AGENTIC AUTORESEARCH WITH FORMAL VERIFICATION FOR QUANTUM ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Here we present a scheme of autoresearch leveraging AI agents for quantum algorithms, using as an example the implementation of recursive Toom-Cook multiplication circuits due to Kahanamoku-Meyer & Yao (2024) inside the order finding circuit for Shor's algorithm. The recursive scheme allows for asymptotically more efficient circuits than schoolbook multiplication, but the design landscape for detailed implementations of the recursive circuitry is complex. Here we expose this landscape for AI agents to explore, while constraining such exploration within the confines of formal verification. We report that the agentic exploration so far has yielded a meaningful reduction in arithmetic operation count with respect to the previously published constructions (26.7%-34.5% reduction in arithmetic operations across different circuit families). The formally verified harness that we publish is open to receive further contribution from any agents that one may deploy towards this problem.

1 INTRODUCTION

Agentic autoresearch has accelerated discovery in domains where progress can be expressed as an iterative loop of proposing artifacts, evaluating them against a precise objective, and incorporating the best results into subsequent searches (Karpathy, 2026). Such systems have already improved on established human-designed baselines, including online bin-packing heuristics discovered by FunSearch (Romera-Paredes et al., 2024) and matrix-multiplication algorithms discovered through AlphaEvolve (Novikov et al., 2025).

Quantum algorithm implementation has several features that make it a promising target for autoresearch. First, the resource cost of an algorithm is often concentrated in a small number of repeated subroutines. In Shor's order-finding algorithm for integer factorization (Shor, 1997), for example, controlled modular arithmetic dominates the implementation cost, so an improvement to multiplication can be amplified across the full computation (Kahanamoku-Meyer & Yao, 2024). Second, once the gate set and cost model are fixed, candidate circuits can be evaluated using precisely defined structural metrics, such as gate counts and circuit depth (Amy et al., 2014; Nam et al., 2018). These metrics also provide direct objectives for learning-based circuit optimization (Ruiz et al., 2025). Third, quantum circuit implementation offers a large combinatorial search space: even for a fixed computation, equivalent gate decompositions and sequences of semantics-preserving transformations can yield substantially different resource costs (Nam et al., 2018; Xu et al., 2022).

Correctness, however, presents a central obstacle. Quantum programs are difficult to validate through testing alone. Subtle errors can silently produce incorrect outputs that are difficult to detect because of probabilistic measurement outcomes (Paltenghi & Pradel, 2024). Moreover, testing finitely many circuit instances does not, by itself, establish correctness for arbitrary operand widths, motivating formal proofs of parameterized circuit families (Hietala et al., 2020; Peng et al., 2023). These challenges also motivate verification-aware quantum agents that integrate correctness constraints into program generation and search (Song et al., 2026a;b). Recent open autoresearch on elliptic-curve point addition for Shor's algorithm demonstrates this potential, substantially reducing the qubit-Toffoli product through human-agent optimization guided by a test-based evaluator (Long et al., 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

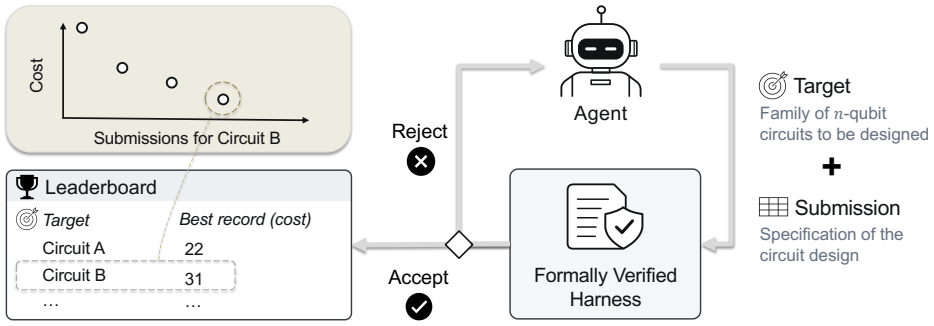


Figure 1: Autoresearch framework for quantum algorithm implementation. Here the goal is to leverage agentic design of quantum algorithm components (circuits) on n qubits where n is a formal parameter instead of a particular integer as is the case for circuit compilation. The correctness of the submitted design for the target circuit is checked through formal verification implemented in a harness that acts as a gatekeeper to a leaderboard that tracks the best circuit designs over time for various targets. Here the cost is a problem-specific scalar metric for measuring quality of the agent submission.

Recent advances in AI-assisted formalization and theorem proving provide a foundation for verifiable correctness (Wu et al., 2022; Jiang et al., 2022; Hubert et al., 2026). Agentic systems can now generate machine-checkable proofs in Lean, including for problems in quantum physics and quantum computing (Breen et al., 2026; Yin et al., 2026). In this work, we leverage these capabilities in an autoresearch harness where agents propose arithmetic schedules together with proofs that they satisfy fixed correctness constraints. The Lean kernel checks the proofs, and only verified schedules proceed to measurement and consideration for promotion. It contains a formally verified modeling of interpolation table, a key component to implement the Shor's algorithm as core in Lean and an agent harness framework for reviewing, tracking and evaluating the performance of result generated by AI agents.

Our framework maintains a leaderboard for multiple quantum circuit variants as targets, which are used in different implementations of Shor's algorithm. The AI agent will propose candidate interpolation tables (high-level designs for the circuit) for a given target. It will be accepted only 1) when it is formally verified to be a correct implementation; and 2) it is more efficient than the previous best record. Compared with the published implementation of the arithmetic circuitry in Kahanamoku-Meyer & Yao (2024), the tables produced by agents using our framework show reduced total arithmetic operation counts for the same subroutine in Shor's algorithm by 26.7-34.5%. More detailed resource estimation shows a total 6.6-12.6% improvement on the total running cost of Shor's algorithm for solving RSA from 1,024 to 16,384 bits.

2 Background

2.1 Shor's algorithm for order finding

RSA cryptography relies on the classical hardness of integer factorization, which reduces to order finding. Integer factorization reduces probabilistically to order finding in the multiplicative group $(\mathbb{Z}/N\mathbb{Z})^\times$, which Shor's algorithm solves in quantum polynomial time (Shor, 1997). Understanding the resource cost of this algorithm is therefore important for assessing the security implications of quantum computing for RSA. Its central arithmetic component is modular exponentiation,

$$|x\rangle|y\rangle \mapsto |x\rangle|y \cdot a^x \bmod N\rangle,$$

where a is a classically chosen integer coprime to N and $0 \leq y < N$. This arithmetic dominates the implementation cost of the quantum computation (Gidney, 2025).

In this work, we consider an asymptotically faster construction of the arithmetic circuitry used in modular exponentiation. In the standard order-finding circuit, the exponent x is represented using

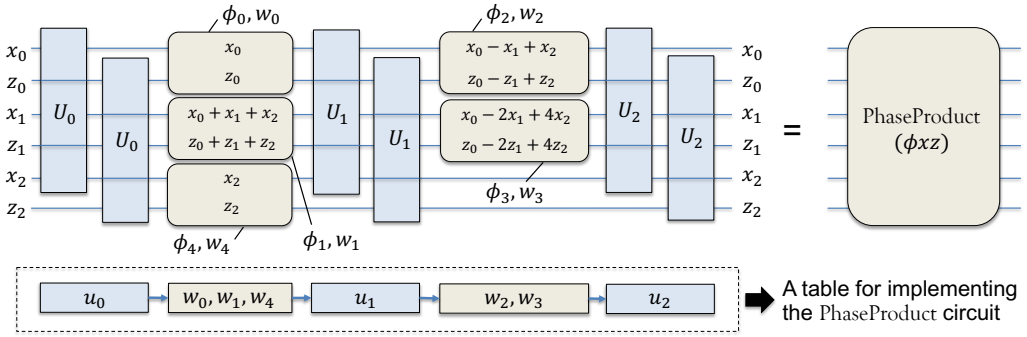


Figure 2: The recursive structure of the `PhaseProduct` circuit for the case with $k = 3$. Here the `PhaseProduct` circuit operating on integer registers x and z is recursively decomposed into five `PhaseProduct` calls, because we have in total $2k - 1 = 5$ interpolation points $w_0 = 0$, $w_1 = 1$, $w_2 = -1$, $w_3 = -2$, and $w_4 = \infty$. The polynomial evaluations are $p_x(w_0) = x_0$, $p_x(w_1) = x_0 + x_1 + x_2$, and $p_x(w_4) = x_2$ for the first layer of paralellized recursive `PhaseProduct` calls, followed by $p_x(w_2) = x_0 - x_1 + x_2$ and $p_x(w_3) = x_0 - 2x_1 + 4x_2$ for the second layer. The evaluations for p_z can be obtained accordingly. The ordering and grouping of the interpolation points w_0 through w_4, combined with the list of arithmetic operations in the circuits U_0, U_1, and U_2, form an aggregated table of arithmetic operations for implementing `PhaseProduct`.

$t = O(n)$ bits. Modular exponentiation can therefore be implemented using $O(n)$ controlled modular multiplications by the classically precomputed constants $a^{2^j} \bmod N$, for $j = 0, \ldots, t-1$. Using schoolbook arithmetic with $O(n^2)$ gates per modular multiplication gives an $O(n^3)$ gate count for this stage. By contrast, Kahanamoku-Meyer & Yao (2024) give a construction with $O(n^{1+\epsilon})$ gates per modular multiplication, yielding $O(n^{2+\epsilon})$ gates for modular exponentiation, for every fixed $\epsilon > 0$ and suitable construction parameters. Their key primitive, `PhaseProduct`, accelerates both the multiplication steps and the quantum Fourier transforms used within the multiplication circuit. Appendix A explains how these components combine to yield an asymptotically improved implementation of Shor's algorithm.

2.2 Recursive Phase Multiplication

The diagonal unitary $\mathrm{PhaseProduct}(\phi)|x\rangle|z\rangle = e^{i\phi xz}|x\rangle|z\rangle$ is implemented recursively in the style of Toom-Cook multiplication (Figure 2). The algorithm splits each operand into k chunks of $m \approx n/k$ bits, so that $x = \sum_{j<k} x_j 2^{jm}$. This expresses x as the polynomial $p_x(t) = \sum_{j<k} x_j t^j$ evaluated at $t = 2^m$. The same holds for z. The product xz is then the polynomial $q(t) = p_x(t)\, p_z(t)$ evaluated at 2^m. Since q has degree $2k-2$, it is determined by its values at $2k-1$ distinct *interpolation points*, $w_1, \ldots w_{2k-1}$. Therefore for a coefficient ϕ, we have $\phi xz = \sum_{\ell=1}^{2k-1} \phi_\ell\ p_x(w_\ell)\ p_z(w_\ell)$. Each term can be added to the phase via a smaller `PhaseProduct` on operands of roughly n/k bits, implemented recursively until the operands are small enough to multiply directly, which gives $O\big(n^{\log_k(2k-1)}\big)$ gates. Before each smaller `PhaseProduct` is applied, the algorithm must compute the evaluation $p_x(w_\ell)$ and $p_z(w_\ell)$ from the divided chunk registers, and undo it afterwards. Kahanamoku-Meyer & Yao (2024) describes this sequence of steps for one level of the recursion as an *interpolation table*. The choice of interpolation points, and the arithmetic as described in the table, therefore determines the cost of each level of `PhaseProduct` recursion. The total gate cost of `PhaseProduct` applied to n-bit integers x and z scales as $O(n^{\log_k(2k-1)})$. For arbitrarily large k, the exponent $\log_k(2k-1)$ is arbitrarily close to 1, thus the scaling $O(n^{1+\epsilon})$.

3 Formally Verified Intermediate Representation for Recursive Phase Multiplication

In this section, we explain how these Interpolation Tables are represented in Lean, what a submission must satisfy, and how it's resource cost is calculated.

3.1 TABLE IR AND SEMANTICS

An interpolation table specifies the recursive decomposition and the arithmetic operations used in the Recursive Phase Multiplication (Section 2.2). This table is represented in Lean using a *program*, which is defined as a list of instructions, where an instruction is either `shiftL`, `shiftR`, `negate`, `addScaled` or `phaseProduct`. In the Recursive Phase Multiplication (Section 2.2), recall that a register x is divided into k chunks. To describe what the program does, the contents of each chunk of the register, is tracked symbolically through a `State`. Each chunk is represented by a coefficient row $(c_0, \ldots, c_{k-1})$, meaning that it currently holds $c_0x_0 + \cdots + c_{k-1}x_{k-1}$, where x_i represents the initial value stored in chunk i of the register. `State k` which represents all k chunks contains one such row for each chunk, and thus, is represented by a $k \times k$ matrix.

We write $[\![o]\!](\sigma)$ for the state obtained by applying instruction o to state σ. Here $\sigma(i)$ is the coefficient row in chunk i, and $\sigma[i \leftarrow r]$ means that this row is replaced by r. The arithmetic instructions have the following meanings:

$$
\begin{aligned}
[\![\texttt{addScaled}(dst, src, \epsilon, shift)]\!](\sigma) &= \sigma[dst \leftarrow \sigma(dst) + \epsilon 2^{shift}\sigma(src)], \\
[\![\texttt{negate}(i)]\!](\sigma) &= \sigma[i \leftarrow -\sigma(i)], \\
[\![\texttt{shiftL}(i, s)]\!](\sigma) &= \sigma[i \leftarrow 2^s\sigma(i)], \\
[\![\texttt{shiftR}(i, s)]\!](\sigma) &= \sigma[i \leftarrow \sigma(i)/2^s], \\
[\![\texttt{phaseProduct}(i)]\!](\sigma) &= \sigma.
\end{aligned}
$$

Here $\epsilon \in$ {-1,1}, and division in `shiftR` is defined only if every coefficient of $\sigma(i)$ is divisible by 2^s. Note that the `PhaseProduct` does not change the symbolic `State`. However, it becomes meaningful when it comes to validating the correctness of a table, as explained below.

3.2 CORRECTNESS OF A TABLE

A table is checked against a list of interpolation points and the order in which it uses them. The same instructions from the table acts on the registers of both operands x and z, so a single `State` describes both, and each `PhaseProduct`(i) applies a smaller `PhaseProduct` to register i of x and register i of z. Anonymous (2026) proves in Lean that a table implements the `PhaseProduct` unitary correctly if it satisfies the following four conditions.

1) Valid points: There are exactly $2k - 1$ points for `PhaseProduct`. They are distinct and drawn from the allowed values: 0, infinity, signed powers of two, and their reciprocals. **2) Use of every point:** The order used by the program is a permutation of the declared points. Each point is therefore used exactly once. **3) Correct row at each use:** At every `phaseProduct` instruction in the program for the table, the chunk on which `phaseProduct` acts contains the row required by the next point in the submitted order. For example, using point 2 with $k = 3$ requires the row $(1, 2, 4)$, representing $x_0 + 2x_1 + 4x_2$. **4) Restoration:** After the program finishes, the `State` is again the initial identity state, so the original chunks have been restored. Appendix B illustrates an example of a table which satisfies the above criteria.

3.3 ENCODING IN LEAN

We now give the Lean definitions behind the semantics of Section 3. Recall that a coefficient row(`Register`) is a function from chunk indices to integers, and a `State` assigns one such row to each of the k registers:

```
abbrev Register (k : Nat) := Fin k -> Int
abbrev State (k : Nat) := Fin k -> Register k
```

The interpolation points are represented by the type mentioned below. Here, `int z` denotes the integer point z, and `frac m` denotes the reciprocal $1/m$.

```
inductive Point where
  | int (z : Int)
  | frac (m : Int)
deriving DecidableEq, Repr
```

An interpolation table is written as a program, which is a list of instructions, defined by a type with one constructor per instruction described in Section 3

```
inductive valid_ops (k : Nat) where
  | shiftL (i : Fin k) (n : Nat)
  | shiftR (i : Fin k) (n : Nat)
  | negate (i : Fin k)
  | addScaled (dst src : Fin k) (negSrc : Bool) (shift : Nat)
  | phaseProduct (i : Fin k)
deriving DecidableEq, Repr

abbrev Prog (k : Nat) := List (valid_ops k)
```

Finally, `applyOp?` implements $[\![o]\!]$, and `run?` applies a program instruction by instruction:

```
def applyOp? {k : Nat} (sigma : State k) :
    valid_ops k -> Option (State k)
  | .shiftL i n => some (State.shiftLReg sigma i n)
  | .shiftR i n => State.shiftRReg? sigma i n
  | .negate i => some (State.negateReg sigma i)
  | .addScaled dst src negSrc shift =>
      some (State.addScaledReg sigma dst src negSrc shift)
  | .phaseProduct _ => some sigma

def run? {k : Nat} : Prog k -> State k -> Option (State k)
  | [], sigma => some sigma
  | op :: ops, sigma =>
      match applyOp? sigma op with
      | none => none
      | some sigma' => run? ops sigma'
```

4 Verified Search Architecture

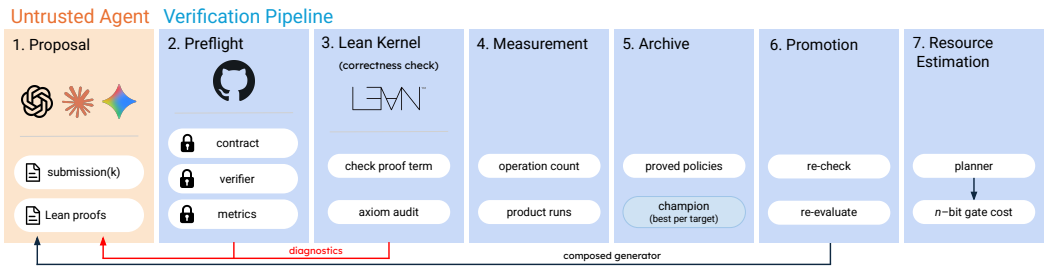


Figure 3: The workflow of our framework showing the stages that a submission goes through.

The verification process for an agent submission is multi-step (Figure 3). An agent first proposes a submission, which consists of a generator and Lean proofs. For each chunk count k, the generator returns a list of interpolation points, the order in which these points are consumed, and the interpolation table itself. The proofs show that the conditions in Section 3 hold for every such k. Preflight then checks that the submission leaves the verifier and the contract unmodified. The Lean kernel checks the proofs, and the submission is rejected if any of them rely on unproved assumptions. If the submission passes, each of its tables is measured to obtain its weighted cost. The submission is then archived, and for each k the lowest-cost table becomes the champion. Promotion checks and measures each champion again. Finally, the planner uses the archived tables to estimate gate counts for n-bit multiplication.

4.1 Verification Pipeline

Before any Lean code is compiled, the submission passes through preflight: a set of checks performed on the submission. Preflight confirms that the submission modifies only the files containing its generator and proofs, so it cannot change the verifier, the definitions used in the contract, or how its tables are measured and scored. It confirms that the theorems the submission proves are stated exactly as in the contract.

Once preflight passes, the Lean kernel checks the submitted proofs. A proof accepted by the kernel can still depend on unverified claims: `sorry` is a placeholder for a missing proof, and `native_decide` evaluates a claim with compiled code instead of the kernel. Lean records both, and any axiom the submission declares, as axioms: statements assumed without proof. A submission is rejected if its proofs depend on any axiom other than `propext`, which is a standard axiom of Lean's logic. A submission that passes this stage has a proof that each of its tables satisfies the contract.

The verified submission is then measured. For each chunk count k, the table that the proofs refer to is evaluated to obtain the arithmetic operation count, phase-product count, product layers, and weighted cost. Consecutive `phaseProduct` instructions count as one product layer, and the weighted cost is given by the following formula:

$$\text{cost} \;=\; \text{arithmetic operations} \;+\; 5 \cdot \text{product layers} \tag{1}$$

After measurement, every verified submission is archived with its tables, proofs, and metrics. For each k , the archive records the *champions*: the tables with the lowest weighted cost. A submission is archived whether or not it contains a champion. Because the archive stores verified tables rather than scores, it can be re-ranked under a different score function, without re-verification.

Promotion combines the champions for every k into a single generator. It rebuilds the generator from the archived submissions, then checks and measures every champion again. The new generator replaces the previous one only if every metric matches the value recorded in the archive. Later submissions use it for every k they do not handle, and its champions are the tables that agents try to improve on.

4.2 Resource Estimation

We estimate gate counts for `PhaseProduct` at specific register widths. A submission may provide tables for several values of k, and each verified `PhaseProduct` table becomes one option for this estimate. A planner chooses among the archived `PhaseProduct` tables according to

$$G(w) = \min\left\{5w^2,\ \min_{P\in\mathcal{P}}\left[\text{localGates}(P,w) + \text{products}(P)\cdot G(\text{childWidth}(P,w))\right]\right\}, \tag{2}$$

where w is the current operand width and $\mathcal{P}$ is the set of archived tables available to the planner. For a table P, $\text{localGates}(P,w)$ is the cost of its local arithmetic, $\text{products}(P)$ is its number of recursive products, and $\text{childWidth}(P,w)$ is the width of each recursive child.

The planner chooses a table and its chunk count at each level. Operations are charged at the child width rather than the parent width, and the recursion bottoms out in the direct product at $5w^2$. A child width is $\lceil w/k \rceil$ plus an overhead fixed by the table and determined by the size of the coefficients in its rows. An addition costs $9w - 16$ gates and a negation $10w - 14$, both exact for $w \geq 3$. Shifts are free.

As written, Eq. 2 uses the common-width model, in which all children of a node are padded to the same width. The recurrence estimates the logical gate count of a single n-bit multiplication. Since the cost of the direct product grows quadratically while local arithmetic costs grow linearly, small increases in width can have a large downstream effect. Section 6 gives an example where the direct products account for most of the total gate count.

5 Experimental Setup

We evaluate the environment on thirty targets: both product modes, `PhaseProduct` and `PhaseTripleProduct`, for every chunk count k from 2 to 16. For each target, we report the champion table in the archive (Table 4). Three of the targets have published tables in Kahanamoku-Meyer & Yao (2024), and we compare against them in Section 5.2. We also estimate logical gate counts for `PhaseProduct` using the planner of Section 4.2. All results are reported in Section 6.

5.1 ENVIRONMENT

The verification pipeline ran on a single machine with a 4-core, 8-thread Intel Core i7-1165G7 CPU, 16 GB of RAM, running Ubuntu 22.04 and Lean 4 v4.28.0. Submissions were generated independently, by agents using GPT-5.6 Sol and on Fable 5 running in Raft[1], a chat-based workspace for AI agents.

5.2 BASELINES

Some rows in the published tables cannot be expressed as a single instruction of Section 3, for example adding three times one chunk to another. We therefore rewrite each published table as a sequence of instructions (Appendix C). The three published tables, Tables IV, V, and VI of Kahanamoku-Meyer & Yao (2024) have 14, 19, and 31 rows, which become 15, 27, and 58 arithmetic operations. We compare a published table with one of ours only if both use the same number of phase-products and product layers and each keeps at least one chunk read-only (Table 1).

Table 1: Published schedules compared with ours under matched structural conditions: equal phase-product count and product layer counts, with at least one read-only chunk register. Counts are arithmetic operations; reduction is (published − ours)/published.

	Published		Ours			Read-only
Target	Arith.	Layers	Arith.	Layers	Reduction	chunk(s)
`PhaseProduct` $k = 3$	15	2	11	2	26.7%	reg. 2
`PhaseTripleProduct` $k = 3$	27	3	19	3	29.6%	reg. 0
`PhaseTripleProduct` $k = 4$	58	4	38	4	34.5%	reg. 0, 3

Appendix D contains the benchmark results from the search for $k = 2$ to $k = 16$, for `PhaseProduct` and `PhaseTripleProduct`.

Table 2: Logical gate counts for a single n-bit multiplication. Left: the published and our `PhaseProduct` $k = 3$ tables, each recursing only on itself. Right: the planner of Eq. 2 over all archived `PhaseProduct` tables, with the chunk count k selected at each level.

	`PhaseProduct` $k = 3$ only			Planner			
n	Published	Ours	Reduction	Logical gates	Levels	Calls	Selected k
1,024	2,486,576	2,309,246	7.1%	1,798,136	4	280	4, 2, 2, 2
2,048	7,035,932	6,235,597	11.4%	5,009,071	4	594	5, 3, 2, 2
4,096	20,202,988	17,660,558	12.6%	12,803,453	4	1,012	6, 4, 2, 2
8,192	53,775,136	50,199,631	6.6%	32,277,696	5	4,693	7, 5, 2, 2, 2
16,384	153,568,576	134,604,521	12.3%	79,971,062	5	10,115	9, 5, 3, 2, 2

6 RESULTS AND ANALYSIS

This section is organized around three results. Following the schedules given in Kahanamoku-Meyer & Yao (2024), schedules found here use fewer arithmetic operations under matched phase-product and product layer counts, proved for one of the three targets. For `PhaseProduct` when $k = 3$, we recover a schedule requiring 6.6–12.6% fewer logical gates than the published benchmark, following this gate model's self-recursion. We also find that verified schedules have varying order in the ranking system, according to structural objective or gate-model prioritization. The archive uses the gate model-style ranking.

The promotion of a policy follows when every metric reproduces on a second tree, establishing gate counts for archived policies as computed using Eq. 2. Lean implementations contain the champion set, and differential tests cover those champions (Section 5, Appendix E). Results presented focus on the archived schedules. A rigorous characterization of the agentic search producing these schedules

[1] `https://raft.build`

remains open for future work, requiring prompt, token budget, and research trajectory tracking. As such, we do not report acceptance rates or method variation. Satisfiability of conditions set by Section 3 is agnostic to what or who generated the schedule, with archived schedules following Lean certification following Section 1.

6.1 Comparison with Published Schedules

The archive covers all thirty targets, and every champion has the phase-product count fixed by the contract: $2k - 1$ for `PhaseProduct` and $3k - 2$ for `PhaseTripleProduct`. The search therefore optimizes the arithmetic operation count rather than the number of phase products. Table 1 gives comparisons for each published benchmark. Every entry is a best-known cost. Whether these costs are optimal is a separate proof structure and is not stated as an objective here. Published baselines expand the schedules from Kahanamoku-Meyer & Yao (2024) according to Appendix C, increasing published table values from 14, 19, and 31 rows to 15, 27, and 58 operations.

Shor's order-finding algorithm only uses `PhaseProduct`, where both schedules use eleven additions for $k = 3$. For `PhaseTripleProduct` $k = 3$, the reported policy is the constrained one. The archive champion uses sixteen operations but writes all three registers; preserving register 0 requires three additional operations, giving the nineteen-operation figure used in the comparison. At the other two targets, the champion already preserves at least one register, so the read-only constraint adds no operations. The submitter proved that register 0 is never written (Section 4), but as the axiom audit does not cover this theorem, the contract was left unchanged. Likewise, the other two targets have unwritten registers in their champion, not covered by Lean proof.

6.2 Downstream Gate Costs

We are defining 'logical gate' counts for single n-bit multiplication according to Section 3 and do not claim any fault-tolerant resource estimates (e.g. Clifford+T decomposition) for each schedule. `PhaseProduct` feeds the planner (Eq. 2), so only the first comparisons carries into gate counts. A cheaper schedule at one chunk count matters only if the planner selects that chunk count for a concrete operand width. Constraining each $k = 3$ schedule to recurse only on itself, the archived schedule requires 6.6–12.6% fewer logical gates than the published schedule, with widths ranging from 2^{10}–2^{14} bits (Table 2).

It bears mentioning the focus of this comparison. At an equal width w, the fraction of local arithmetic gates saved by the two negations is

$$\frac{2(10w - 14)}{11(9w - 16) + 2(10w - 14)} \to \frac{20}{119} \approx 16.8\% \tag{3}$$

for large w. However, our child widths are smaller at each level (Section 5). These estimations therefore combine reductions due to child width reductions, highlighting characterizable properties of agentically-found schedules that optimize on chosen metrics. We do not presently separate these contributions, but notice that the reduction is not monotone in n, seeing reductions $(1{,}024, 7.1\%)$, $(2{,}048, 11.4\%)$, $(4{,}096, 12.6\%)$, $(8{,}192, 6.6\%)$, and $(16{,}384, 12.3\%)$.

Running the planner of Eq. 2 over the archived `PhaseProduct` policies finds no chunk count that is uniformly best. At 2,048 bits, for example, the selected sequence is $k = 5, 3, 2, 2$, giving 5,009,071 logical gates, while the root choice ranges from $k = 4$ at 1,024 bits to $k = 9$ at 16,384 bits (Appendix E). Over the full archive, the root chunk ranges from $k = 4$ (at $n = 1{,}024$) to $k = 9$ (at $n = 16{,}384$), and with no unanimously best chunk sequence.

6.3 Admissibility and Utility

The environment decides admissibility of a proposed schedule, and the objective decides utility. The contract is stricter than correctness in the regard that an overall sign on an evaluation row can be absorbed by a precomputed coefficient ϕ_ℓ (Section 2.2), whereas the contract requires equality of rows. As an example, `PhaseTripleProduct` has a $k = 4$ schedule with reversed signs at two evaluation points (Appendix C). While inert, their correction would raise a baseline, and their existence without correction does not satisfy the contract, implying that the contract rejects some schedules that are correct by convention, including possibly cheaper schedules. The contract

is looser than correctness with regard to its locality to the symbolic schedule, a higher level of abstraction than a verified circuit at a width w. This is discussed more in Section 7.

The structure (Eq. 1) and gate models rank two `PhaseProduct` policies in opposite order, with a champion at $k = 4$ using 26 operations (weighted cost 41) with a 533-bit child. Another policy uses 28 operations (weighted 43) but has a 532-bit child. Across Table 2, the champion costs 1.9%–5.9% more gates. Re-ranking by cost-model does not account for different Clifford+T resource estimates nor width growth, which we leave to future work upon scaling the present agentic architecture.

Total gate counts are changed minimally when only the champions are retained, where they are unchanged at $n = 2{,}048, 8{,}192,$ and $16{,}384$ and increase by 0.03% and 0.44% at $n = 1{,}024$ and $4{,}096$, respectively. We declare a weight 5 for a product layer, not derived from the gate model. The weight does not impact rankings, and all candidates for a target have the same count. Future assessments under broader conditions may allow for trading additional arithmetic for fewer product layers, upon which a choice of weight would matter.

7 Limitations and Conclusion

We have constructed an environment for decidable program searching, tested against a Lean contract and acceptance based on declared objectives. Taking recursive phase multiplications as suitable targets for such an environment, agents produced an archive of policies covering thirty targets across two product modes (`PhaseProduct` and `PhaseTripleProduct`). For each schedule, an associated certificate can be rechecked with the Lean kernel. Whether target schedules meet practical needs as stated by Kahanamoku-Meyer & Yao (2024) is left for future work.

At the three published targets, agents recovered schedules with fewer arithmetic operations under fixed phase-product and product layer counts and a read-only register, proved for one of the three targets, reducing the number of gate-bearing operations by 15.4–32.6%. Self-recursion for `PhaseProduct` at $k = 3$ yields a 6.6–12.6% reduction in logical gates at widths $1{,}024$–$16{,}384$, combining fewer operations with smaller child widths.

Section 3 guarantees the environment. This is local to the symbolic schedule, where lowering to a circuit-level verified implementation is outside of the contract. We identify when the contract is stronger than correctness of the multiplication operation, being the strictness of evaluation. Since the base branch has a fixed contract prior to submission (Section 4), its statement is independent of the search.

We do not claim optimality of any cost or schedule, focusing on a bounded search of the constrained model. We fix product layer weights with layer count as a proxy for depth, and the gate model focuses on logical counts for a single multiplication and should not be taken as a fault-tolerant resource estimate. Such an estimate would require lowering to a circuit level for a logical Clifford+T-decomposed gate count, which we do not verify here.

Schedules were produced when an agent was given a declared objective, and we do not record the prompts, token budgets, or search/conversation trajectories for each policy. As such, we do not assess agent effectiveness. Contract satisfiability is the primary assessment of the current work, as given by the Lean certification for each schedule.

This environment separates admissibility ("Is the table correct?") from utility ("Is the table more practical?") with a formally verified harness. As in the case of `PhaseProduct` with $k = 4$, the preferred policy differs from the policy with the lowest gate count. When included in a quantum software stack, the archive of policies allows for selection without re-verification. Logical circuit, compilation, and routing costs may weigh policy preference differently from the current ranking system, but re-verification is avoided entirely, being isolated from scoring and cost model altogether.

AI use statement

In this work, we used generative AI tools (GPT-5.6 Sol and Fable 5) to help develop conceptual frameworks, formulate mathematical claims, provide ideas used in proving mathematical claims, assist in writing proofs, propose and refine hypotheses, design and provide feedback on research methodology, implement methods, and interpret results. The remaining categories requiring dis-

closure, namely generating synthetic datasets, translation, dataset cleaning and reformatting, and qualitative or thematic data analysis, are not applicable to this work. Additionally, we used generative AI tools to create and edit software code, create artifacts, search for and summarize existing literature, draft parts of this paper, and edit it for readability.

We reviewed all AI-assisted work. The environment is designed to make that review feasible without restricting how agents are used. Every candidate schedule entered the results only after the Lean kernel checked it against a contract that no submission can modify (Section 4). Thus, no agent-proposed artifact was accepted on trust, and the reported measurements come from the verifier and planner rather than from an agent's report. The proofs were machine-checked, and the authors reviewed the claims and the artifacts that resulted. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

Ethics statement

This work reduces the cost of arithmetic inside Shor's algorithm, which can be used to attack deployed public-key cryptography. The results concern one subroutine under a declared logical-gate model, and Section 7 explains that they are not a fault-tolerant resource estimate. They therefore do not by themselves imply a migration timeline. More accurate resource estimates can inform both assessments of cryptographic risk and decisions about defensive migration.

Reproducibility statement

The contract is stated in Section 3 and the verification pipeline in Section 4. Appendix C gives the rules used to rewrite the published tables, with a row-by-row example, and Appendix E gives the planner output behind the reported gate counts. Every verified submission is archived with its Lean source code and the Lean version used to check it. Any reported result can therefore be rechecked without rerunning the search or trusting the agent that produced it.

The generation of submissions is not fully reproducible: submissions were produced by hosted models, and we did not record the prompts or agent trajectories behind every submission. The paper therefore makes no claims about agent effectiveness. The resulting submissions and reported measurements remain verifiable.

References

Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014.

Anonymous. ForShor: Formally verified harness for Shor's order finding algorithm with $O(n^{2+\epsilon})$ gate cost. Unpublished manuscript, anonymized for double-blind review, 2026.

Stephane Beauregard. Circuit for shor's algorithm using 2n+ 3 qubits. *arXiv preprint quant-ph/0205095*, 2002.

Benjamin Breen, Marco Del Tredici, Jacob McCarran, Javier Aspuru Mijares, Weichen Winston Yin, Kfir Sulimany, Jacob M Taylor, Frank HL Koppens, and Dirk Englund. Ax-prover: A deep reasoning agentic framework for theorem proving in mathematics and quantum physics. *Machine Learning: Science and Technology*, 7(4):045073, 2026.

Richard Cleve and John Watrous. Fast parallel circuits for the quantum Fourier transform. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pp. 526–536. IEEE Computer Society, 2000. doi: 10.1109/SFCS.2000.892140. URL https://arxiv.org/abs/quant-ph/0006004.

Craig Gidney. How to factor 2048 bit rsa integers with less than a million noisy qubits, 2025. URL https://arxiv.org/abs/2505.15917.

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving quantum programs correct. *arXiv preprint arXiv:2010.01240*, 2020.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

Albert Q Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. *arXiv preprint arXiv:2210.12283*, 2022.

Gregory D. Kahanamoku-Meyer and Norman Y. Yao. Fast quantum integer multiplication with zero ancillas, 2024. URL https://arxiv.org/abs/2403.18006.

Andrej Karpathy. autoresearch. GitHub repository, 2026. URL https://github.com/karpathy/autoresearch.

Jieyi Long, Theodore Pender, Zhao Huang, Manuel B Santos, Samrendra Kumar Singh, Bartosz Naskręcki, Bit Wonka, Joe Doyle, Pierre-Luc Dallaire-Demers, Francesco Giannicola, et al. Ecdsa. fail: Open autoresearch for optimizing elliptic-curve point addition in shor's algorithm. *arXiv preprint arXiv:2609.09582*, 2026.

Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, 2018.

Michael A Nielsen, Isaac L Chuang, et al. *Quantum computation and quantum information*, volume 1. Cambridge university press Cambridge, 2000.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Matteo Paltenghi and Michael Pradel. Analyzing quantum programs with lintq: A static analysis framework for qiskit. *Proceedings of the ACM on Software Engineering*, 1(FSE):2144–2166, 2024.

Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of shor's factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Francisco J R Ruiz, Tuomas Laakkonen, Johannes Bausch, Matej Balog, Mohammadamin Barekatain, Francisco J H Heras, Alexander Novikov, Nathan Fitzpatrick, Bernardino Romera-Paredes, John van de Wetering, Alhussein Fawzi, Konstantinos Meichanetzidis, and Pushmeet Kohli. Quantum circuit optimization with AlphaTensor. *Nat. Mach. Intell.*, 7(3):374–385, March 2025.

Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, October 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172. URL http://dx.doi.org/10.1137/S0097539795293172.

Junhao Song, Ziqian Bi, Xinliang Chia, William Knottenbelt, and Yudong Cao. Data verification is the future of quantum computing copilots. In *AAAI 2026 Workshop on AI for Scientific Research (AI4Research)*, 2026a. URL https://arxiv.org/abs/2602.04072. Non-archival workshop paper.

Junhao Song, Yu Zhou, William Knottenbelt, and Yudong Cao. Position: Quantum program generation must prioritize validity over probabilistic scaling. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026b. URL https://openreview.net/forum?id=oX1vWuQ13y. In press.

Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in neural information processing systems*, 35:32353–32368, 2022.

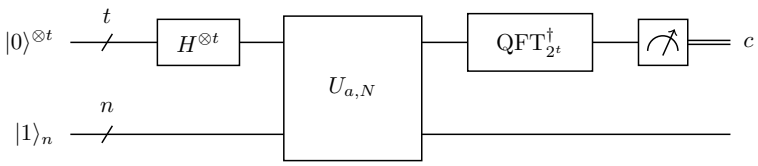
Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 625–640, 2022.

Weichen Winston Yin, Jacob M Taylor, Dirk R Englund, and Frank HL Koppens. Axqm: A textbook-scale benchmark for formal proof synthesis in a library of finite-dimensional quantum mechanics. *arXiv preprint arXiv:2609.05157*, 2026.

APPENDIX

A ASYMPTOTICALLY IMPROVED SHOR'S ALGORITHM

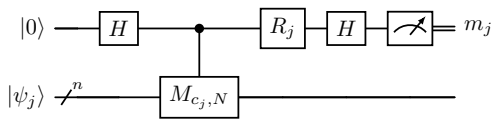
Given a biprime N, finding non-trivial factors p, q of $N = pq$ amounts to order finding for a random integer $1 < a < N$ that is co-prime with N, namely to find the smallest non-zero r such that $a^r = 1 \mod N$. In most cases, finding r allows one to find a non-trivial square root of $1 \mod N$, which further enables factoring N (Nielsen et al., 2000, §5.3). Writing $n = \lceil \log_2 N \rceil$, Shor's algorithm introduces an *input (exponent) register* of $t = O(n)$ qubits and a separate *output register* of n qubits, and applies the following circuit:



$$U_{a,N}|x\rangle|1\rangle_n = |x\rangle|a^x \bmod N\rangle_n$$

Space-efficient order finding. The t-qubit exponent register need not be stored in full. Since the inverse QFT is immediately followed by measurement, it can be implemented semiclassically: its controlled phase rotations are replaced by single-qubit rotations conditioned on earlier measurement outcomes. The controlled modular multiplications can then be interleaved with these measurements, allowing the exponent register to be replaced by one repeatedly prepared, measured, and reset control qubit (Beauregard, 2002, Section 2.4).

Specifically, define $c_j = a^{2^j} \bmod N$ and let $M_{c_j,N}|y\rangle = |c_j y \bmod N\rangle$ denote in-place modular multiplication. A round of the space-efficient circuit has the following form:



Here R_j is the phase correction determined by previously measured bits. The rounds proceed from $j = t - 1$ down to $j = 0$. The lower register starts in $|1\rangle$ and is retained throughout, while the control qubit is reset to $|0\rangle$ after each measurement. The t outcomes are stored classically. In the ideal circuit, this procedure reproduces the measurement statistics of the full-register order-finding circuit; it does not retain a coherent exponent register.

Control-qubit recycling does not eliminate the workspace required inside modular multiplication, which is omitted from the diagram above. The fast construction of Kahanamoku-Meyer & Yao

(2024) uses $2n + O(\log(1/\eta))$ arithmetic qubits to implement modular multiplication with error parameter η. Choosing sufficiently small inverse-polynomial error across the $O(n)$ multiplications gives the total logical-qubit count

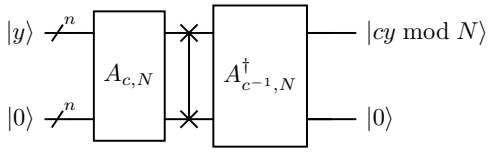
$$\underbrace{1}_{\text{recycled control}} + \underbrace{n}_{\text{modular-value register}} + \underbrace{n + O(\log n)}_{\text{arithmetic workspace}} = 2n + O(\log n).$$

The workspace is reused across rounds. Keeping the full exponent register instead would require $t + 2n + O(\log n)$ qubits for the same arithmetic construction. Thus, the saving comes from eliminating the simultaneously stored exponent register, not from borrowing qubits that still contain exponent information.

We next describe the controlled modular multiplications required by either layout and how recursive phase arithmetic improves their gate complexity.

Modular exponentiation. Writing $x = \sum_{j=0}^{t-1} 2^j x_j$ with $x_j \in \{0, 1\}$, we classically precompute $c_j = a^{2^j} \bmod N$, so that $a^x \bmod N = (\prod_{j=0}^{t-1} c_j^{x_j}) \bmod N$. Starting with the output register in $|1\rangle$, modular exponentiation therefore requires $t = O(n)$ in-place multiplications $M_{c_j,N}|y\rangle = |c_j y \bmod N\rangle$, each controlled by x_j. Since the c_j are classical constants, each operation multiplies a quantum register by a known integer.

To illustrate reversibility, define the modular multiply-add $A_{c,N}|y\rangle|w\rangle = |y\rangle|(w + cy) \bmod N\rangle$ for $0 \le y, w < N$. Since $\gcd(c, N) = 1$, the inverse $c^{-1} \bmod N$ exists, allowing in-place multiplication through a compute-swap-uncompute circuit:



After the swap, the registers contain $|cy \bmod N\rangle|y\rangle$. The final block subtracts $c^{-1}(cy \bmod N)$ modulo N from the second register, restoring it to zero. Internal workspace is omitted and must likewise be restored.

Schoolbook arithmetic requires $O(n^2)$ gates per modular multiplication, giving $O(n^3)$ gates for modular exponentiation. The construction of Kahanamoku-Meyer & Yao (2024) improves this scaling through recursively implemented phase operations.

Multiplication through phase operations. Let c be a classical integer, let y be stored in an n-qubit register, and let w be stored in a b-qubit accumulator, where $b = O(n)$. Define

$$B_c^{(b)}|y\rangle|w\rangle = |y\rangle|(w + cy) \bmod 2^b\rangle. \tag{4}$$

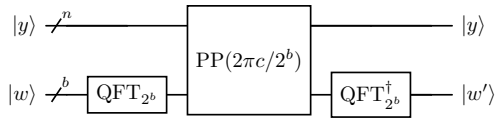
Using the Fourier-transform convention

$$\mathrm{QFT}_{2^b}|w\rangle = \frac{1}{\sqrt{2^b}} \sum_{z=0}^{2^b-1} e^{2\pi i w z/2^b}|z\rangle,$$

adding cy to the accumulator is equivalent to multiplying each Fourier-basis amplitude by $e^{2\pi i c y z/2^b}$. This is precisely the operation

$$\mathsf{PhaseProduct}(\phi)|y\rangle|z\rangle = e^{i\phi y z}|y\rangle|z\rangle$$

with $\phi = 2\pi c/2^b$. Consequently, $B_c^{(b)}$ has the following decomposition, where PP abbreviates PhaseProduct and $w' = (w + cy) \bmod 2^b$:



For example, $b = 2n$ is sufficient to store the product of two n-bit integers when the accumulator starts at zero. The classical multiplier c appears only in the phase angle; it does not require an additional quantum input register.

As described in Section 2.2, the quantum Toom-Cook construction implements PhaseProduct by splitting each operand into k chunks and replacing one product with $2k - 1$ smaller products. The interpolation table specifies how to construct the required evaluations, apply the recursive phase operations, and restore the registers.

For fixed k and fixed interpolation points, the evaluation operands have at most $\lceil s/k \rceil + d_k$ bits, where d_k is independent of the parent width s. The gate count $P_k(s)$ therefore satisfies a recurrence of the form

$$P_k(s) \leq (2k-1)P_k(\lceil s/k \rceil + d_k) + O_k(s).$$

Here the linear term accounts for the local arithmetic, and the recursion stops below a fixed width. Hence

$$P_k(s) = O(s^{\alpha_k}), \qquad \alpha_k = \log_k(2k-1).$$

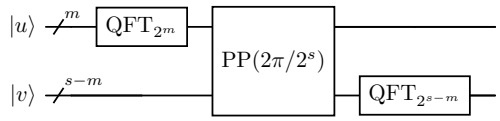
The case $k = 2$ gives the Karatsuba exponent $\alpha_2 = \log_2 3$. More generally, $\alpha_k \to 1$ as $k \to \infty$, so for every fixed $\epsilon > 0$ a sufficiently large fixed k gives $P_k(s) = O(s^{1+\epsilon})$.

A recursively implemented quantum Fourier transform. Improving the phase operation alone is insufficient for the multiply-add circuit: a textbook exact QFT would still contribute $O(b^2)$ gates. The same PhaseProduct primitive also removes this quadratic cost.

Consider an s-qubit input split into two registers, representing the integer

$$q = 2^{s-m}u + v, \qquad 0 \leq u < 2^m, \quad 0 \leq v < 2^{s-m}.$$

A Cooley-Tukey decomposition first applies QFT_{2^m} to the upper register, then a phase product between the two registers, and finally $\mathrm{QFT}_{2^{s-m}}$ to the lower register (Cleve & Watrous, 2000):



In this diagram the upper output register contains the m least significant output bits, and the lower register contains the remaining $s - m$ most significant bits. A final exchange of the two output blocks, omitted from the diagram, gives the conventional ordering of QFT_{2^s}.

Choosing $m = \lfloor s/2 \rfloor$ and recursively implementing both smaller transforms gives

$$Q_k(s) = Q_k(\lfloor s/2 \rfloor) + Q_k(\lceil s/2 \rceil) + O(s^{\alpha_k}),$$

where the final term includes the phase product and any required wire permutation. Since $\alpha_k > 1$,

$$Q_k(s) = O(s^{\alpha_k}).$$

Thus the QFT, its inverse, and the Fourier-basis multiply-add all have the same asymptotic exponent.

Overall gate complexity. The power-of-two arithmetic in Eq. 4 is not itself multiplication modulo N. To obtain the latter, we use the modular construction of Kahanamoku-Meyer & Yao (2024), which combines fast multiplication with finite-precision arithmetic and reversible corrections. Their construction implements in-place modular multiplication to a chosen accuracy while retaining the

subquadratic gate scaling. The precision is chosen so that the accumulated arithmetic error across the $t = O(n)$ controlled multiplications remains bounded.

Writing $M_k(n) = O(n^{\alpha_k})$ for the resulting modular multiplication cost, one invocation of the quantum order-finding circuit has gate count

$$G_{\text{order}}(n) = O(tM_k(n) + Q_k(t) + t) \tag{5}$$

$$= O\big(n^{1+\alpha_k}\big)\,. \tag{6}$$

For every fixed $\epsilon > 0$, choosing a sufficiently large fixed k therefore yields

$$G_{\text{order}}(n) = O(n^{2+\epsilon}).$$

The constants depend on k, so this asymptotic statement does not imply that larger chunk counts are preferable at every finite operand width. These are abstract logical-gate bounds; compiling phase rotations into a fault-tolerant gate set requires an additional precision and resource analysis.

Connection to the present work. The interpolation tables optimized in this paper specify the local arithmetic within the recursive PhaseProduct blocks. Such blocks occur both in classical–quantum multiplication and in the recursive QFT. Improving their schedules can therefore reduce the cost of the arithmetic used by order finding without changing the high-level algorithm.

Our reported numerical gate reductions concern the multiplication resource model defined in Section 4.2. They are not end-to-end percentage reductions for Shor's algorithm: establishing those would additionally require accounting for modular-arithmetic corrections, controlled operations, approximation errors, and gate synthesis.

B A SIMPLE EXAMPLE TABLE

We illustrate the correctness conditions of Section 3 with $k = 2$. Each operand is split into two chunks, $x = x_0 + 2^m x_1$, so $p_x(t) = x_0 + x_1 t$. Since $2k - 1 = 3$, the table uses three interpolation points, 0, 1, and ∞, whose evaluations and coefficient rows are

$$p_x(0) = x_0 \;\; (1,0), \qquad p_x(1) = x_0 + x_1 \;\; (1,1), \qquad p_x(\infty) = x_1 \;\; (0,1).$$

Two of these rows are present in the registers from the start; the row $(1,1)$ must be built.

Step	Instruction	Register 0	Register 1
	(initial)	$(1,0)$: x_0	$(0,1)$: x_1
1	`phaseProduct 0`	$(1,0)$: x_0	$(0,1)$: x_1
2	`phaseProduct 1`	$(1,0)$: x_0	$(0,1)$: x_1
3	`addScaled 0 1 false 0`	$(1,1)$: $x_0 + x_1$	$(0,1)$: x_1
4	`phaseProduct 0`	$(1,1)$: $x_0 + x_1$	$(0,1)$: x_1
5	`addScaled 0 1 true 0`	$(1,0)$: x_0	$(0,1)$: x_1

Table 3: Execution of the $k = 2$ table. The same program acts on the registers of z.

Table B traces the `State` through the program. It also satisfies all the conditions mentioned in Section 3.2:
Valid Points. There are $2k - 1 = 3$ distinct points, and each is allowed: 0, ∞, and $1 = 2^0$.
Every point is used. The three `phaseProduct` instructions consume the points $0, \infty, 1$, a permutation of the declared list.
Correct row at each use. At step 1, register 0 holds $(1,0)$, the row of point 0. At step 2, register 1 holds $(0,1)$, the row of ∞. At step 4, register 0 holds $(1,1)$, the row of point 1.
Restoration. After step 5, the state is again the identity, so both registers hold their original chunks.

C EXPANDING THE PUBLISHED TABLES

C.1 EXPANSION RULES

The published tables use a condensed notation that the operation language cannot express in one step. `addScaled(dst, src, negSrc, shift)` computes $\text{dst} \leftarrow \text{dst} \pm 2^{\text{shift}} \cdot \text{src}$, which gives four expansion rules:

- scaling the *source* by a signed power of two requires no additional operation, since the sign and shift are arguments to `addScaled`;
- scaling the *destination* ("to $2x$ reg. 0", "to $-1x$ reg. 1") costs an additional `shiftL`, `negate`, or both;
- a non-power-of-two coefficient ("add $3x$", "$6x$", "$-15x$", "$48x$") costs one `addScaled` per term of its signed power-of-two decomposition;
- "divide reg. i by 2^m" is one `shiftR`.

Counting raw table rows therefore does not give a baseline in the same units as ours:

Source table	Target	Raw rows	Expanded
Table IV	`PhaseProduct` $k = 3$	14	15
Table V	`PhaseTripleProduct` $k = 3$	19	27
Table VI	`PhaseTripleProduct` $k = 4$	31	58

C.2 Table V Expansion

Table V is expanded row by row below and underlies the 29.6% comparison. The table below normalizes each arithmetic row into the operation language using the four rules above. Product rows are omitted because they do not contribute arithmetic operations.

Row as printed	Normalized	Ops
Add reg. 0 to reg. 1	`addScaled`	1
Add reg. 2 to reg. 1	`addScaled`	1
Add $-1x$ reg. 2 to reg. 1	`addScaled`	1
Add $2x$ reg. 0 to $-1x$ reg. 1	`negate`, `addScaled`	2
Add reg. 1 to reg. 2	`addScaled`	1
Add reg. 2 to reg. 1	`addScaled`	1
Add $2x$ reg. 0 to reg. 1	`addScaled`	1
Add reg. 0 to $2x$ reg. 2	`shiftL`, `addScaled`	2
Add reg. 2 to $-2x$ reg. 1	`negate`, `shiftL`, `addScaled`	3
Add $2x$ reg. 2 to reg. 1	`addScaled`	1
Add reg. 1 to $-4x$ reg. 2	`negate`, `shiftL`, `addScaled`	3
Add $2x$ reg. 1 to reg. 2	`addScaled`	1
Add $8x$ reg. 0 to reg. 2	`addScaled`	1
Add $2x$ reg. 0 to reg. 2	`addScaled`	1
Add reg. 2 to $-1x$ reg. 1	`negate`, `addScaled`	2
Add $-1x$ reg. 0 to reg. 2	`addScaled`	1
Add $-1x$ reg. 1 to $2x$ reg. 2	`shiftL`, `addScaled`	2
Divide reg. 1 by 4	`shiftR`	1
Divide reg. 2 by 8	`shiftR`	1
Total		**27**

This gives seventeen `addScaled`, four `negate`, four `shiftL`, and two `shiftR` operations. Applying the same rules to Table IV gives 15 operations and to Table VI gives 58, so the same convention reproduces all three baselines.

D TABLE SEARCH BENCHMARK

Table 4: Arithmetic operation count, product layers, and weighted cost (Eq. 1) of the champion table for each chunk count k, for both product modes.

	`PhaseProduct`			`PhaseTripleProduct`		
k	Arith.	Layers	Cost	Arith.	Layers	Cost
2	2	2	12	4	2	14
3	11	2	21	16	3	31
4	26	3	41	38	4	58
5	43	4	63	65	6	95
6	64	5	89	93	7	128
7	91	6	121	136	9	181
8	120	7	155	174	10	224
9	155	8	195	231	12	291
10	192	9	237	279	13	344
11	235	10	285	350	15	425
12	280	11	335	408	16	488
13	331	12	391	493	18	583
14	384	13	449	561	19	656
15	443	14	513	660	21	765
16	504	15	579	738	22	848

E PLANNER OUTPUT BY WIDTH

The table below reports the dynamic program of Eq. 2 over the archived `PhaseProduct` policies using the declared cost-model version. "Calls" is the total number of recursive calls across all levels, and "Selected k" gives the chunk count chosen at each level from the root down.

No single chunk count is best at all widths. The root choice increases across these widths, while deeper levels use smaller chunk counts. Restricting the planner to the champion set leaves the totals at $2{,}048$, $8{,}192$, and $16{,}384$ bits unchanged, while increasing the $1{,}024$-bit total by 0.03% and the $4{,}096$-bit total by 0.44%. The selected chunk counts differ only at $4{,}096$ bits, where the champion-only planner chooses $7, 4, 2, 2$ rather than the $6, 4, 2, 2$ shown above.

The Lean implementation shipped with the environment contains the champion set, so it reproduces three of these five rows exactly and differs by at most 0.44% on the other two.

These totals are not comparable to the head-to-head results in Table 2, where each plan recurses on a single schedule, or to the fault-tolerant estimates quoted in Section 2.