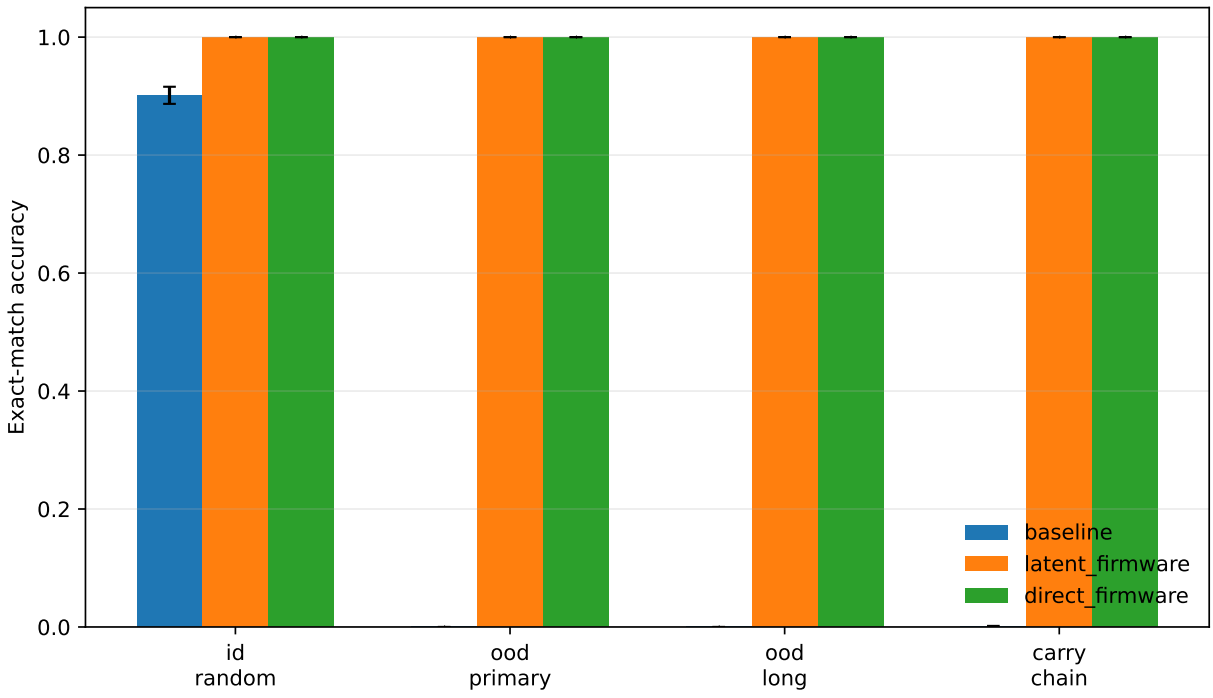

Figure 1: Exact-match accuracy by evaluation split. Error bars show standard deviation across three training seeds.

The primary latent-minus-baseline difference was 1.000 for each training seed, or 100 percentage

points. A bootstrap over the three seed-level differences therefore returned a degenerate 95% interval of [1.000, 1.000]. This interval describes these seeds only; three seeds are insufficient for a broad claim about optimization variability.

Across its three runs, latent firmware answered 10,500 of 10,500 confirmatory examples correctly. The baselines answered 2,704 of 3,000 in-range examples, zero of 6,000 random extrapolation examples, and one of 1,500 carry-chain examples.

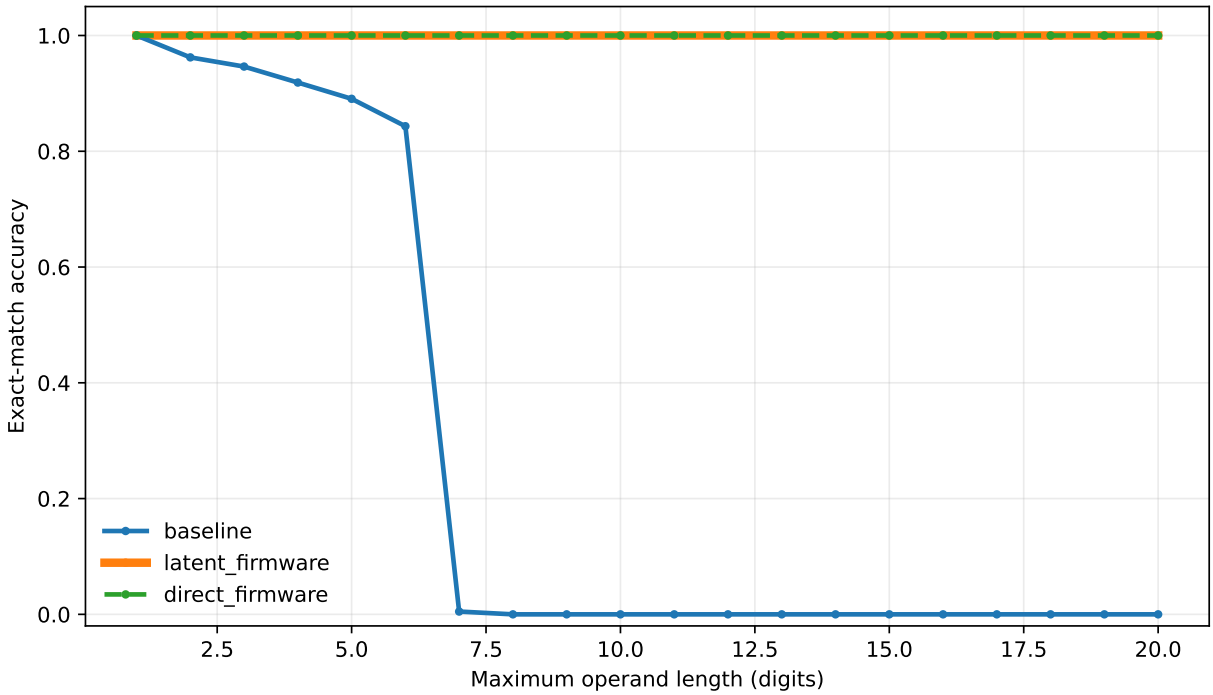

Figure 2: Accuracy grouped by maximum operand length. The generic transformer's performance drops sharply immediately beyond the six-digit training boundary; both firmware paths remain exact through 20 digits.

4.2 Optimization and parameter counts

Every variant had 264,480 trainable parameters. Table 2 reports observed training time. Because the step budgets were intentionally different and the Python firmware parser added overhead, these numbers do not constitute an efficiency comparison.

Table 2: Training budget and mean wall time over three seeds.

Model	Steps	Trainable parameters	Mean wall time
Baseline	10,000	264,480	689.8 s
Latent firmware	1,000	264,480	224.5 s
Direct firmware	1	264,480	0.17 s

4.3 Post-hoc firmware ablation

After confirmatory analysis, firmware strength was set to zero without retraining the three latent models. This was not preregistered and is reported as a post-hoc causal check. In-range exact match fell to 0.2%, 0.4%, and 0.1%; every model scored 0% on OOD primary, OOD long, and carry chains. Thus the confirmatory performance depended on the injected signal rather than an arithmetic algorithm learned incidentally during the 1,000 interface steps.

5 Discussion

5.1 What the experiment establishes

The study supports a limited proposition: exact algorithmic content can be computed by an immutable module, encoded in a fixed latent subspace, and decoded by a small generative transformer across sequence lengths not represented in training. The trainable model did not need to rediscover ripple-carry addition. It learned an interface to a process whose behavior was specified.

The pilot strength failure is equally informative. A deterministic component does not automatically make the hybrid system deterministic. At insufficient signal strength, the learned residual stream overrode the correct latent code at unfamiliar positions and caused premature termination. Reliability belongs to the entire path:

$$P(\text{correct}) = P(\text{parse})\,P(\text{route})\,P(\text{execute})\,P(\text{decode}).$$

This experiment made execution exact and parsing trivial, then tuned the decoding interface until it was reliable over the tested domain. Open-ended mathematics would restore difficult parse and routing terms.

5.2 Why this is not yet mathematical reliability

The model never decided whether addition was appropriate. It received a narrow formal grammar, and a fixed parser extracted the operands. A natural-language word problem could still be formalized incorrectly and then computed perfectly. Likewise, subtraction, multiplication, algebra, units, proof, and uncertainty were not tested.

The observed exactness is empirical over bounded inputs, not a formal end-to-end guarantee. The ripple-carry transition table is exact for valid decimal tokens, but the full implementation includes an autoregressive decoder, finite context, and software execution. "Always mathematically correct" would require typed interfaces, validated routing, explicit preconditions, and possibly proof-carrying outputs.

5.3 Comparison with learned arithmetic methods

The baseline uses fixed sinusoidal positions and ordinary causal attention. Specialized architectures can learn length-generalizing addition [12, 13, 15]. Therefore, Figure 2 is evidence against this generic baseline, not against all learned arithmetic systems.

The distinction is architectural rather than leaderboard-oriented. A learned method may achieve excellent empirical extrapolation and adapt to new operations; a frozen process can preserve a known invariant by construction but needs an explicit representation and interface. A useful future system may combine both: learned decomposition and routing around a library of exact typed primitives.

6 Limitations

1. **Fixed formalization.** Parsing and operation selection were deterministic. The experiment does not test natural language.
2. **Not compiled into standard transformer weights.** The firmware is a PyTorch module using integer tables and Python control flow. Calling it "neural firmware" describes architectural placement and latent integration, not a claim that ripple-carry addition emerged entirely from dense neural weights.
3. **Generic baseline.** No relative-position, position-coupled, scratchpad, Neural GPU, or NALU baseline was implemented.
4. **Single operation and number system.** Only nonnegative base-10 addition was evaluated.
5. **Bounded evaluation.** Operands ended at 20 digits and sequences at 96 tokens.
6. **No retained general language capability.** The small model was trained only on the controlled arithmetic grammar.
7. **Three training seeds.** Replication was exact across all seeds, but the seed-level sample is small.
8. **Unequal training budgets.** The baseline received ten times more steps; this is conservative for accuracy but prevents an efficiency comparison.

7 Next Experiments

The next study should move one boundary at a time:

1. compile the ripple-carry program into frozen tensor operations or transformer weights using a Tracr- or ALTA-like path;
2. replace the fixed grammar parser with a supervised typed encoder and measure parse, route, execute, and decode failures separately;
3. introduce several operations and a learned hard router, including malformed-input rejection;
4. inject the subsystem into a pretrained 100–500M parameter language model and measure general-language retention;
5. compare with strong learned arithmetic baselines such as position coupling;
6. extend the same architecture to stateful deterministic systems, including board-game transition rules and programming-language type constraints.

8 Conclusion

A compact transformer trained to imitate addition generalized poorly beyond its training-length boundary. The same trainable architecture, given an immutable ripple-carry process through a sufficiently strong fixed latent code, was exact across every tested length and seed. This does not solve mathematical reasoning, but it validates the first architectural step: known deterministic

processes can be treated as internal computational structure rather than facts that gradient descent must repeatedly approximate. It does not establish the first integrated calculator in a language model; IGC is direct prior art. Its contribution is the controlled small-transformer evidence and exact out-of-range comparison reported here.

Acknowledgments and Attribution

Josiah Wilson originated the research hypothesis: deterministic processes could be installed as locked components inside generative neural models, with mathematics serving as the first controlled test. OpenAI Codex assisted with literature discovery, experimental design, implementation, execution, analysis, visualization, and manuscript drafting under Wilson's direction. All claims in this report are limited to the committed artifacts and observed results.

References

[1] T. B. Brown et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. https://arxiv.org/abs/2005.14165.

[2] L. Kaiser and I. Sutskever. Neural GPUs learn algorithms. arXiv:1511.08228, 2015. https://arxiv.org/abs/1511.08228.

[3] K. Freivalds and R. Liepins. Improving the Neural GPU architecture for algorithm learning. arXiv:1702.08727, 2017. https://arxiv.org/abs/1702.08727.

[4] A. Trask, F. Hill, S. Reed, J. Rae, C. Dyer, and P. Blunsom. Neural arithmetic logic units. *Advances in Neural Information Processing Systems*, 31, 2018. https://arxiv.org/abs/1808.00508.

[5] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. H. S. Torr. Adaptive neural compilation. *Advances in Neural Information Processing Systems*, 29, 2016. https://arxiv.org/abs/1605.07969.

[6] P. Velickovic and C. Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021. https://arxiv.org/abs/2105.02761.

[7] G. Weiss, Y. Goldberg, and E. Yahav. Thinking like transformers. In *Proceedings of the 38th International Conference on Machine Learning*, 2021. https://arxiv.org/abs/2106.06981.

[8] D. Lindner, J. Kramar, S. Farquhar, M. Rahtz, T. McGrath, and V. Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36, 2023. https://arxiv.org/abs/2301.05062.

[9] P. Shaw, J. Cohan, J. Eisenstein, K. Lee, J. Berant, and K. Toutanova. ALTA: Compiler-based analysis of transformers. arXiv:2410.18077, 2024. https://arxiv.org/abs/2410.18077.

[10] L. Saldyt and S. Kambhampati. Algorithmic language models with neurally compiled libraries. arXiv:2407.04899, 2024. https://arxiv.org/abs/2407.04899.

[11] W. Bounsi, B. Ibarz, A. Dudzik, J. B. Hamrick, L. Markeeva, A. Vitvitskyi, R. Pascanu, and P. Velickovic. Transformers meet neural algorithmic reasoners. arXiv:2406.09308, 2024. https://arxiv.org/abs/2406.09308.

[12] S. Jelassi, S. d'Ascoli, C. Domingo-Enrich, Y. Wu, Y. Li, and F. Charton. Length generalization in arithmetic transformers. arXiv:2306.15400, 2023. https://arxiv.org/abs/2306.15400.

[13] H. Cho, J. Cha, P. Awasthi, S. Bhojanapalli, A. Gupta, and C. Yun. Position coupling: Improving length generalization of arithmetic transformers using task structure. arXiv:2405.20671, 2024. https://arxiv.org/abs/2405.20671.

[14] H. Cho, J. Cha, S. Bhojanapalli, and C. Yun. Arithmetic transformers can length-generalize in both operand length and count. arXiv:2410.15787, 2024. https://arxiv.org/abs/2410.15787.

[15] S. Duan, Y. Shi, and W. Xu. From interpolation to extrapolation: Complete length generalization for arithmetic transformers. arXiv:2310.11984, 2023. https://arxiv.org/abs/2310.11984.

[16] T. Vergara-Browne and A. Soto. Tracr-Injection: Distilling algorithms into pre-trained language models. arXiv:2505.10719, 2025. https://arxiv.org/abs/2505.10719.

[17] L. Sheneman. The Neural Compiler: Program-to-network translation for hybrid scientific machine learning. arXiv:2605.22498, 2026. https://arxiv.org/abs/2605.22498.

[18] F. Dietz and D. Klakow. IGC: Integrating a Gated Calculator into an LLM to Solve Arithmetic Tasks Reliably and Efficiently. 2025. https://arxiv.org/abs/2501.00684.

A Reproducibility Checklist

- Frozen confirmatory configuration: `configs/study.json`.
- Configuration commit: `22f1649e28079ce28620ac8c01b774d8009231cd`.
- Configuration SHA-256: `6a2d375e789f6eaabc5cfa5d295ab9eb6752e040afe25aaf357b9425f29b33c1`.
- Evaluation logical SHA-256: `917935e56a1d1aecf72589c11721a695f1092066c9b97b875e1a49622393bd9b`.
- Environment lock: `uv.lock`.
- Unit tests: eight passing tests at confirmatory execution.
- Compact run manifests, errors, summaries, and figures: `results/` and `figures/`.

Core reproduction commands:

```
uv sync --extra dev
uv run pytest
uv run nf-study --config configs/study.json
uv run nf-analyze \
  --study artifacts/confirmatory_v1/study.json
```

B Pilot-to-Confirmatory Separation

Pilots used training seed 17 and evaluation seed 260725. The initial latent strength of 8 decoded correctly in distribution but failed at unseen positions. A pilot-only sweep established that strengths of 20 or more removed observed truncation. The final strength of 32 was frozen before confirmatory training. Confirmatory seeds 101, 211, and 307 and evaluation seed 918273 were not used during tuning. The complete chronological record is in `PILOT_LOG.md`.

Artifact scope. Large model checkpoints and full per-example predictions are retained locally under ignored artifact directories. The Git-tracked artifact contains compact study manifests, every baseline error, aggregate tables, environment records, source code, configuration, figures, and this manuscript.