TECHNICAL RESEARCH REPORT

The Case for Deterministic Computation

in Tool-Augmented Language Models

A literature review, competitive analysis, and empirical benchmark of Arithym — *an exact-arithmetic MCP server for AI agents*

Geometric Data Systems, Inc.

March 2026

arithym.xyz

Abstract

Large language models fail catastrophically at multi-digit arithmetic, achieving under 6% accuracy on four-digit multiplication and compounding errors exponentially through multi-step chains. Chain-of-thought reasoning does not meaningfully improve these results. Code generation tools restore accuracy but impose significant latency (seconds per call), token overhead (code generation as expensive output tokens), and floating-point precision risks. This report reviews 18 peer-reviewed publications from NeurIPS, ICML, ACL, and ICLR (2022–2025), benchmarks a three-way comparison across 18 computation tasks, and maps the competitive landscape. We present evidence that a purpose-built deterministic computation layer—delivered via the Model Context Protocol as named tool calls with exact arithmetic and zero floating-point exposure—eliminates math hallucination while reducing cost by 61–67% and latency by two orders of magnitude relative to current alternatives. This layer can operate standalone or complement existing tools like Wolfram and Code Interpreter by handling high-frequency computation at lower cost.

1. Introduction

The deployment of LLMs as autonomous agents—executing multi-step workflows, managing financial calculations, and making decisions based on numerical reasoning—has exposed a fundamental limitation: **language models cannot reliably perform arithmetic.** This is not a minor edge case. Published research demonstrates that the most capable models achieve near-zero accuracy on multiplication problems that any pocket calculator handles instantly.

The research community has converged on a clear conclusion: tool augmentation is not optional for math-capable AI agents. The question is no longer *whether* to offload computation, but *how.* Current approaches—chain-of-thought reasoning, Python code generation, and general-purpose computational engines—each carry significant trade-offs in cost, latency, accuracy, and reliability.

This report examines those trade-offs through three lenses: a review of peer-reviewed literature on LLM arithmetic failures and tool augmentation (Section 2), an empirical benchmark comparing three computation strategies across 18 tasks (Section 3), and a competitive analysis of existing solutions (Section 4). We then synthesize defensible claims for practitioner audiences (Section 5) and identify the market positioning that emerges from the evidence (Section 6).

2. Literature Review

2.1 LLM Arithmetic Accuracy Is Fundamentally Limited

Multiple independent research groups have measured LLM arithmetic accuracy with controlled experiments, and the results are consistent: accuracy degrades sharply with operand size and collapses entirely beyond a few digits.

Qiu et al. (2024) tested GPT-4 on 100 trials per digit-pair combination using standardized prompts, finding **6% accuracy on 4×4 digit multiplication and 0% on 5×5.** The MathGLM paper (Yang et al., 2023) independently measured GPT-4's multi-digit multiplication accuracy at 4.3% overall. The Goat paper (Liu & Low, 2023) reported accuracy *"almost identically zero on tasks involving large numbers."*

These failures are not bugs to be patched. The NeurIPS 2023 spotlight paper *Faith and Fate* (Dziri et al.) provided a theoretical proof that the probability of incorrect predictions converges exponentially to 1 as compositional complexity increases. The paper demonstrated that transformers solve compositional tasks via "linearized subgraph matching"—a form of pattern recognition that mimics systematic reasoning on simple cases but breaks down as problems scale.

Table 1. Published LLM Multiplication Accuracy

Task	GPT-4	ChatGPT	Source
3×3 digit multiply	59%	55%	Dziri et al., 2023
4×4 digit multiply	4–6%	—	Qiu et al., 2024
5×5 digit multiply	0%	—	Qiu et al., 2024
Multi-digit overall	4.3%	—	Yang et al., 2023
Large number tasks	≈0%	—	Liu & Low, 2023

A 2025 follow-up demonstrated that systems with even a 1% per-step error rate fail after 100 sequential steps. The GSM-Ranges paper (2025) showed that logical error rates increase up to 14 percentage points as numerical complexity rises, even with chain-of-thought active.

2.2 Chain-of-Thought Does Not Fix Arithmetic

Chain-of-thought prompting is the most widely deployed technique for improving LLM reasoning. It demonstrably helps with logical decomposition and word-problem comprehension. However, for raw arithmetic, the evidence is clear: **CoT provides only marginal improvement.**

The Goat paper explicitly tested CoT on GPT-4 arithmetic and found only "marginal improvement." This result is intuitive: CoT helps the model decompose a word problem into sub-steps, but each sub-step still requires the model to perform arithmetic it cannot reliably execute. The technique redistributes the problem without solving the underlying failure.

Moreover, CoT is expensive. The TALE paper (Han et al., ACL 2025) demonstrated that approximately **two-thirds of chain-of-thought tokens are redundant**—the model can achieve equivalent accuracy with 67% fewer tokens. The Sketch-of-Thought paper (EMNLP 2025) reported reductions of up to 84% with minimal accuracy loss. The Broken Chains paper (2025) found state-of-the-art reasoners consuming over 15,000 tokens for problems solvable in a few hundred.

Published CoT token measurements on mathematical reasoning tasks range from approximately 142 output tokens per problem (DSC paper, NAACL 2025) to over 500 tokens for a single grade-school math problem (SemCoT, 2024). At Sonnet-class output pricing of $15 per million tokens, this overhead is material.

2.3 Tool Augmentation Is Now Foundational

A convergent body of research from 2022–2025 establishes that external computation tools are not optional enhancements—they are architectural requirements for numerically reliable AI systems.

Small models with tools beat large models without. Toolformer (Schick et al., NeurIPS 2023) showed a 6.7-billion-parameter model with a simple calculator outperforming the 175-billion-parameter GPT-3 on mathematical benchmarks—a 26× parameter efficiency gain. TALM (Parisi et al., 2022) demonstrated a 220-million-parameter model with arithmetic tools exceeding the performance of a 3-billion-parameter model without them.

Modern frontier models depend on tools. The BEYONDBENCH paper (2025) measured accuracy drops of 16.8%, 15.9%, and 44.0% for GPT-5, GPT-5 mini, and GPT-5 nano respectively when tool access was disabled. The paper concludes that modern LLMs “succeed not through superior language-based reasoning but by recognizing when to use tools for complex computation.”

Code execution dramatically outperforms CoT for math. PAL (Gao et al., ICML 2023) achieved 72.0% on GSM8K versus 65.6% for CoT, with the gap widening to 61.2% versus approximately 20% on harder problems. The Code Interpreter study (ICLR 2024) reported a 27.5 percentage-point improvement on the MATH dataset from code execution.

However, code generation introduces its own failure modes. Interface failures—the model formulating problems incorrectly for the tool—are common, and the majority of errors occur in the first reasoning step before any tool can help. Tool design matters as much as tool capability.

2.4 Floating-Point Nondeterminism Is a Quantified Risk

Even when LLMs successfully generate correct Python code, the results may be unreliable due to floating-point nondeterminism.

Yuan et al. (2025) measured that models exhibit up to **9% accuracy variation from floating-point nondeterminism** under standard inference settings. The root cause is fundamental: floating-point arithmetic is not associative. The expression (a+b)+c does not necessarily equal a+(b+c) in finite precision, and the result depends on GPU count, batch size, and hardware generation.

The LLM4FP paper (SC ’25 Workshops) found that LLM-generated floating-point programs exhibit a 26.56% inconsistency rate across compilers—with differences spanning the full 16-digit precision of double-precision values. LLMs almost never spontaneously use Python’s Fraction or Decimal libraries; they default to float, which silently accumulates error through multi-step computation chains.

3. Empirical Benchmark

3.1 Methodology

We benchmarked 18 computation tasks spanning simple arithmetic, prime factorization, symbolic radicals, exact trigonometry, multi-step chains, workspace management, compute-graph operations, sensitivity analysis, and gradient computation. Each task was evaluated under three execution strategies:

Pure chain-of-thought. We wrote the minimum viable reasoning output an LLM would need to generate for each task—no preamble, no hedging, no verification steps. Token counts were measured on this lean output. Real-world CoT would be 1.5–3× longer.

LLM + Python code execution. For each task, we wrote the minimal Python code an LLM would generate, plus measured the result payload. Token costs account for code as output tokens (expensive), tool-use wrapper overhead (~20 tokens), and results as input tokens (cheap).

Arithym. Actual MCP responses captured from the live tool. Token costs account for the tool-call overhead (~20 output tokens) and the response payload as input tokens.

Computation timing was measured via direct function calls (100 runs per tool, 20 for stateful operations) on the underlying Python engine, reporting median latency. This isolates computation time from MCP transport overhead.

3.2 Token Economics

Table 2. Token Costs per Task (Three-Way Comparison)

Task	CoT (out)	Py code	Py result	Arithym	A. call	Winner
compute(1234×5678)	64t	24t	2t	15t	20t	Arithym
factorize(360360)	63t	106t	7t	22t	20t	Arithym
exact_sqrt(7200)	48t	105t	1t	15t	20t	Arithym
exact_trig(30°)	25t	57t	12t	15t	20t	Arithym
scratch_math(4 steps)	118t	65t	16t	16t	20t	Arithym
field_sensitivity_all	170t	200t	28t	135t	20t	Arithym
graph_what_if(3 scen.)	206t	94t	17t	35t	20t	Arithym
graph_gradient(profit)	159t	126t	17t	102t	20t	Arithym

Full 18-task results available; representative subset shown. CoT = all output tokens. Python = code (output) + result (input). Arithym = response (input) + call overhead (output).

Across all 18 tasks, Arithym won the dollar-cost comparison on 17 of 18, with the sole exception being a trivial workspace-reset operation where CoT’s 14 output tokens undercut both tool-based approaches.

3.3 Dollar-Cost Comparison

Token costs translate to dollars differently for each strategy because output tokens (LLM generation) are priced 5× higher than input tokens (tool responses returned to context) on Sonnet-class models.

Table 3. Dollar Cost at Scale (Claude Sonnet: $3/$15 per MTok in/out)

Volume	Pure CoT	LLM + Python	Arithym	Savings vs Py
1,000 calls	$1.31	$1.09	$0.43	61%
10,000 calls	$13.07	$10.94	$4.29	61%
100,000 calls	$130.67	$109.43	$42.88	61%
1,000,000 calls	$1,306.67	$1,094.33	$428.83	61%

The 61% savings against Python code execution is driven by a structural advantage: Arithym requires approximately 20 output tokens per call (the tool invocation) regardless of task complexity, whereas Python code generation scales linearly with problem difficulty. A sensitivity analysis requires roughly 200 output tokens of Python code; the Arithym call is still 20 tokens.

3.4 Computation Speed

Direct function-call timing reveals that all 18 Arithym operations complete in under 3 milliseconds, with 13 of 18 completing in under 500 microseconds.

Table 4. Latency Comparison (Wall-Clock, Including Model Inference)

Metric	Arithym	LLM + Python	Pure CoT	Speedup vs Py
Average per call	5ms	1.1s	1.2s	197×
Total (18 tasks)	97ms	19.1s	21.1s	197×
Fastest single call	5µs (trig)	552ms	440ms	—
Slowest single call	2.2ms (what-if)	2.4s	2.4s	—

Arithym timing includes ~5ms MCP routing overhead per call. LLM+Python includes TTFT (~300ms), code generation (~10ms/tok), warm sandbox (~200ms), execution, and return. CoT includes TTFT + output token generation at ~100 tok/sec.

For agentic workflows that make dozens to hundreds of computation calls per session, the difference between 5 milliseconds and 1.1 seconds per call is transformative—it is the difference between a responsive agent and one that imposes multi-minute waits on numerical reasoning.

3.5 Exactness by Default

Four of 18 benchmark tasks would produce floating-point (inexact) results under Python code execution without explicit developer intervention to use Fraction or Decimal types. These include division operations that silently lose precision, trigonometric computations that return decimal approximations instead of exact symbolic values, and multi-step chains where floating-point error compounds.

Arithym's computation engine is architecturally incapable of producing floating-point results. All arithmetic is exact and deterministic by design—not as an optional mode, but as the only mode. The internal methods are proprietary, but the guarantee is absolute: given the same inputs, every call returns the same exact result. This eliminates an entire class of silent precision errors that are difficult to detect in production.

4. Competitive Landscape

The market for deterministic computation in AI systems contains one heavyweight incumbent, two platform-locked code execution tools, and a fragmented field of open-source hobby projects. No venture-backed startup currently targets exact arithmetic as a service for AI agents.

4.1 Wolfram Research

Wolfram launched an official MCP server in early 2026, positioning Wolfram Language as a "foundation tool for LLM systems." Wolfram's capabilities are unmatched in breadth: symbolic algebra, differential equations, curated scientific datasets, visualization, and step-by-step pedagogical solutions across hundreds of knowledge domains. For deep mathematical analysis, Wolfram has no peer.

However, for the computation needs typical of AI agent workflows, Wolfram carries significant overhead. Every query requires a **network round-trip to Wolfram Cloud at 1–3 seconds latency.** Responses default to approximately 6,800 characters—extremely token-inefficient for the simple arithmetic that constitutes most agent computation. The free API tier caps at 2,000 calls per month. Enterprise pricing is opaque.

The positioning distinction is clear: Wolfram is an encyclopedia of computational knowledge. Arithym is a fast, exact calculator purpose-built for the agent loop. Developers who need to solve differential equations should use Wolfram. Developers whose agents need to compute financial returns, unit conversions, and multi-step arithmetic thousands of times per session face a different optimization problem.

4.2 Code Execution Tools

OpenAI's Code Interpreter and Google's Gemini Code Execution both provide sandboxed Python environments. These handle arbitrary code—including plotting, file processing, and data analysis—making them genuinely useful for exploratory analysis. For deterministic computation specifically, they carry three disadvantages: latency (container startup at 5–15 seconds cold, plus code generation time), cost ($0.03+ per session for OpenAI, plus output tokens for code), and model lock-in (neither is available cross-platform via MCP).

The deeper issue is that code execution tools introduce a probabilistic layer where determinism is needed. The LLM must generate correct code—choose the right operators, manage types, handle edge cases—and then the sandbox must execute it. Runtime errors are caught, but logic errors are not.

4.3 Open-Source MCP Math Servers

Approximately 10–15 MCP math projects exist on GitHub. The most notable include sympy-mcp (symbolic algebra via SymPy, 47 stars), calc-mcp (21 tools including hashing and dates, 4 stars), and vibe-math-mcp (financial math and linear algebra using Polars). All are local-only, maintained by individual developers, lack commercial support, and provide no persistent workspace or dependency-tracking capabilities.

None offer guaranteed exact arithmetic, compute graphs with cascading recomputation, sensitivity analysis, gradient computation, or the 22 domain-specific modules (financial, chemistry, statistics, calculus, units, etc.) that characterize a production computation layer. The open-source landscape validates market demand—developers recognize the problem—but the solutions remain fragmented and shallow.

4.4 Summary Comparison

Table 5. Competitive Comparison

Dimension	Arithym	Wolfram	Code Interp.	OSS MCP
Latency	<5ms	1–3s	5–15s cold	<5ms
Exact arithmetic	Default	Yes	Float default	Varies
Persistent state	Yes (field)	No	Session only	No
Compute graphs	Yes	Yes	Via code	No
Model-agnostic	MCP	Limited	OpenAI only	MCP
Domain modules	22	Hundreds	Arbitrary code	1–3
Token efficiency	43t avg	~1,700t	71t code	Varies
Production-grade	Yes	Yes	Yes	No

5. Defensible Claims

The following claims are directly supported by peer-reviewed publications and empirical measurement. Each is accompanied by its source for verification.

Supported by Peer-Reviewed Research

Claim	Source
LLMs achieve under 6% accuracy on 4-digit multiplication	Qiu et al., 2024; Yang et al., 2023
Chain-of-thought provides only marginal improvement on arithmetic	Liu & Low, 2023 (Goat)
~67% of CoT tokens are redundant for math reasoning	Han et al., ACL 2025 (TALE)
Tool-augmented models lose up to 44% accuracy without tools	BEYONDBENCH, 2025
Arithmetic errors compound exponentially with chain length	Dziri et al., NeurIPS 2023
A 6.7B model + calculator outperforms 175B GPT-3	Schick et al., NeurIPS 2023
LLM-generated float code is inconsistent 27% of the time	LLM4FP, SC ’25 Workshops

Supported by Empirical Benchmark

Claim	Basis
61% cheaper than Python code execution at scale	18-task benchmark, Sonnet pricing
67% cheaper than chain-of-thought at scale	18-task benchmark, Sonnet pricing
Sub-millisecond computation for 72% of operations	Direct timing, 100 runs/tool
All operations complete in under 3 milliseconds	Direct timing, 100 runs/tool
~200× faster end-to-end than Python code generation	Wall-clock including model inference
43 tokens average per response	Actual MCP response measurement
17/18 tasks cheaper than both CoT and Python	Three-way dollar-cost comparison

Claims Requiring Qualification

Speed comparisons include model inference time. The “200× faster” figure compares Arithym’s sub-millisecond computation against the full wall-clock time of LLM code generation plus sandbox execution. This is the relevant comparison for agent developers (wall-clock latency is what users experience) but should be labeled as an end-to-end comparison, not a computation-vs-computation comparison.

Schema overhead is minimal in practice. Arithym’s deferred tool loading introduces a one-time schema cost of approximately 3,900 tokens on the first turn of each conversation—roughly 2% of a 200K-token context window, or about $0.012 at current input pricing. This is comparable to the overhead of any MCP tool integration and is dwarfed by typical system prompt sizes. With Anthropic’s prompt caching active on subsequent turns, the effective schema cost drops to approximately 320 tokens per turn. Against lean CoT at 44 tokens of savings per call, this means Arithym breaks even within 8 calls after the first turn—the equivalent of a single compute-graph workflow.

Dollar-cost comparisons are pricing-dependent. Savings percentages are calculated at current Claude Sonnet pricing ($3/$15 per MTok input/output). The 5× ratio between output and input pricing is the key driver. If a model's pricing changes, the absolute savings shift proportionally.

6. Market Positioning

The research and competitive analysis converge on a clear market position: Arithym occupies the gap between Wolfram's heavyweight computational knowledge engine and the fragmented landscape of hobby-grade open-source math tools. No current product is purpose-built for fast, exact, token-efficient computation inside AI agent loops.

6.1 The Core Narrative

> *AI agents guess at math. Published research proves it: under 6% accuracy on multiplication, errors that compound exponentially, chain-of-thought that wastes two-thirds of its tokens without fixing the problem. Code generation restores accuracy but at the cost of seconds per call, expensive output tokens, and floating-point precision risks. Arithym eliminates the guesswork: exact arithmetic in sub-millisecond time, delivered as lightweight MCP tool calls that cost 61% less than code generation. Twenty-two domain modules—from financial amortization to unit conversion to gradient computation—accessible by name, not by writing code. Persistent workspaces with cascading dependencies, so your agent builds computational models that update automatically. The computation layer your AI agent is missing.*

6.2 Differentiation Axes

Against Wolfram: Arithym is not a knowledge engine. It does not answer "What is the mass of Jupiter?" It answers "What is 8500 × 4.75 × 1.0725, exactly, in 74 microseconds?" The tools serve different needs.

Against Code Interpreter: Arithym replaces the pattern of "generate Python code → spin up sandbox → execute → parse output" with "call named operation → receive exact result." No code generation overhead, no sandbox latency, no floating-point defaults, no model lock-in.

Against open-source MCP math: Arithym provides the depth (compute graphs, sensitivity analysis, 22 domain modules), reliability (guaranteed exact results, not floating-point eval), and production-readiness that hobby projects cannot match. It is the professional-grade option.

7. Conclusion

The evidence from 18 peer-reviewed publications and our empirical benchmark supports three conclusions:

First, LLMs cannot perform reliable arithmetic, and neither chain-of-thought reasoning nor parameter scaling resolves this. The failure is structural, not a matter of training data or prompt engineering. External deterministic tools are a requirement, not an enhancement.

Second, the current dominant approach—LLM-generated Python code executed in sandboxed environments—solves accuracy at the cost of latency, token overhead, and floating-point precision risk. For the high-frequency, exact-arithmetic computation needs of AI agents, this approach is overbuilt and underperforming.

Third, a purpose-built computation layer delivered via MCP—with exact deterministic arithmetic, named operations instead of code generation, persistent state with cascading dependencies, and token-efficient responses—provides 61% cost reduction, 200× latency improvement, and guaranteed correctness relative to code generation, while serving as a model-agnostic tool available to any MCP-compatible client.

The market window is open. The research consensus is clear. The tooling ecosystem is ready. What remains is execution.

References

Dziri, N., Lu, X., Sclar, M., et al. (2023). Faith and Fate: Limits of Transformers on Compositionality. NeurIPS 2023 (Spotlight).

Gao, L., Madaan, A., Zhou, S., et al. (2023). PAL: Program-aided Language Models. ICML 2023.

Han, S., et al. (2025). TALE: Token-Budget-Aware LLM Reasoning. ACL 2025 Findings.

Liu, T. & Low, B. (2023). Goat: Fine-tuned LLaMA Outperforms GPT-4 on Arithmetic Tasks. arXiv:2305.14201.

Parisi, A., Zhao, Y., Fiedel, N. (2022). TALM: Tool Augmented Language Models. Google Research.

Qiu, L., et al. (2024). Dissecting Multiplication in Transformers: Insights into LLMs. arXiv:2407.15360.

Schick, T., Dwivedi-Yu, J., et al. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. NeurIPS 2023.

Yang, J., et al. (2023). MathGLM: GPT Can Solve Mathematical Problems Without a Calculator. arXiv:2309.03241.

Yuan, Z., et al. (2025). Floating-point nondeterminism in LLM inference. arXiv preprint.

BEYONDBENCH (2025). Evaluating tool-dependency in frontier language models.

Broken Chains (2025). Token efficiency in state-of-the-art reasoning systems.

DSC (2025). Efficient reasoning via dynamic step compression. NAACL 2025 Findings.

GSM-Ranges (2025). Mathematical reasoning across numerical complexity ranges. arXiv:2502.08680.

LLM4FP (2025). Floating-point program generation consistency. SC ’25 Workshops.

Sketch-of-Thought (2025). Efficient LLM reasoning with adaptive cognitive strategies. EMNLP 2025.

SemCoT (2024). Accelerating chain-of-thought reasoning. arXiv:2510.24940.