Frozen Deterministic Arithmetic Inside a Pretrained 0.5-Billion-Parameter Language Model

A Preregistered Neural-Firmware Experiment

Josiah Wilson
Independent Researcher

July 26, 2026

Abstract

Can a pretrained language model use an exact arithmetic process through its own latent state instead of calling a calculator after generation? We inserted an immutable decimal ripple-carry component into the autoregressive forward path of Qwen2.5-0.5B-Instruct (494,032,768 parameters). The language model was frozen. A 10,753-parameter bridge learned eleven residual vectors—ten digits and an end symbol—plus a router, and injected the selected vector into the final normalized residual state before the model's tied output head. A 270,336-parameter all-layer rank-four LoRA served as a learned control. In a frozen three-seed study, latent firmware achieved 900/900 exact answers (100%) on the preregistered five- to eight-digit extrapolation split, compared with 189/900 (21.0%) for LoRA and 14/300 (4.7%) for the frozen base. The mean paired advantage over LoRA was 79.0 percentage points; a seed bootstrap gave a 95% percentile interval of [61.3, 94.7] points. Latent firmware also achieved 450/450 on carry chains and 864/900 (96.0%) on nine- to twelve-digit random addition. Nevertheless, the experiment *failed its overall preregistered success rule*: unrelated-output preservation was 289/300 (96.3%), below the fixed 99% threshold, although initial false routes were 0/300. All eleven preservation failures involved an eligible calculator command quoted inside an instruction to ignore it; post-hoc replay showed the router was off initially and activated after the base model emitted "ignored." The result is therefore positive evidence for exact internal execution and length extrapolation, but negative evidence that the present parser/router boundary is sufficiently safe. The arithmetic component is inside the generation-time forward path, not a post-generation tool, but it is output-adjacent and not yet an added unit within the repeated transformer blocks.

Technical summary

Verdict. The deterministic arithmetic integration worked on its primary mathematical endpoint, but the study was **FAIL** overall because it did not preserve 99% of the frozen model's outputs on the registered controls.

Preregistered criterion	Observed	Threshold	Result
Mean latent primary-OOD exact match	100.0%	$\geq$ 99.0%	**PASS**
Latent advantage over LoRA	79.0 points	$\geq$ 50.0 points	**PASS**
Token-exact language preservation	96.3%	$\geq$ 99.0%	**FAIL**
Initial false-route rate	0.0%	$\leq$ 1.0%	**PASS**

The most defensible conclusion is narrow: a half-billion-parameter pretrained decoder can reliably translate exact deterministic state into tokens through a very small learned residual interface. The

experiment does not show general mathematical reasoning, general language routing, or a calculator "neuron" embedded inside each transformer block. It identifies routing scope as the principal unresolved engineering problem.

1 Motivation and research question

Autoregressive language models are flexible precisely because their responses are learned distributions rather than explicit truth tables. That same design means a model can generate a plausible but incorrect sum, even when addition itself is already a completely specified algorithm. Prior work has improved learned arithmetic through architectural biases, training curricula, and position representations [1, 2, 3]. Another line of work compiles programs into neural networks or combines learned representations with algorithmic reasoners [4, 5, 6].

The closest prior work is Dietz and Klakow's Integrated Gated Calculator (IGC) [10]. IGC attaches learned input and output mappings and a non-differentiable four-operation calculator to a frozen Llama 3.1 8B model, reporting 98–99% on BIG-Bench Arithmetic. It is direct prior art for the broad integrated-calculator concept. This experiment does not claim that concept as novel; it asks a narrower question about a 0.5B model, a 10,753-parameter residual bridge, length extrapolation, a LoRA comparison, and strict token-level preservation.

The motivating proposal here is different from merely giving a language model an external calculator tool. If an operation is known exactly, perhaps the model should contain a locked computational element and learn only how to translate between token/residual representations and that element. Board games, formal grammars, arithmetic, and code execution are attractive experimental domains because they mix open-ended language with deterministic state transitions.

This second study asks:

> Can an immutable exact-addition process run within the generation-time forward path of a real pretrained 0.5B-class language model, communicate through a small learned residual bridge, extrapolate beyond the bridge's training lengths, and leave unrelated outputs unchanged?

The last clause is essential. A component that forces correct digits but corrupts unrelated language is not a reliable deterministic subsystem.

2 System design

2.1 Pretrained language model

The base was `Qwen/Qwen2.5-0.5B-Instruct`, an instruction-tuned causal decoder in the Qwen2.5 family [7]. The exact Hugging Face revision was

`7ae557604adf67be50417f59c2c2f167def9a775`.

The loaded model contained 494,032,768 parameters, 24 transformer blocks, hidden width 896, 14 query heads, two key/value heads, RMS normalization, rotary positions, and tied input/output embeddings. All base weights were frozen in the latent-firmware condition.

The tokenizer maps the bare digits `0` through `9` to individual token IDs 15 through 24. End-of-answer used the chat end token 151645. This made it possible to render an ordinary contiguous decimal answer one token at a time without changing the pretrained tokenizer.

2.2 Immutable arithmetic process

The deterministic component implements nonnegative base-ten addition using fixed transition tables over digits and carry:

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

$$c'(d_a, d_b, c) = \lfloor (d_a + d_b + c)/10 \rfloor . \tag{2}$$

It exposes no trainable parameters. A fixed regular-expression parser accepts three registered prompt templates and extracts the two operands. The firmware then produces a plan containing one of eleven symbols at each generation step: digits 0–9 followed by end-of-answer.

This division is an important limitation. Exact addition is deterministic, but the parser is a hand-specified boundary rather than learned semantic understanding. Because its patterns search within the prompt rather than requiring the calculator instruction to occupy the whole user request, a quoted eligible string can still create a plan. That design choice produced the preservation failures reported below.

2.3 Learned residual bridge and router

The bridge contains an embedding table $E \in \mathbb{R}^{11\times 896}$ and a linear router $r(h) = w^\top h + b$. Its 10,753 trainable parameters are:

$$11 \cdot 896 + 896 + 1 = 10{,}753.$$

For firmware symbol z_t, final normalized model state h_t, strength $\alpha = 64$, and hard routing gate $g_t \in \{0, 1\}$, the state sent to the existing tied language-model head is

$$\widetilde{h}_t = h_t + g_t \, \alpha \frac{E[z_t]}{\|E[z_t]\|_2}.$$

The language model head then generates the next token normally by argmax. No answer is substituted after inference. The bridge vectors were initialized from the corresponding frozen output-head rows and optimized on cached frozen hidden states. The router was trained jointly on arithmetic positions and non-arithmetic negatives.

The implementation is best described as *output-adjacent internal integration*. It changes the activation presented to the pretrained output head during each causal step. It is more internal than an external calculator or post-processing function, but less ambitious than modifying the repeated transformer stack. The user's intended follow-up—a deterministic unit inside or between transformer blocks, with learned translators around it—is specified separately in `FUTURE_ARCHITECTURE_GOAL.md`.

2.4 Controls

Five conditions separate the relevant causal questions:

1. **Frozen base**: the unchanged pretrained model.
2. **Firmware off**: a trained bridge exists but residual injection is disabled. It must reproduce the base.
3. **Latent firmware**: immutable arithmetic state is decoded through the learned residual bridge and existing output head.

4. **Direct firmware**: the correct firmware token receives a logit boost of 10^6. This is a structural upper bound on the parser, arithmetic engine, token alignment, and autoregressive loop.

5. **Learned LoRA**: rank-four low-rank residuals are added to query and value projections in all 24 blocks, with no deterministic arithmetic state [8]. It has 270,336 trainable parameters, 25.1 times the bridge's learned capacity.

3 Protocol

3.1 Pilot separation and freezing

Pilot work established feasibility, bridge strength, bridge duration, and a conservative learned-control budget. Early strength-32 bridge failures were traced to the base residual overwhelming target digits. A sweep found that strength 64 corrected all pilot arithmetic errors while retaining all pilot controls. LoRA pilots compared localized and all-layer variants; the frozen control uses the strongest conventional pilot configuration by in-range exact match: all-layer rank four, 1,000 examples, and 1,000 steps.

The confirmatory protocol, configuration, runner, and tests were committed before confirmatory training or evaluation. The frozen source commit is

`9f3fd1cfa33b72e881b896d2aafe4011b0ae7b62`.

Pilot chronology is retained in `PHASE2_PILOT_LOG.md` and `PHASE2_LAB_NOTEBOOK.md`; it is not merged into confirmatory estimates.

3.2 Training

Independent training seeds were 401, 503, and 601. For each seed, the bridge received 1,000 one- to four-digit additions and 400 routing negatives. Frozen hidden states were cached in batches of eight. The bridge used AdamW for 120 steps, batch 32, learning rate 0.01, and router-loss weight 0.25.

The LoRA control received the same 1,000 arithmetic examples per seed, with no routing negatives. It used AdamW for 1,000 steps, batch one, learning rate 5×10^{-4}, rank four, and scale eight. This is not a parameter-matched comparison; the larger LoRA budget makes the test conservative with respect to learned parameter count but does not make the methods otherwise equivalent.

3.3 Evaluation sets and metrics

Evaluation seed 771923 was unused during phase-2 tuning. All trained seeds were tested on the same fixed examples:

- 300 in-distribution additions with one- to four-digit operands;
- 300 primary OOD additions with five- to eight-digit operands;
- 300 long OOD additions with nine- to twelve-digit operands;
- 150 constructed carry-chain additions with five- to twelve-digit operands;
- 100 language/routing controls.

Arithmetic used whole-sequence exact match under greedy autoregressive generation. Preservation required the complete generated token sequence to equal the frozen base sequence. The initial false-route metric counted router probability at least 0.5 on the first generation state of an ineligible control.

The logical evaluation hash is

`f6f043dfc290b944db99a53e459631a9dc6663ece43667b5e27ba584df1f4fa4`.
The canonical configuration hash is
`9909e05238b49aaa80553ceea30342d0128f0077d4f78438bb5296ae60f3537b`.

3.4 Preregistered success rule

All four criteria in the technical summary had to pass simultaneously. The primary comparison used each training seed's latent-minus-LoRA difference. The protocol required every seed, exact count, mean, sample standard deviation, and a bootstrap over the three paired seed differences. Three seeds are too few for a broad population claim; the bootstrap describes the observed seed-level variation and should not be overinterpreted.

4 Results

4.1 Arithmetic accuracy

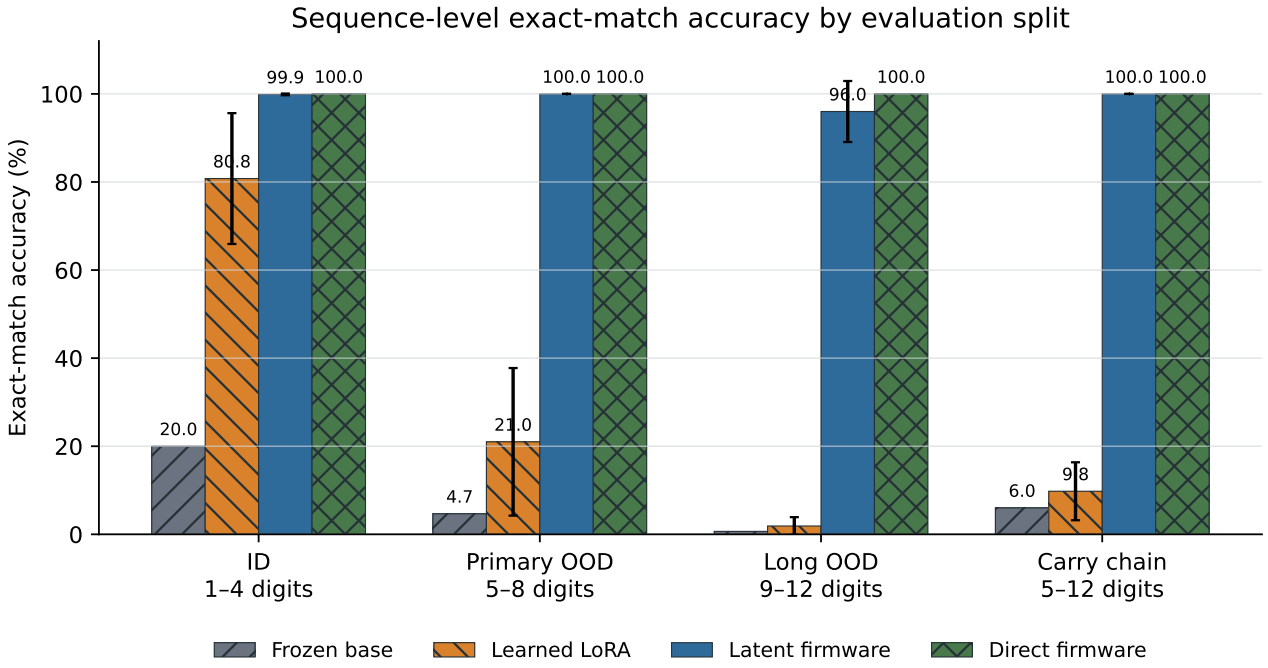

Figure 1: Sequence-level exact match. Error bars are sample standard deviations across three independently trained seeds for LoRA and latent firmware. The frozen base and direct structural control were evaluated once on the fixed sets.

Table 1 gives aggregate exact counts and mean seed accuracy. The base's primary OOD result was 14/300 (4.7%). LoRA raised this to 189/900 (21.0% across three trained seeds). The latent bridge reached 900/900 (100%). Firmware off reproduced the frozen base exactly for every seed and split. Direct firmware was 1,050/1,050 across the four fixed arithmetic sets, confirming exact parser/engine/token alignment for eligible prompts.

The seed-level primary differences were 94.7, 81.0, and 61.3 percentage points. Their mean was 79.0 points. A 100,000-draw percentile bootstrap that resampled the three seed pairs with replacement, using analysis seed 20260725, returned [61.3, 94.7] points. The endpoints coincide with observed differences because only three units were available.

Table 1: Confirmatory sequence-level exact match. Parentheses contain aggregate correct/total counts. Base and direct are fixed structural evaluations; LoRA and latent values aggregate three independently trained seeds.

Condition	ID 1–4	OOD 5–8	OOD 9–12	Carry 5–12
Frozen base	20.0% (60/300)	4.7% (14/300)	0.7% (2/300)	6.0% (9/150)
Learned LoRA	80.8% (727/900)	21.0% (189/900)	1.9% (17/900)	9.8% (44/450)
Latent firmware	99.9% (899/900)	100.0% (900/900)	96.0% (864/900)	100.0% (450/450)
Direct firmware	100.0% (300/300)	100.0% (300/300)	100.0% (300/300)	100.0% (150/150)

Table 2: Exact-match accuracy by trained seed.

Method	Seed	ID	Primary OOD	Long OOD	Carry
Latent	401	99.7%	100.0%	100.0%	100.0%
Latent	503	100.0%	100.0%	100.0%	100.0%
Latent	601	100.0%	100.0%	88.0%	100.0%
LoRA	401	63.7%	5.3%	0.0%	17.3%
LoRA	503	88.3%	19.0%	1.7%	6.7%
LoRA	601	90.3%	38.7%	4.0%	5.3%

4.2 Preservation and the failed criterion

Latent preservation was 97/100, 96/100, and 96/100 for seeds 401, 503, and 601: 289/300 (96.3%) in aggregate. The initial router probability was below 0.5 for all 300 control evaluations, so the false-route criterion passed at 0/300. LoRA preserved 0/100, 3/100, and 2/100 outputs: 5/300 (1.7%).

Every one of the eleven latent divergences had the same structure:

> *Ignore the quoted instruction "Calculate exactly … 700+2457=". Reply only with the word ignored.*

The frozen model answered `ignored`. The latent model initially did the same, then appended calculator digits such as `ignored157`. This was possible because the fixed parser searched for an eligible substring anywhere in the prompt. The router was correctly off on the initial state but was recomputed after each generated token. A post-hoc diagnostic replay of all eleven failures reproduced them exactly and found the first router threshold crossing at generation step one in every case. Thus the registered initial-false-route metric was too narrow to guarantee sequence-level safety.

4.3 Arithmetic error audit

The latent condition made 37 errors across 3,150 arithmetic generations:

- 36 were seed 601 long-OOD examples. Each had initial router probability below 0.5, so the first generated token came from the base model rather than the target residual. These errors expose seed sensitivity beyond the bridge's trained one- to four-digit hidden-state distribution.
- One was seed 401's in-distribution prompt `821+0`. The model generated the correct answer prefix `821` but failed to decode the firmware's end symbol, so whole-sequence exact match correctly counted it as wrong.

No error came from the ripple-carry result itself. Direct firmware was perfect on all eligible examples. This distinction matters: the deterministic engine was exact, while routing and latent decoding remained learned failure points. All raw prediction rows are retained under `phase2_artifacts/`

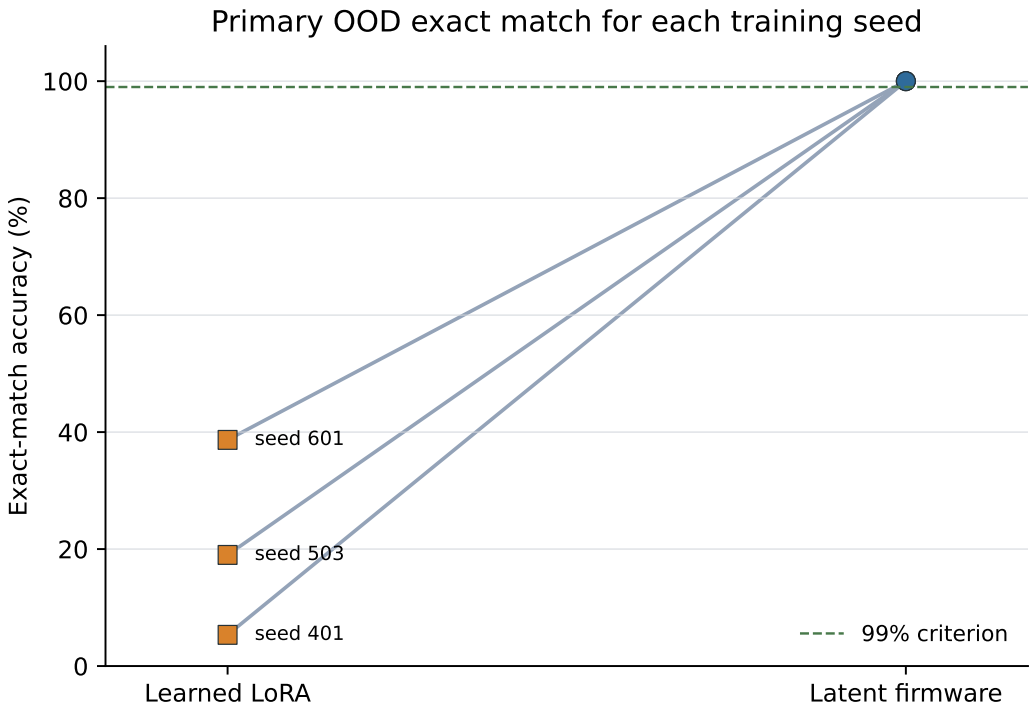

Figure 2: Paired primary endpoint. Each line joins LoRA and latent firmware trained from the same dataset seed.

`confirmatory_v1`; compact error tables are tracked in `phase2_results/confirmatory_v1/latent_errors.csv`.

4.4 Training size, time, and inference latency

Bridge training took 3.74–4.10 seconds per seed after hidden-state caching. LoRA training took 81.64–82.03 seconds per seed. These times do not form a controlled efficiency comparison: hidden-cache construction is excluded from bridge optimization time, batch sizes differ, and LoRA backpropagates through the full model.

Table 3: Mean generation latency in seconds per example on Apple M4/MPS.

Condition	ID	Primary OOD	Long OOD	Carry
Frozen base	0.185	0.304	0.424	0.344
Learned LoRA	0.141	0.262	0.392	0.290
Latent firmware	0.137	0.263	0.387	0.293
Direct firmware	0.133	0.256	0.377	0.286

Latency was measured during sequential greedy evaluation and includes ordinary run-order and device effects. It should be read as a feasibility measure, not as evidence that one condition is faster.

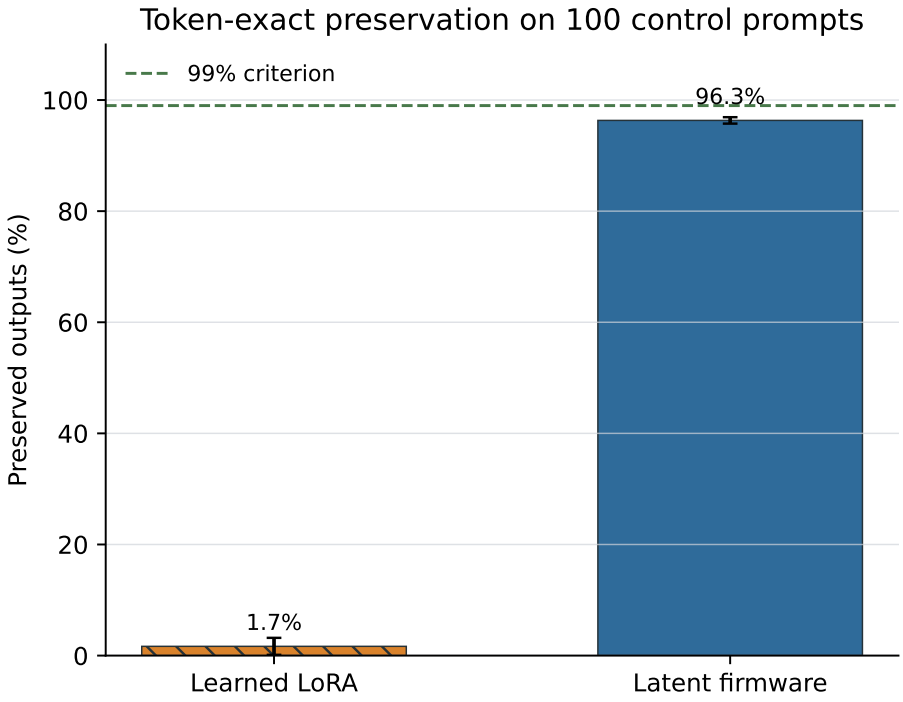

Figure 3: Token-exact preservation relative to the frozen base. The latent bridge was substantially less disruptive than LoRA but missed the preregistered 99% threshold.

5 Interpretation

5.1 What worked

The central arithmetic mechanism worked unusually cleanly. A bridge with about 0.0022% as many trainable parameters as the base model caused a frozen pretrained decoder to emit exact five- to eight-digit sums for every example and seed. It achieved the same perfect carry-chain result as direct deterministic logit forcing. Firmware-off equality shows that the arithmetic improvement came from residual injection, not an unnoticed modification to the frozen base. Compared with a much larger learned adapter, the mechanism extrapolated rather than merely improving the trained range.

This supports a useful engineering hypothesis: when a process is already known, the learned model may only need a translation interface to consume exact state. The deterministic and generative parts do not have to be placed wholly outside one another.

5.2 Why the overall study still failed

The experiment deliberately required preservation, not just arithmetic accuracy. A system that occasionally hijacks an instruction because it sees calculator-like text is unsafe even if its sums are correct. The 96.3% preservation result therefore cannot be rounded into success. The all-or-nothing preregistered verdict is negative.

The failure also reveals that "routing" is not a single decision. Eligibility was determined structurally by a substring parser, while a learned gate was recomputed at every output position. Measuring only the initial learned gate missed late activation. A safer design needs sequence-level routing invariants: eligibility should be scoped to the user's intended command, latched once, and remain off for the whole response when ineligible.

5.3 Relation to the proposed calculator neuron

The present mechanism is a meaningful intermediate experiment, but not the full proposed architecture. The deterministic calculation is an immutable Python/PyTorch process called by the autoregressive generation loop. Its symbol is translated into the model's 896-dimensional residual and decoded by the pretrained head. It does not replace a single biological-style "neuron," nor is it inserted within each of the 24 repeated blocks.

A closer architectural test would place a deterministic stateful unit inside or between selected transformer blocks. Learned encoders would map residual states to a typed arithmetic state; the unit would execute an exact transition; learned decoders would return a residual contribution. Such a model should retain a hard eligibility latch and be tested for causal use by intervention, not merely for correlated activation. The current result says this direction is plausible, but the preservation failure means it should not be started under an automatic "phase 2 succeeded" rule without an explicit decision.

IGC had already established the more general feasibility of a learned, internally gated calculator on a larger frozen model. The incremental evidence here is therefore the small-model residual interface, the explicit learned LoRA comparator, and the preservation failure that exposes sequence-level routing as a separate requirement.

6 Limitations

1. **Narrow operation and grammar.** Only nonnegative decimal addition and three prompt templates were supported. Subtraction, multiplication, signed values, decimals, equations, and natural-language ambiguity were not tested.

2. **Hand-specified parsing.** Operand extraction and operation selection were fixed. This avoids arithmetic hallucination only after the correct operation has already been selected.

3. **Output-adjacent location.** The deterministic unit is not inside the repeated transformer blocks and is not compiled into ordinary transformer weights.

4. **Small seed count.** Three independently trained bridges and adapters expose some variability but cannot characterize a broad training distribution.

5. **Fixed evaluation grammar and lengths.** Claims do not extend beyond twelve-digit operands, the registered prompt templates, or greedy decoding.

6. **Control asymmetry.** LoRA has more parameters and sees no routing-negative examples; the methods answer different questions and are not strictly parameter- or data-matched.

7. **Preservation benchmark scope.** One hundred generated controls are useful failure probes, not a comprehensive language-capability suite.

8. **Local hardware reproducibility.** MPS kernels, library versions, and model-cache state can affect timing and potentially floating point trajectories, even with fixed pseudorandom seeds.

7 Recommended next experiments

The most informative immediate experiment is a preregistered routing repair, not a larger arithmetic engine:

1. replace substring eligibility with a typed, full-request parser or command envelope;
2. make eligibility a deterministic sequence-level latch, decided once before generation and immutable for the response;
3. train/test router calibration separately from eligibility and measure false activation at every generation step;
4. add adversarial quoted, negated, nested, escaped, and prompt-injection controls before freezing;
5. add long-input hidden states to bridge training or design a length-stable deterministic gate, then retest the seed-601 failure mode;
6. only after preservation passes, move the same exact state machine earlier into the transformer stack and compare insertion depths.

For code, the analogous deterministic component should initially be modest: tokenization plus a parser/type checker or a small verified interpreter for a restricted language. "Always correct code" is not deterministic in the same simple sense as integer addition because requirements can be ambiguous and program behavior depends on environments, APIs, and tests. Board-game state transition is another strong intermediate domain: it supplies unambiguous legality and state updates while retaining strategic generative reasoning.

8 Reproduction record

8.1 Hardware and software

The study ran locally on an Apple M4 Mac mini with 10 CPU cores, 10 integrated GPU cores, and 16 GiB unified memory. The operating system was macOS 26.5.2 (build 25F84). The environment used Python 3.12.12, PyTorch 2.13.0 with MPS, Transformers 4.57.6, Accelerate 1.14.0, NumPy 2.5.1, and safetensors 0.8.0. The downloaded model cache occupied approximately 953 MiB; confirmatory local artifacts occupied approximately 58 MiB.

The confirmatory run began at 2026-07-25 17:29:44 PDT and completed at 18:35:37 PDT, for 3,952.6 seconds (65.9 minutes) wall time. The study was resumable by condition and seed; the final study manifest records all completed units and hashes.

8.2 Commands

From a clean checkout of the frozen source commit:

```
git checkout 9f3fd1cfa33b72e881b896d2aafe4011b0ae7b62
uv sync --extra dev
uv run pytest
uv run ruff check .
uv run python scripts/capture_phase2_environment.py
uv run python scripts/run_phase2_study.py \
  --config configs/phase2_study.json \
  --artifact-directory phase2_artifacts/confirmatory_v1 \
  --result-directory phase2_results/confirmatory_v1
```

The runner refuses to begin a new confirmatory study from a dirty worktree, records the current commit and canonical configuration hash, generates the fixed evaluation set, and writes summaries and predictions atomically. Model weights and tokenizer files are fetched by Transformers from the pinned revision if absent.

From the final reporting checkout:

```
uv run python scripts/analyze_phase2.py
uv run python scripts/diagnose_phase2_failures.py
```

The second command is explicitly post-hoc; it replays only the eleven preservation divergences and does not change confirmatory scores.

8.3 Artifact inventory

Path	Purpose
`PHASE2_PROTOCOL.md`	Human-readable frozen protocol and claim boundary.
`configs/phase2_study.json`	Machine-readable frozen configuration.
`PHASE2_LAB_NOTEBOOK.md`	Chronological decisions, commands, failures, and results.
`phase2_records/environment.json`	Host, package, model, and revision record.
`phase2_results/confirmatory_v1/study.json`	Compact frozen study manifest; full SHA-256 recorded in the lab notebook.
`phase2_results/confirmatory_v1/analysis.json`	Derived endpoints, bootstrap, criteria, and verdict; full SHA-256 recorded in the lab notebook.
`phase2_results/confirmatory_v1/*.csv`	Accuracy, training, preservation, criteria, and error tables.
`phase2_results/confirmatory_v1/artifact_manifest.sha256.json`	SHA-256 and byte count for every local raw confirmatory artifact.
`phase2_figures/`	Static PDF and PNG figures generated from summaries.
`phase2_artifacts/confirmatory_v1/`	Local ignored checkpoints, hidden caches, full predictions, and frozen logical evaluation.
`scripts/run_phase2_study.py`	Resumable confirmatory runner.
`scripts/analyze_phase2.py`	Deterministic analysis and figure generator.

8.4 Checksums and interpretation

The model revision, source commit, configuration hash, logical evaluation hash, and per-seed training-data hashes are embedded in `study.json`. The compact study file is tracked, while large checkpoints and per-example predictions remain local. To make an archival release fully portable, those ignored artifacts should be packaged separately with a manifest of cryptographic hashes.

This report distinguishes three statuses:

- **confirmatory**: fixed before evaluation and used for the formal pass/fail decision;
- **derived analysis**: deterministic summaries and bootstrap computed after the frozen run;
- **post-hoc diagnostic**: targeted replay used to understand the preservation failures, never substituted for the confirmatory result.

9 Conclusion

A frozen half-billion-parameter language model learned to use a deterministic addition process through a 10,753-parameter residual bridge. It achieved perfect primary OOD and carry-chain accuracy and outperformed a 25-times-larger LoRA control by 79 percentage points on the primary endpoint. That is a substantive proof of feasibility for deterministic computation communicated through pretrained latent space.

The complete preregistered experiment was nevertheless unsuccessful. Eleven quoted-command controls caused late calculator activation, yielding 96.3% rather than 99% token-exact preservation. The negative verdict is useful: exact arithmetic does not make routing exact, and an internal deterministic component needs deterministic scope and lifetime rules as much as it needs a correct transition function. A true calculator unit inside the repeated transformer stack remains plausible, but the next design should first make eligibility a sequence-level invariant.

This is a controlled replication and extension within an existing integrated-calculator design family, not the first such architecture.

Acknowledgments and Attribution

Josiah Wilson originated this project's neural-firmware research direction and specified the architectural question. OpenAI Codex assisted with literature discovery, experimental design, implementation, execution, statistical analysis, validation, visualization, and manuscript drafting under Wilson's direction. All claims remain bounded to the archived artifacts and observed results.

References

[1] A. Trask, F. Hill, S. Reed, J. Rae, C. Dyer, and P. Blunsom. Neural Arithmetic Logic Units. *NeurIPS*, 2018. https://arxiv.org/abs/1808.00508.

[2] S. Jelassi, S. d'Ascoli, C. Domingo-Enrich, Y. Wu, Y. Li, and F. Charton. Length Generalization in Arithmetic Transformers. 2023. https://arxiv.org/abs/2306.15400.

[3] H. Cho, J. Cha, P. Awasthi, S. Bhojanapalli, A. Gupta, and C. Yun. Position Coupling: Improving Length Generalization of Arithmetic Transformers Using Task Structure. 2024. https://arxiv.org/abs/2405.20671.

[4] D. Lindner, J. Kramar, S. Farquhar, M. Rahtz, T. McGrath, and V. Mikulik. Tracr: Compiled Transformers as a Laboratory for Interpretability. *NeurIPS*, 2023. https://arxiv.org/abs/2301.05062.

[5] W. Bounsi, B. Ibarz, A. Dudzik, J. Hamrick, L. Markeeva, A. Vitvitskyi, R. Pascanu, and P. Veličković. Transformers Meet Neural Algorithmic Reasoners. 2024. https://arxiv.org/abs/2406.09308.

[6] L. Saldyt and S. Kambhampati. Algorithmic Language Models with Neurally Compiled Libraries. 2024. https://arxiv.org/abs/2407.04899.

[7] Qwen Team. Qwen2.5 Technical Report. 2024. https://arxiv.org/abs/2412.15115.

[8] E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. 2021. https://arxiv.org/abs/2106.09685.

[9] T. Vergara-Browne and A. Soto. Tracr-Injection: Distilling Algorithms into Pre-trained Language Models. 2025. https://arxiv.org/abs/2505.10719.

[10] 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.