Supplementary Information for:
Symbolic Analysis of Grover Search Algorithm via Chain-of-Thought Reasoning and Quantum-Native Tokenization

SUPPLEMENTARY INFORMATION

1. GROVER'S QUANTUM SEARCHING ALGORITHM

Grover's quantum search algorithm theoretically provides a quadratic speed-up over its classical counterpart for unstructured search problems. Specifically, classical searching requires $\mathcal{O}(N)$ query complexity to address a computational problem with the input that can be accessed only through the queries, while Grover's algorithm only uses $\mathcal{O}(\sqrt{N})$ evaluations. This advantage is significant when there is a big problem size N.

Generally, there are two main steps to construct the Grover's quantum circuits: the Oracle construction and the diffuser operation construction. The Oracle is responsible for applying a selective phase flip to a designated marked state in the computational basis, which is a critical step in Grover's algorithm. Our implementation of the oracle construction follows the design principles introduced in the QCircuitNet framework [1]. The Python implementation is shown in Supplementary Note 1. Specifically, given a bit-string representing the marked state, the bit-string is first reversed to match the internal qubit indexing convention adopted in

Qiskit. This adjustment ensures that the circuit operations are correctly aligned with the intended computational basis states. Next, we identify the qubits corresponding to the ’0’ entries in the reversed bit-string. An X gate is applied to each of these qubits, thereby transforming the marked state into the all-ones state $|11\ldots1\rangle$. Following this transformation, we apply a multi-controlled Z gate (MCMT-Z) to the circuit. This gate flips the phase of the $|11\ldots1\rangle$ state while leaving all other basis states unchanged. The controlled operation ensures that only the transformed marked state acquires a phase of -1. After the phase flip, the same X gates are reapplied to the qubits to revert the computational basis back to its original configuration. Finally, the resulting circuit is converted into a quantum gate object named "Oracle," which can be seamlessly integrated into the broader Grover search framework. This construction guarantees that the oracle satisfies the necessary condition of Grover’s algorithm: applying a -1 phase to the marked state while leaving the others invariant.

Supplementary Note 1. Python implementation for the oracle construction under single marked state

```python
def create_oracle(n, marked_state):
    oracle = QuantumCircuit(n)
    rev_target = marked_state[::-1]
    zero_inds = [ind for ind, char in enumerate(rev_target) if char == "0"]
    if zero_inds:
        oracle.x(zero_inds)
    oracle.compose(MCMT(ZGate(), n - 1, 1), inplace=True)
    if zero_inds:
        oracle.x(zero_inds)
    oracle_gate = oracle.to_gate()
    oracle_gate.name = "Oracle"
    return oracle_gate
```

Here, we extend the implementation to the case of multiple marked states in Supplementary Note 2. Instead of constructing a single oracle for one marked state, we iterate over a set of marked states and apply the corresponding oracle subroutine for each. For each bit-string in the list of marked states, we follow the same transformation as in the single-state case: the bit-string is first reversed to match Qiskit’s internal qubit ordering, and X gates are applied to the qubits corresponding to ’0’ entries to convert open controls into closed controls. Afterward, a multi-controlled Z gate is applied to flip the phase of the mapped $|11\ldots1\rangle$ state. The applied X gates are then reversed to restore the original computational basis. By repeating this process across all marked states, the resulting oracle circuit introduces a -1 phase to each of the target states, allowing Grover’s algorithm to function in the multiple marked states searching setting.

Supplementary Note 2. Python implementation for the oracle construction under multiple marked states

```python
def create_oracle(n, marked_states):
    """Create␣an␣oracle␣circuit␣for␣multiple␣marked␣states."""

    # Initialize an empty oracle circuit with n qubits
    oracle = QuantumCircuit(n)

    # Loop through each marked state to construct corresponding oracle operations
    for marked_state in marked_states:
        # Reverse the marked state bit-string to match Qiskit’s bit-ordering convention
        rev_target = marked_state[::-1]

        # Find indices of bits set to ’0’ (these require open controls)
        zero_inds = [ind for ind, char in enumerate(rev_target) if char == "0"]

        # Apply X-gates on qubits corresponding to ’0’ to convert open controls to closed
           ↪ controls
        if zero_inds:
            oracle.x(zero_inds)

        # Apply multi-controlled Z gate (MCMT) with n-1 controls and 1 target qubit
        oracle.compose(MCMT(ZGate(), n - 1, 1), inplace=True)

        # Re-apply X-gates to restore qubits to original state
        if zero_inds:
            oracle.x(zero_inds)

    # Convert constructed oracle to a gate object for modular use
    oracle_gate = oracle.to_gate()
```

```
    oracle_gate.name = "Oracle"

    return oracle_gate
```

2. LARGE LANGUAGE MODELS, THEIR SUPERVISED FINE-TUNING AND PARAMETER-EFFICIENT FINE-TUNING

General-purpose Large Language Models (LLMs). LLMs have emerged as powerful computational systems capable of understanding and generating human language at a giant scale. These models, built upon the Transformer architecture [2], leverage self-attention mechanisms to capture complex linguistic patterns and semantic relationships. Modern LLMs such as LLaMA [3], GPT [4], and DeepSeek [5], contain billions of parameters trained on vast textual corpora using self-supervised learning objectives. The auto-regressive pre-training approach establishes their general linguistic capabilities through next-token prediction tasks, creating representations that capture syntactic structures, factual knowledge, and even reasoning capabilities [5–7]. This architecture enables LLMs to generalize across diverse domains, making them suitable for adaptation to specialized applications such as quantum computing, where our work demonstrates that LLMs can effectively learn and reason about quantum algorithms (*e.g.*. Grover's search) when provided appropriate QASM code examples and chain-of-thought demonstrations.

LLMs for Quantum Computing. Recently, an increasing number of studies have begun exploring how LLMs can contribute to the field of quantum computing. QGAS [8] proposes high-performance ansatz architectures tailored for quantum chemistry and quantum finance tasks. GPT-QE [9] focuses on generating quantum circuits with specific desired properties for quantum simulations. AdaInit [10] alleviates the barren plateau problem [11–14] in quantum machine learning by providing effective initialization parameters for quantum neural network models. QAOA-GPT [15] demonstrates the potential of the Generative Pre-trained Transformer framework to generate high-quality quantum circuits for solving quadratic unconstrained binary optimization (QUBO) problems. GroverGPT [16] explores the boundary of classical simulatability by leveraging the pattern recognition capabilities of LLMs and novel prompt design strategies to simulate Grover's algorithm. Building upon this, GroverGPT+ eliminates the need for explicit prompt guidance, introducing techniques that reveal how LLMs can comprehend the underlying logic of quantum algorithms while effectively simulating Grover's algorithm.

Supervised Fine-Tuning (SFT). SFT refines pre-trained LLMs for specific tasks using high-quality labeled data. Unlike pre-training, which relies on self-supervised objectives, SFT applies direct supervision with input-output pairs curated to guide model behavior toward desired outputs [17–19]. This process typically requires significantly fewer samples than pre-training but depends critically on data quality and alignment with target applications. SFT has proven effective for adapting general LLMs to specialized domains, including programming [20–23], mathematics [4, 22, 23], and scientific applications [24, 25]. The technique updates the model's weights to better align with domain-specific knowledge while preserving general capabilities established during pre-training. In our work, we leverage SFT to adapt LLaMA to understand quantum computing patterns in QASM format and analyze circuits and infer outputs of Grover's algorithm.

Parameter-Efficient Fine-Tuning (PEFT). PEFT addresses computational limitations of conventional full LLM model fine-tuning by updating only a small subset of the model parameters while keeping most weights frozen [17, 18, 26, 27]. Techniques such as Low-Rank Adaptation (LoRA) [17], Prefix Tuning [27], and adapter modules [28] significantly reduce memory requirements and computational costs while maintaining performance comparable to full fine-tuning. PEFT methods typically introduce trainable matrices that modify the forward pass of frozen layers through low-rank decomposition or adapter architectures. These approaches have democratized LLM adaptation by enabling fine-tuning on consumer hardware and facilitating efficient domain adaptation [29]. Our work leverages LoRA to adapt LLaMA to the quantum computing domain (*i.e.* Grover's algorithm) efficiently, enabling the model to understand QASM representations of Grover's algorithm, generate appropriate chain-of-thought reasoning, and generate outputs through symbolic analysis.

3. COMPARISON BETWEEN GROVERGPT AND GROVERGPT+

We provide a comparison between GroverGPT [16] and our proposed GroverGPT+. Supplementary Table I summarizes the key distinctions across task definition, evaluation focus, model scale, data scale, and the relative parameter increase induced by tokenizer extension. This highlights that the two methods should be regarded as complementary. Regarding the extra parameters, we estimate through following calculations:

LLaMA-8B has a hidden size or so-called embedding dimension of $d = 4096$. Each new token introduces one additional embedding vector of dimension d, as well as one additional row in the tied output projection matrix (LM head). Hence, the parameter increase per token is

$$2 \times d = 2 \times 4096 = 8192.$$

With $n = 266$ new tokens, the total number of additional parameters is

Supplementary Table I. Comparison between GroverGPT and GroverGPT+.

Aspect	GroverGPT	GroverGPT+
Task	Approximate simulation of Grover's algorithm outcomes	Symbolic analysis of the step-by-step quantum logic of Grover's algorithm
Task focus	Approximation accuracy	Interpretability and logical correctness of the reasoning trace
Model scale	8 billion parameters	8 billion parameters
Data scale	3–20 qubits w/o analytic trace	Analytic CoT Data, **with Full-circuit Input:** 2–7 qubits, **with Oracle-only Input:** 2–13 qubits
Relative parameter increase induced by extending tokenizer	N/A	0.027%

$$n \times 8192 = 266 \times 8192.$$

Carrying out the multiplication,

$$266 \times 8192 = 2{,}179{,}072 \approx 2.18\text{M}.$$

Thus, extending the vocabulary with 266 tokens induces approximately 2.18 million new parameters in LLaMA-8B. Relative to the full LLaMA-8B model, which contains approximately 8×10^9 parameters, the increase is

$$\frac{2.179 \times 10^6}{8 \times 10^9} \approx 2.72 \times 10^{-4} \approx 0.027\%.$$

Hence, extending the vocabulary by 266 tokens adds only about 0.03% to the total parameter count of LLaMA-8B, a negligible increase.

4. RELATION BETWEEN CLASSICAL FIDELITY AND QUANTUM STATE FIDELITY

For quantum states expressed as its density matrix forms ρ and σ, the fidelity is

$$F(\rho,\sigma) \;=\; \Big(\mathrm{tr}\sqrt{\sqrt{\rho}\,\sigma\,\sqrt{\rho}}\Big)^2, \tag{1}$$

and for pure states $\rho = |\psi\rangle\langle\psi|$, $\sigma = |\phi\rangle\langle\phi|$ it reduces to $F(\rho,\sigma) = |\langle\psi|\phi\rangle|^2$. Let $\psi_i = \langle i|\psi\rangle$ and $\phi_i = \langle i|\phi\rangle$ be amplitudes in the computational basis, and define $p_i = |\psi_i|^2$, $q_i = |\phi_i|^2$. By Cauchy–Schwarz [30],

$$|\langle\psi|\phi\rangle| = \Big|\sum_i \psi_i^* \phi_i\Big| \;\le\; \sum_i |\psi_i||\phi_i| \;=\; \sum_i \sqrt{p_i q_i}\,,$$

hence

$$F(\rho,\sigma) \;=\; |\langle\psi|\phi\rangle|^2 \;\le\; \Big(\sum_i \sqrt{p_i q_i}\Big)^2 \;=\; CF(p,q).$$

Equality holds if and only if the component-wise relative phases are aligned in this basis, *i.e.*, there exists a global phase θ with $\phi_i = e^{i\theta}|\phi_i|$ and $\arg\psi_i$ equal modulo θ for all i, in which case the Classical Fidelity (CF) coincides with the quantum state fidelity. In general circuits with nontrivial relative phases, CF measures *distributional similarity* appropriate for tasks defined on measurement outcomes, while full state fidelity $F(\rho,\sigma)$ should be used when amplitudes are accessible.

5. PARAMETER EFFICIENT FINE-TUNING WITH LOW-RANK ADAPTATION

When adapting the LLMs for domain-specific applications, such as quantum computing, traditional full fine-tuning methods would require updating all parameters and present significant computational overhead due to the LLMs' extensive parameter counts [21, 31–33], *e.g.* the LLaMA-3 [3] model with 8 billion parameters used in our work. To alleviate this, we use a parameter-efficient fine-tuning (PEFT) technique called Low-Rank Adaptation (LoRA) [17].

Specifically, LoRA builds on the observation that weight updates of pre-trained LLMs typically have low intrinsic dimensionality. Consider a pre-trained LLM weight $\mathtt{W} \in \mathbb{R}^{d\times k}$ in the LLM. Rather than directly modifying $\mathtt{W}$, LoRA introduces a decomposition of the weight update:

$$\mathtt{W}' = \mathtt{W} + \Delta\mathtt{W} = \mathtt{W} + \mathtt{BA}, \tag{2}$$

where $\mathtt{B} \in \mathbb{R}^{d\times r}$, $\mathtt{A} \in \mathbb{R}^{r\times k}$, and rank $r \ll \min(d, k)$. During fine-tuning, we freeze the original weights W and only update the significantly smaller matrices $\mathtt{A}$ and $\mathtt{B}$, which reduces the trainable parameter count from $d \times k$ to $r \times (d + k)$.

For training the model, we apply LoRA only to query and value attention modules to improve efficiency further. We apply supervised fine-tuning to the chain-of-thought data as introduced in Section IV B, with a dataset $\mathcal{D} = (x_i, y_i)_{i=1}^N$, where x_i is the input text and y_i are the expected outputs including intermediate reasoning steps. The training objective is the standard autoregressive language model loss [34, 35]:

$$\mathcal{L}_{\text{SFT}} = -\sum_{i=1}^{N}\sum_{j=1}^{|y_i|}\log p_\theta(y_{i,j}|x_i, y_{i,<j}), \tag{3}$$

where θ represents only the LoRA parameters, $y_{i,j}$ is the j-th token of the i-th response, and $y_{i,<j}$ denotes the preceding tokens in the sequence.

This parameter-efficient supervised fine-tuning approach enables the processing of longer QASM and quantum chain-of-thought sequences in our work while preserving the model's general capabilities. Notably, it achieves comparable performance to full fine-tuning with less than 1% of the trainable parameters.

6. THE RULES OF QUANTUM-NATIVE TOKENIZATION

Our quantum-native tokenizer leverages specifically designed parsing rules to systematically process and tokenize QASM circuit descriptions. This tokenizer enables accurate and efficient parsing of QASM syntax, thus significantly reducing token sequence length and computational overhead. The rules, along with their detailed implementation (provided in Supplementary Note 3), are elaborated below:

Gate Definition Parsing. Gate definitions in QASM typically have the format:

```
gate gate_name(parameter_list) qubit_list {
```

To accurately parse these definitions, we utilize regular expressions to capture three main components:

- *Gate Name*: Extracted as a single token.
- *Parameters*: Extracted individually if present; otherwise, this component can be empty.
- *Target Qubits*: Qubit arguments are extracted separately, supporting multiple qubit entries.

After extracting these components, the tokenizer performs normalization by removing numerical suffixes that follow specific internal naming conventions, such as `_gate_q_`, `unitary_`, or `mcx_vchain_`. This normalization ensures consistency and compactness, abstracting from redundant indexing information which does not affect semantics.

Operation Command Handling. Standard quantum operation commands in QASM typically appear as follows:

```
operation_name(parameter_list) qubit_list;
```

Each operation command is parsed by first identifying the operation name, optional parameters (when applicable), and targeted qubits separately. This parsing rule uses regular expressions to isolate and tokenize these segments. Similar to gate definition parsing, any numerical suffixes associated with the internal naming conventions are removed to achieve uniformity. This approach ensures each token reflects a meaningful semantic unit rather than arbitrary indexing.

Bracket and Structural Delimiters. Correctly interpreting hierarchical and nested structures in QASM descriptions is crucial, particularly when handling gate definitions and quantum circuit subroutines. To explicitly address this, our parsing rules incorporate handling of structural delimiters, such as opening (`{`) and closing braces (`}`). Each occurrence of such delimiters is tokenized independently, enabling accurate reconstruction and analysis of nested and hierarchical structures within complex quantum circuit definitions.

Empty Lines and Whitespace Management. As part of our robust tokenization approach, the tokenizer explicitly strips and ignores any empty or whitespace-only lines, ensuring only meaningful QASM commands are processed and tokenized.

Error Handling. If the tokenizer encounters any line that does not conform to the recognized patterns (gate definition, standard operation, or structural delimiters), a clear and informative syntax error is raised. This strict rule enforcement helps ensure the integrity and correctness of the parsing and subsequent analyses.

Collectively, these parsing rules effectively reduce redundant tokenization, maintain semantic coherence, and significantly decrease sequence length. Consequently, this enhances computational efficiency during the symbolic analysis of quantum circuits.

Supplementary Note 3. Python implementation for the Rules of Quantum-Native Tokenization

```python
def _tokenize_line(command):
    """Tokenizes a line of quantum assembly command into structured tokens.

    Args:
        command: Input command string to be tokenized

    Returns:
        List of tokens representing the command
    """
    command = command.strip()
    if not command:
        return []

    # Handles gate definitions (e.g., "gate h q {")
    if command.startswith("gate"):
        gate_match = re.match(r"gate\s+(\w+)(?:\s*\((.*?)\))?\s+([^{]+)\s*{", command)
        if not gate_match:
            raise SyntaxError(f"Invalid gate definition: {command}")

        # Extract components from gate declaration
        gate_name = gate_match.group(1)
        params_part = gate_match.group(2) or ""
        qubits_part = gate_match.group(3)

        # Process parameters and qubits
        params = [p.strip() for p in params_part.split(",") if p.strip()]
        qubits = [q.strip() for q in qubits_part.split(",") if q.strip()]

        # Generate tokens and standardize names
        tokens = ["gate", gate_name] + params + qubits + ["{"]
        tokens = [re.sub(r'^(_gate_q_|unitary_|mcx_vchain_)\d+$', r'\1', t) for t in tokens]
        return tokens

    # Handles standard operations (e.g., "h q[0];")
    groups = re.match(r"^(\w+)(?:\((.*?)\))?\s+([^;]+);", command)
    if groups:
        op_name = groups.group(1)
        params = groups.group(2)
        targets = groups.group(3)

        # Build token sequence
        tokens = [op_name]
        if params:
            tokens += ["("] + [p.strip() for p in params.split(",")] + [")"]
        tokens += [t.strip() for t in targets.split(",")]
        tokens = [token for token in tokens if token]

        # Normalize special tokens
        tokens = [re.sub(r'^(_gate_q_|unitary_|mcx_vchain_)\d+$', r'\1', t) for t in tokens]
        return tokens
```

```
    # Handles closing braces
    if command == "}":
        return ["}"]

    raise SyntaxError(f"Unrecognized␣command:␣{command}")
```

7. ADDITIONAL DETAILS ON SCALABILITY

Supplementary Note 4. Python implementation of Unitary simulation baseline

```
def simulate_qasm(qasm_str: str) -> float:
    """Perform␣Unitary␣simulation␣and␣return␣elapsed␣time."""
    st = time.time()
    try:
        processed_code = preprocess_qasm_code(qasm_str)
        qc = parse(processed_code)
        qc.remove_final_measurements(inplace=True)

        unitary = Operator(qc).data
        initial_state = np.zeros(2**qc.num_qubits)
        initial_state[0] = 1
        final_state = unitary @ initial_state
        probabilities = np.abs(final_state)**2
        num_qubits = qc.num_qubits
        states = [f"{i:0{num_qubits}b}" for i in range(2**num_qubits)]
        for state, prob in zip(states, probabilities):
            print(f"|{state}>:␣{prob:.10f}")
    except Exception as e:
        print(f"Simulation␣failed:␣{str(e)}")
        return 0.0
    return time.time() - st
```

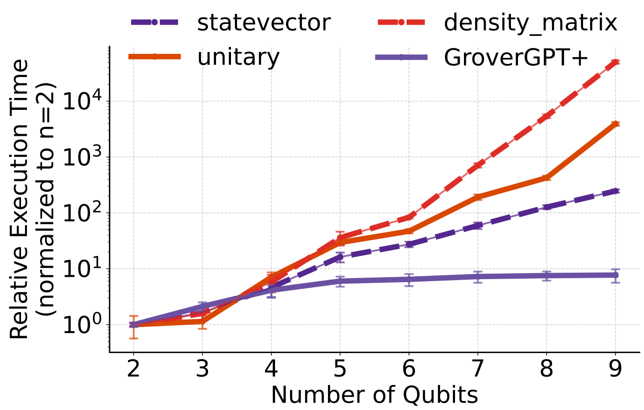
Supplementary Note 5. Python implementation of SV simulation baseline

```
def simulate_qasm(qasm_str: str) -> float:
    """Perform␣Statevector␣simulation␣and␣return␣elapsed␣time."""
    st = time.time()
    try:
        processed_code = preprocess_qasm_code(qasm_str)
        qc = parse(processed_code)
        qc.remove_final_measurements(inplace=True)
        statevector = Statevector(qc)
        probabilities = np.abs(statevector.data)**2
        num_qubits = qc.num_qubits
        states = [f"{i:0{num_qubits}b}" for i in range(2**num_qubits)]
        for state, prob in zip(states, probabilities):
            print(f"|{state}>:␣{prob:.10f}")
    except Exception as e:
        print(f"Simulation␣failed:␣{str(e)}")
        return 0.0
    return time.time() - st
```

Supplementary Note 6. Python implementation of DM simulation baseline

```
def simulate_qasm(qasm_str: str) -> float:
    """Perform␣Density␣Matrix␣simulation␣and␣return␣elapsed␣time."""
    st = time.time()
    try:
        processed_code = preprocess_qasm_code(qasm_str)
        qc = parse(processed_code)
        qc.remove_final_measurements(inplace=True)
```

```
        density_matrix = DensityMatrix(qc)
        probabilities = np.real(np.diag(density_matrix.data))
        num_qubits = qc.num_qubits
        states = [f"{i:0{num_qubits}b}" for i in range(2**num_qubits)]
        for state, prob in zip(states, probabilities):
            print(f"|{state}>:␣{prob:.10f}")
    except Exception as e:
        print(f"Simulation␣failed:␣{str(e)}")
        return 0.0
    return time.time() - st
```



Supplementary Figure 1. Relative execution time of GroverGPT+'s symbolic analysis task compared to traditional classical simulation methods—State Vector, Unitary, and Density Matrix simulations—across different numbers of qubits. All values are normalized to the execution time at 2 qubits. Solid lines show means; discrete error bars indicate uncertainty (mean $\pm$ std).

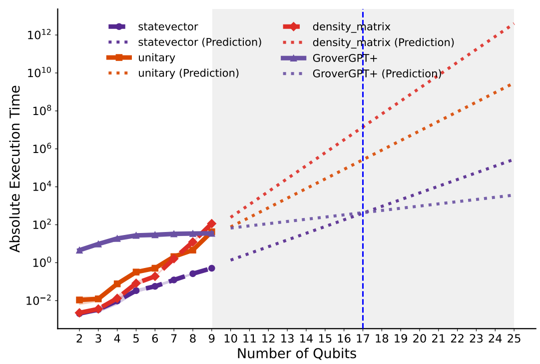
In this section, we provide a detailed comparison of the *computational scalability* of GroverGPT+ against traditional simulation baselines in terms of end-to-end latency, as referenced in Section II E (Please refer to this section for the motivation of this comparison). We compare both relative latency, which highlights scaling trends, and absolute latency, which provides a concrete performance reference. Below we first introduce the baselines and task specifications.

(i) Baselines. We choose some traditional simulation approaches as the baselines, including State Vector (SV), Unitary, and Density Matrix (DM) simulations. Specifically, the SV simulation maintains a length-2^n complex vector and updates it gate by gate, with time complexity $\mathcal{O}(G \cdot 2^n)$ for G gates. This is the standard exact simulation method for pure states. The Unitary simulation explicitly constructs the full $2^n \times 2^n$ unitary matrix corresponding to the circuit and applies it to the initial state. This requires at least $\mathcal{O}(4^n)$ time for a single matrix–vector multiplication, and is therefore only practical for small n, serving as a reference implementation. The DM simulation computes the final density matrix $\rho \in \mathbb{C}^{2^n \times 2^n}$ from which the output probabilities are extracted from the diagonal entries. This method naturally generalizes to noisy or mixed-state settings but incurs even higher cost as $\mathcal{O}(G \cdot 8^n)$ time. Detailed code implementations are provided, see Supplementary Note 4, Supplementary Note 5 and Supplementary Note 6.

(ii) Task Specifications. For this comparison, it is crucial to recognize that the methods perform fundamentally different tasks. The classical baselines execute **numerical state evolution**, a purely computational process to calculate the final state vector. In contrast, GroverGPT+ performs **symbolic analysis**, generating a reasoning trace that interprets the circuit's logic and infers the final probabilities from that understanding. Although the tasks differ, they share a common input format (QASM) and a comparable final output (a probability distribution), which makes a comparison of their end-to-end latency a useful, albeit indirect, point of reference.

(iii) Configurations. The experiments are conducted by measuring the execution time of each method across qubit sizes ranging from 2 to 9. For each method, three runs are performed to compute the mean and standard deviation of execution time. For GroverGPT+, the reported runtimes refer to inference time only and do not include training.

Relative Computational Scalability. Results are similarly plotted on a logarithmic scale to highlight scalability differences. As shown in Supplementary Figure 1, classical methods exhibit steep growth in execution time with increasing qubit numbers. For example, DM and Unitary simulations show sharp exponential scaling, with relative execution times surpassing 10^2 by 7 qubits.



Supplementary Figure 2. Absolute execution time of GroverGPT+ and traditional classical simulation methods, with an extrapolation to visualize the implications of their different scaling trends. Solid lines correspond to measured data (2–9 qubits); dashed lines show extrapolated predictions based on measured data fitting. In the legend, the term **"Prediction"** represents the extrapolated portions of the curves.

In contrast, GroverGPT+ displays a notably gentler slope, maintaining sub-linear growth. Instead of performing tensor-based state evolution, GroverGPT+ generates output probability distributions from its symbolic analysis of the QASM input via CoT reasoning, thereby avoiding the exponential computational overhead inherent to matrix-based simulations. While GroverGPT+ does not defeat exponential complexity in the worst case, it effectively trades offline training for more favorable runtime scaling on its targeted analysis task.

Remark. Due to the inherent overhead of large language models, the absolute runtime of GroverGPT+ on small qubit instances is indeed larger than that of classical simulators. Therefore, we do not claim an absolute runtime advantage. Instead, we focus on the empirical scaling trend *of GroverGPT+, which, in the case of Grover's algorithm, exhibits a surprisingly gentle growth compared to the steep exponential increase of classical methods. This is intriguing as it suggests a complementary perspective on runtime scaling beyond traditional simulation techniques.*

Absolute Computational Scalability. Beyond the measured data, we compare the absolute execution time and perform an extrapolation to visualize where the different scaling trends might lead, see Supplementary Figure 2. In this figure, solid lines show the scaling of measured data points (2–9 qubits), while dashed lines represent extrapolations (predictions) based on the observed data fitting. For clarity of visualization, we do not include shaded regions indicating standard deviation for the curves of measured data, as it does not influence the scaling trend. See the standard deviation of the curves of measured data in Supplementary Figure 1. As for the extrapolated portions, the predicted curves are not actual measurements and do not have associated uncertainty estimates from empirical data. Based on these trends observed within the tested range (2–9 qubits), a **crossover point**, where the latency of GroverGPT+ might hypothetically become favorable compared to the SV simulator, is projected to occur around 17 qubits. We emphasize that this projection is purely speculative and should not be interpreted as a performance claim or guarantee. We present this extrapolation not as a definitive prediction, but as an illustration of the long-term implications of the different scaling behaviors observed within our tested range (2–9 qubits). Notably, this extrapolation for GroverGPT+ is conservative, as it assumes log-linear growth, whereas the observed data in Figure 6 in the main text and Supplementary Figure 1 suggests a sub-linear trend within the tested regime. However, we cannot guarantee that this trend will continue beyond the tested range, and the crossover projection should be viewed as a hypothetical scenario rather than an empirical finding.

Below we detail the hardware and inference setup, relation related to training size and storage and token-length to time complexity and context limitations regarding these comparisons.

i) Hardware and Inference Setup. All classical simulator baselines are executed on a MacBook Pro equipped with the Apple M4 Pro SoC (14-core CPU, 20-core GPU, and 24 GB unified memory). GroverGPT+ inference is performed on an NVIDIA RTX A6000 GPU with 48 GB GDDR6 VRAM. We do not apply batching or parallel inference; all runtimes are measured in single-input, single-instance inference mode.

ii) On Training-set Size and Storage. Reported runtimes refer strictly to inference. We do not include training time or training dataset storage in the benchmarks. While the training corpus has a non-trivial size, its storage footprint does not affect inference latency.

iii) Token-length to Time Complexity and Context Limitations. GroverGPT+ inference time scales approximately linearly with the number of input tokens. Full-circuit QASM inputs grow rapidly in token length (approximately exponential in qubit count), which encounters the model's context-length bottleneck. By contrast, the Oracle-only input format reduces token growth to a more manageable rate, leading to a milder increase in runtime.

8. COMPLETE ANALYSIS PROCESS VIA CHAIN-OF-THOUGHT REASONING

Supplementary Note 7: Chain-of-Thought Analysis Output Analysis

=== Analysis ===
The Oracle entity is extracted below:

```
  x _gate_q_3;
  mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
  x _gate_q_3;
  x _gate_q_1;
  mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
  x _gate_q_1;
```

=== Block 1 ===
Operation sequence:

```
x _gate_q_3;
mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
x _gate_q_3;
```

State construction:

x _gate_q_0: Absent → 1, then → 1
x _gate_q_1: Absent → 1, then → 11
x _gate_q_2: Absent → 1, then → 111
x _gate_q_3: Present → 0, then → 0111

Final state: `0111`
=== Block 2 ===
Operation sequence:

```
x _gate_q_1;
mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
x _gate_q_1;
```

State construction:

x _gate_q_0: Absent → 1, then → 1
x _gate_q_1: Present → 0, then → 01
x _gate_q_2: Absent → 1, then → 101
x _gate_q_3: Absent → 1, then → 1101

Final state: `1101`
=== Final Marked States ===
`0111`
`1101`
=== Results ===

```
{
 '0111': 0.4727,
 '1101': 0.4727,
 '0000': 0.0039,
 '0001': 0.0039,
 '0010': 0.0039,
 '0011': 0.0039,
 '0100': 0.0039,
 '0101': 0.0039,
 '0110': 0.0039,
...
}
```

In this section, we present the complete analysis workflow for Grover's algorithm based on GroverGPT+'s CoT reasoning paradigm. Supplementary Note 7 provides an example of the CoT process. Through this example, we draw some conclusions regarding the capability of LLMs as tools for the symbolic analysis of quantum algorithms:

Conclusion 8.1. *An LLM can effectively perform entity extraction from the quantum circuit representation of an algorithm.*

This conclusion highlights a non-trivial capability of LLMs beyond conventional natural language tasks. In the context of quantum circuit analysis, entity extraction refers to identifying structured and meaningful components (e.g., oracle constructions) from a sequence of low-level QASM. GroverGPT+ demonstrates the ability to parse quantum programs, including those with nested gate definitions, parameterized operations, and qubit registers. Beyond syntactic parsing, it is capable of understanding the semantic structure of circuits, recognizing functionally coherent blocks, such as the Oracle, that are essential to the algorithm's logic. Furthermore, GroverGPT+ can localize and extract specific substructures (entities) from quantum circuits purely based on its learned CoT reasoning, even in the absence of explicit markers or prompts. This suggests that LLMs trained with appropriately designed CoT steps can generalize entity extraction to structured QASM-like inputs.

Conclusion 8.2. *An LLM can effectively distinguish qubit operations associated with marking different target states based on the sequential structure of qubit manipulations.*

In our application, this capability is critical for accurately identifying the marked states, which in turn enables GroverGPT+ to output the corresponding probability amplitudes with high fidelity. As illustrated in Supplementary Note 7, this conclusion captures how GroverGPT+ successfully locates the marked states, as detailed below:

First, GroverGPT+ is able to decompose the full oracle construction into sub-regions, referred to as *blocks*, where each block corresponds to the operations defining a specific marked state. Second, within each block, GroverGPT+ identifies the sequence of single-qubit operations leading up to the application of the multi-controlled multi-target (MCMT) gate. By systematically analyzing the presence (0) or absence (1) of X gates on each qubit, GroverGPT+ incrementally constructs the corresponding computational basis state. The final constructed string (e.g., `0111` as shown in Supplementary Note 7) is recognized as the marked state associated with the block's operations. This demonstrates the model's ability to not only trace quantum operations but also to semantically translate operational sequences into measurable quantum states.

Moreover, in the Qiskit implementation of oracle constructions, bit-string reversal typically occurs twice: first, explicitly within the oracle to match Qiskit's qubit indexing convention; and second, implicitly during final state measurement to restore the original bit-order. To streamline the analysis process, GroverGPT+ effectively consolidates these two reversal steps into a single operation, thereby eliminating redundancy and significantly enhancing reasoning efficiency.

Conclusion 8.3. *An LLM can learn an effective mapping from its symbolic analysis to the algorithm's correct output probability distribution.*

After completing the symbolic analysis to identify key parameters like the number of qubits (n) and the number of marked states (t), GroverGPT+ must infer the final probabilities. This is not a numerical simulation of state evolution, but rather an approximation of a complex mapping that implicitly accounts for factors such as the optimal number of Grover iterations. Analytically, the outcome probabilities for both marked and unmarked states in Grover's algorithm can be determined based on the following mathematical formulation:

Given an n-qubit quantum system, the total number of computational basis states is $N = 2^n$. Suppose there are t marked states within the system. The initial amplitude angle θ is defined as:

$$\theta = \arcsin\left(\sqrt{\frac{t}{N}}\right). \tag{4}$$

The optimal number of Grover iterations k_{opt} that maximizes the success probability is approximately:

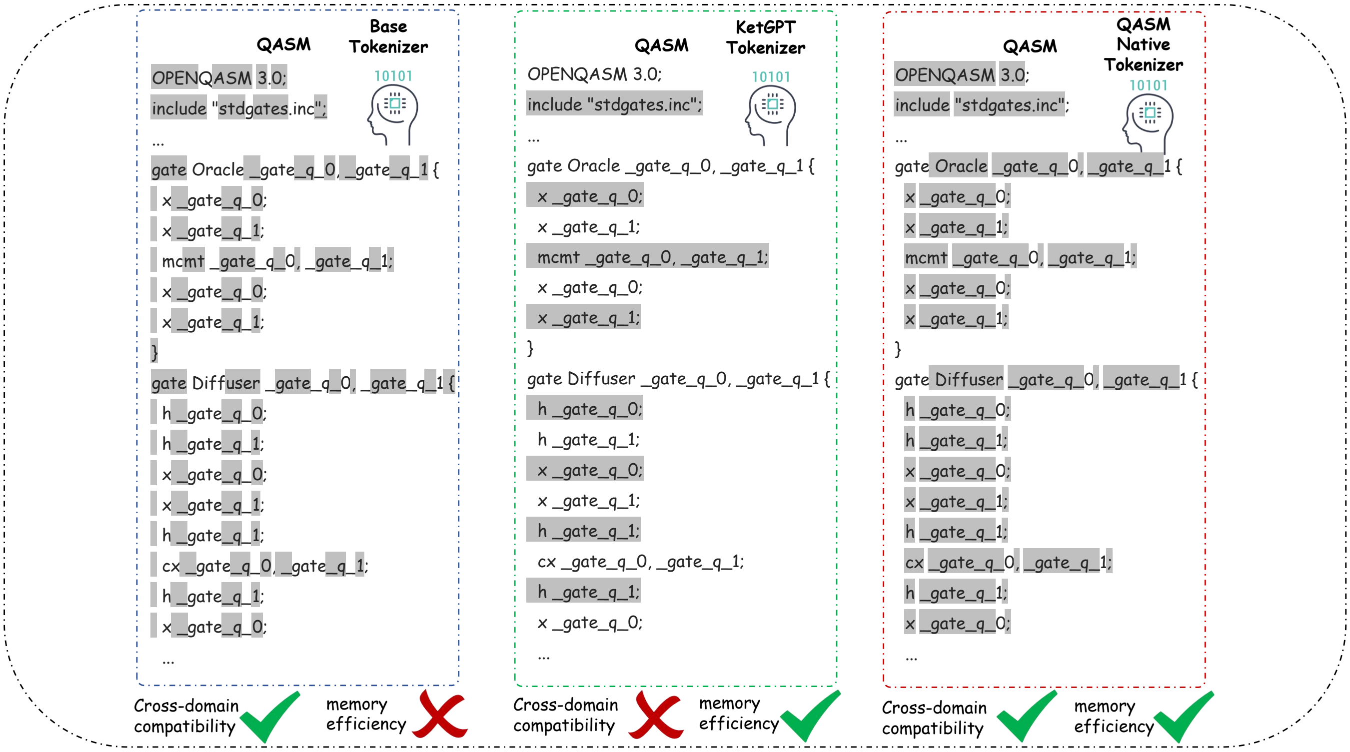
$$k_{\text{opt}} = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{t}} \right\rfloor. \tag{5}$$

After performing k_{opt} iterations, the probability P_{marked} of measuring one of the marked states is given by:

$$P_{\text{marked}} = \sin^2\left((2k_{\text{opt}} + 1)\theta\right), \tag{6}$$

while the probability P_{unmarked} of measuring any unmarked state is:

$$P_{\text{unmarked}} = \cos^2\left((2k_{\text{opt}} + 1)\theta\right). \tag{7}$$



Supplementary Figure 3. Comparison of tokenization strategies on a sample QASM snippet across three tokenizers: LLaMA-3 base tokenizer, KetGPT tokenizer, and the proposed quantum-native tokenizer. Each grey and non-grey segment represents a distinct token. The LLaMA-3 tokenizer demonstrates cross-domain compatibility, applicable to both natural language and QASM domains; however, its non-domain-specific token splitting leads to fragmented semantics and significantly increases input context length, thus consuming more GPU memory. The KetGPT tokenizer effectively reduces the token sequence length and optimizes GPU memory by tokenizing each QASM line as a single token, but it lacks compatibility with natural language and suffers from rapidly increasing model parameters and potential under-training as the QASM length grows. The quantum-native tokenizer, our proposed method, is designed to address these limitations effectively by balancing token granularity, memory efficiency, and domain compatibility.

Thus, based on the number of qubits and the number of marked states identified during the analysis, one can analytically calculate the expected output distribution of Grover's algorithm.

Leveraging its high representational capacity, GroverGPT+ is capable of approximating this complex input–output mapping. Based on the number of qubits, the number of extracted blocks (each corresponding to a marked state), and the identified marked states, GroverGPT+ assigns output probabilities for both marked and unmarked states without explicitly performing quantum evolution. This mapping is learned via parameter-efficient supervised fine-tuning (see *Supplementary Note 5* for more details).

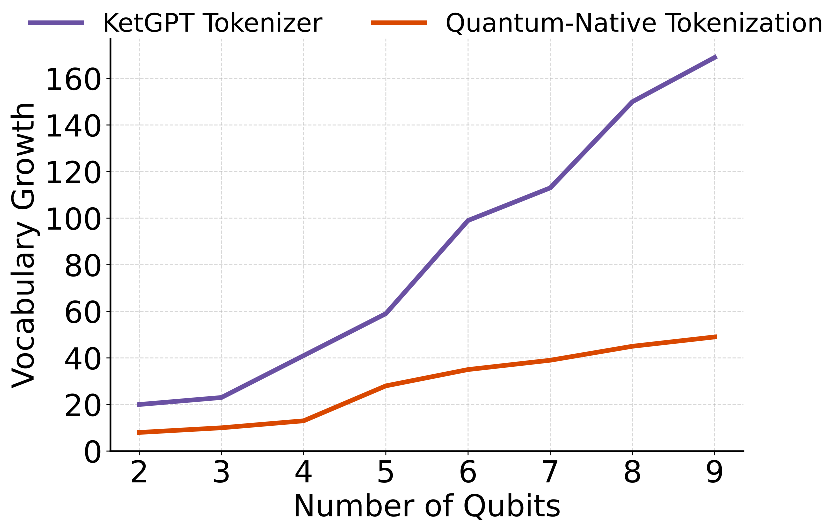
Notably, while GroverGPT+ can generalize this mapping, a slight fidelity loss may occur when analyzing unseen configurations not present in the training data. Empirical results show that GroverGPT+ successfully captures the key characteristics of probability amplitude distributions, distinguishing marked states from unmarked states. Nevertheless, without explicitly calculating the underlying quantum amplitudes, the predicted outputs may exhibit minor deviations compared to ground truth results from classical simulators, particularly when extrapolating to unseen scenarios.

Conclusion 8.4. *Tokenization requires task-specific and cross-domain-aware designs to achieve scalability, memory efficiency, and compatibility with mixed natural and quantum language outputs.*

The design of a tokenizer significantly impacts the efficiency and effectiveness of downstream tasks, especially when dealing with domain-specific representations, such as QASM. As demonstrated in Supplementary Figure 3, three tokenizers, including the base tokenizer, KetGPT [36]'s tokenizer and our quantum-native tokenizer, are compared. KetGPT's tokenizer treats each line of QASM as an individual token ID, facilitating efficiency in the data augmentation task. However, KetGPT's tokenizer may exhibit several limitations when applied to our task as detailed below:

Firstly, KetGPT's line-level tokenization scheme specifically tailors for tasks where both inputs and outputs are purely QASM-based, overlooking the challenges posed by cross-domain tasks where natural language and quantum programming syntax are interleaved. Using a coarse-grained tokenizer for QASM risks interfering with natural language generation. A finer-grained tokenization for QASM elements is therefore essential for enabling flexible decoding across heterogeneous domains. Secondly, it tokenizes entire QASM lines as atomic units. While this may suffice for tasks purely centered on quantum circuit augmentation, it leads to potentially infinite combinations of gate types, qubit indices, and parameters. Consequently, the tokenizer's vocabulary would expand uncontrollably, significantly increasing both the size of the model's embedding table and the total number of trainable parameters. This expansion may result in under-training of the model, as the required data size must scale accordingly with the growth in trainable parameters [37], limiting the model's scalability in practice. To better demonstrate this point, we aim to measure the increase in trainable parameters. Since the size of the embedding layer is directly proportional to the vocabulary size, it suffices to compare the increase in vocabulary size. The statistical analysis is shown below:

We generate a corpus of QASM circuit descriptions with varying numbers of qubits, ranging from 2 to 9 qubits. For each qubit number, we apply both tokenization strategies to the corpus and measure the resulting vocabulary size growth after tokenizing all circuits. The results are presented in Supplementary Figure 4.



Supplementary Figure 4. Vocabulary growth as a function of the number of qubits under different tokenization strategies.

Accordingly, there are several key observations:

The vocabulary size under our quantum-native tokenizer grows slowly with the number of qubits. This reflects the effectiveness of the fine-grained design in capturing the compositional structure of QASM without inducing vocabulary explosion. In contrast, the KetGPT tokenizer shows faster vocabulary growth, driven by the combinatorial increase in unique line patterns as circuit complexity rises. Hence, treating entire QASM lines as atomic tokens may inflate the model's embedding size significantly.

Therefore, combining the analysis for the comparison between the base tokenizer and quantum-native tokenization in Section IV A, designing a domain-aware tokenizer that captures the semantic structures of QASM while being compatible with natural language modeling is crucial. This conclusion highlights that tokenization is not a mere preprocessing step but a foundational design choice.

9. COT TRAINING DETAILS

CoT reasoning is an emergent capability in LLMs that enables them to reason through complex problem-solving tasks by generating step-by-step reasoning paths [5, 7, 38]. Techniques like appending "*Let's think step by step*" to prompts [39] or providing in-context examples [6] can elicit CoT reasoning. Advanced methods such as Self-Consistency [40], Tree-of-Thought [41], and Least-to-Most prompting [42] further improve CoT performance.

CoT Training for GroverGPT+. Yao *et al.* [43] analyzes the advantages and underlying mechanisms of explicit CoT training [44], which refers to the training strategy of explicitly incorporating intermediate reasoning steps before producing the final output. This approach has been shown to significantly improve the reasoning ability and generalization performance of LLMs, especially in tasks that require multi-step logical inference.

In the context of quantum algorithm analysis, CoT training similarly plays a critical role. Unlike the previous approach [16] that directly predicts the final output probabilities from the quantum circuit, we explicitly model the intermediate analysis process through structured reasoning chains. These chains guide the model to generate step-by-step approximations or logical justifications for the predicted outcome. Our implementation of CoT training is primarily based on supervised fine-tuning [45, 46], where high-quality CoT annotations are generated using classical simulators and then used to fine-tune the model (See Supplementary Figure 3). This allows the LLM to internalize the procedural knowledge of quantum circuit analysis.

Furthermore, in *Supplementary Note 10*, we conduct a detailed ablation study to evaluate the effectiveness of each CoT component. The results validate that reasoning chains contribute significantly to the model's accuracy across different circuit depths and oracle configurations, which aligns with the conclusions emphasized in Yao *et al.* [43]: CoT training emerges as a key enabler for the generalization capabilities observed in the GroverGPT+, particularly in its ability to extrapolate to unseen configurations and maintain consistency in output distributions. In our setting, where the input is pure QASM and no additional natural language prompt is provided, CoT training acts as the backbone for aligning symbolic input understanding with semantic reasoning trajectories.

When applying the CoT training technique, we design two types of CoT data:

- **CoT Data with Oracle-only Input**: This dataset contains inputs consisting solely of oracle definitions, with CoT reasoning processes provided as outputs. It is specifically designed to enable GroverGPT+ to acquire the capability of inferring marked states directly from oracle descriptions. This type of training dataset utilized in Section II D in the main text spans qubit numbers $n \in \{2, 3, \dots, 10\}$, ensuring sufficient coverage across small- and medium-scale circuits.

- **CoT Data with Full-circuit Input**: This dataset includes complete QASM circuit descriptions of Grover's algorithm as inputs, paired with CoT reasoning processes as outputs. Its purpose is to guide GroverGPT+ to concentrate specifically on oracle construction and simultaneously enhance its capability. The qubit range for this dataset is restricted to $n \in \{2, 3, \dots, 7\}$, due to the increased token length and complexity of full-circuit inputs.

The data configurations of the qubit range are consistent in all experiments reported in Section II D in the main text width, ensuring consistency across different evaluation settings. In *Supplementary Note 10*, we show that both types of data are indispensable in our training strategy.

10. FULL ABLATION STUDY

This section conducts a full ablation study to systematically examine the contribution of each CoT component and input strategy in the symbolical analysis of Grover's algorithm with GroverGPT+. We specifically verify two major CoT reasoning modules, each corresponding to a key conclusion discussed in *Supplementary Note 8*:

- **Entity Extraction Module** (corresponding to **Conclusion 8.1**): extracting the Oracle entity from the QASM circuit.

- **State Construction Module** (corresponding to **Conclusion 8.2**): constructing the marked computational basis states from sequential qubit operations.

In addition, we evaluate the two input types introduced in Section IV B in the main text

- **CoT Data with Oracle-only Input**: using only the Oracle subcircuit as input.

- **CoT Data with Full-circuit Input**: using the full Grover's algorithm circuit as input.

In each ablation experiment, we remove one CoT module or input type from the training process, keeping all other settings identical. We then evaluate the resulting model behavior by analyzing its generated outputs under the standard label format described below.

Supplementary Note 8: Ablation Label: Without Entity Extraction

=== Block 1 ===
Operation sequence:

```
x _gate_q_1;
mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
x _gate_q_1;
```

State construction:

x _gate_q_0: Absent → 1, then → 1
x _gate_q_1: Present → 0, then → 01
x _gate_q_2: Absent → 1, then → 101
x _gate_q_3: Absent → 1, then → 1101

Final state: `1101`
=== Final Marked States ===
`0111`
`1101`
=== Results ===

```
{
 '0111': 0.4727,
 '1101': 0.4727,
...
}
```

521

Supplementary Note 9: Without State Construction

=== Analysis ===
The Oracle entity is extracted below:

```
x _gate_q_3;
mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
x _gate_q_3;
...
```

=== Results ===

```
{
 '0111': 0.4727,
 '1101': 0.4727,
 '0000': 0.0039,
 '0001': 0.0039,
 '0010': 0.0039,
 '0011': 0.0039,
 '0100': 0.0039,
 '0101': 0.0039,
...
}
```

522

523 Following the above setup, we show a representative output generated by GroverGPT+ under each ablation condition as below:

Supplementary Note 10: An example of the outputs when training without the Entity Extraction Module

=== Analysis ===
=== Block 1 ===
Operation sequence: (Incorrect)
x _gate_q_1;
x _gate_q_2;
x _gate_q_3;
x _gate_q_4;
x _gate_q_5;
x _gate_q_6;
x _gate_q_7;
h _gate_q_0; (H gate should not exist in the Oracle definition)
x _gate_q_1;
x _gate_q_2;
x _gate_q_3;
...

Supplementary Note 11: An example of the outputs when training without the State Construction Module

=== Analysis ===

The Oracle entity is extracted below: (Correct)

x _gate_q_0;
mcmt _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3;
x _gate_q_0;

=== Final Marked States ===
1100 (Though the Oracle is extracted correctly, the searched marked state is incorrect due to the missing of the state construction module)

=== Results ===
(Low Fidelity)

```
{
'1100': 0.9613,
 '0000': 0.0026,
 '0001': 0.0026,
 '0010': 0.0026,
...
}
```

Supplementary Note 12: An example of the outputs when training without the CoT Data with Full-circuit Input

(Incorrect due to not being capable of identifying the analysis task)

The infamous "spaghetti code"!

This is a very long and complex sequence of gates (logic operations) in the Q# programming language, which is used for quantum computing. I'll try to break it down and provide some insights:

Gates

The sequence is composed of various gates, which are the basic building blocks of quantum algorithms. Some common gates include:

* ‘H‘: Hadamard gate, which applies a specific transformation to a qubit (quantum bit).
* ‘CX‘: Controlled-NOT gate, which applies a NOT operation to a target qubit if the control qubit is in the state 1.
* ‘MEASURE‘: Measures the state of a qubit, collapsing the superposition to either 0 or 1.
* ‘RESET‘: Resets the state of a qubit to 0.

...

Supplementary Note 13: An example of the outputs when training without the CoT Data with Oracle-only Input

=== Analysis ===
The Oracle entity is extracted below: (Correct)

x _gate_q_1;
x _gate_q_2;
x _gate_q_7;
x _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3, _gate_q_4, _gate_q_5, _gate_q_6, _gate_q_7;
x _gate_q_1;
x _gate_q_2;
x _gate_q_7;
...

=== Block 1 ===
Operation sequence: (Incorrect)
x _gate_q_1;
x _gate_q_2;
x _gate_q_7;
x _gate_q_0, _gate_q_1, _gate_q_2, _gate_q_3, _gate_q_4, _gate_q_5, _gate_q_6, _gate_q_7; (Missing)
x _gate_q_1; (Missing)
x _gate_q_2; (Missing)
x _gate_q_7; (Missing)

State construction:

...

11. PROMPT DESIGN FOR BENCHMARKING LLMS

To benchmark various LLMs, we designed the following unified prompt:

Supplementary Note 14: Prompt Design for Benchmarking LLMs

You are given a quantum circuit written in OpenQASM 3.0 that implements Grover’s algorithm. Your task is to classically simulate this circuit and return the final output probability distribution over all computational basis states after all gates and before measurement. The output should be a dictionary where each key is a bitstring and each value is the corresponding probability, rounded to four decimal places. Sort the entries in descending order by probability. If the total number of basis states exceeds 30, only return the top 30 most probable states. Include entries with probability 0.0000 if they appear in the top 30.
Output format (example):

```
{
  ’1010’: 0.9732,
  ’0011’: 0.0087,
  ’0110’: 0.0087,
  ...
}
```

Here is the QASM code for simulation:

```
OPENQASM 3.0;
include "stdgates.inc";

gate mcmt _gate_q_0, _gate_q_1 {
  cz _gate_q_0, _gate_q_1;
}
gate Oracle _gate_q_0, _gate_q_1 {
  x _gate_q_0;
  x _gate_q_1;
  mcmt _gate_q_0, _gate_q_1;
  x _gate_q_0;
  x _gate_q_1;
}
gate Diffuser _gate_q_0, _gate_q_1 {
  h _gate_q_0;
  h _gate_q_1;
  x _gate_q_0;
  x _gate_q_1;
  h _gate_q_1;
  cx _gate_q_0, _gate_q_1;
  h _gate_q_1;
  x _gate_q_0;
  x _gate_q_1;
  h _gate_q_0;
  h _gate_q_1;
}
bit[2] c;
qubit[2] q;
h q[0];
h q[1];
Oracle q[0], q[1];
Diffuser q[0], q[1];
c[0] = measure q[0];
c[1] = measure q[1];
```

12. DETAILS OF GENERAL EXPERIMENTAL SETTINGS

We describe the general experimental settings used throughout our study. Specifically, we consider different input types, qubit numbers, and marked states to comprehensively analyze the model performance. We evaluate two types of inputs in our experiments:

- **Full-circuit Input:** The complete QASM code of Grover's quantum search algorithm, including all gate definitions and execution sequences.

- **Oracle-only Input:** A partial QASM sequence that contains only the Oracle construction, excluding other gate definitions and the ordering of gate execution.

The design of the **Oracle-only input** serves two main purposes: 1) This input format is used during training and enables the model to efficiently learn the reasoning steps in the CoT process for analyzing quantum circuits. 2) Considering that LLMs are constrained by a maximum context length (measured in token IDs), the full-circuit input can easily exceed this limit, especially for circuits with a large number of qubits. In contrast, the Oracle-only input allows us to extend the qubit size while remaining within the context boundary, thus facilitating the exploration of the model's extrapolation performance (See *Supplementary Note 15*).

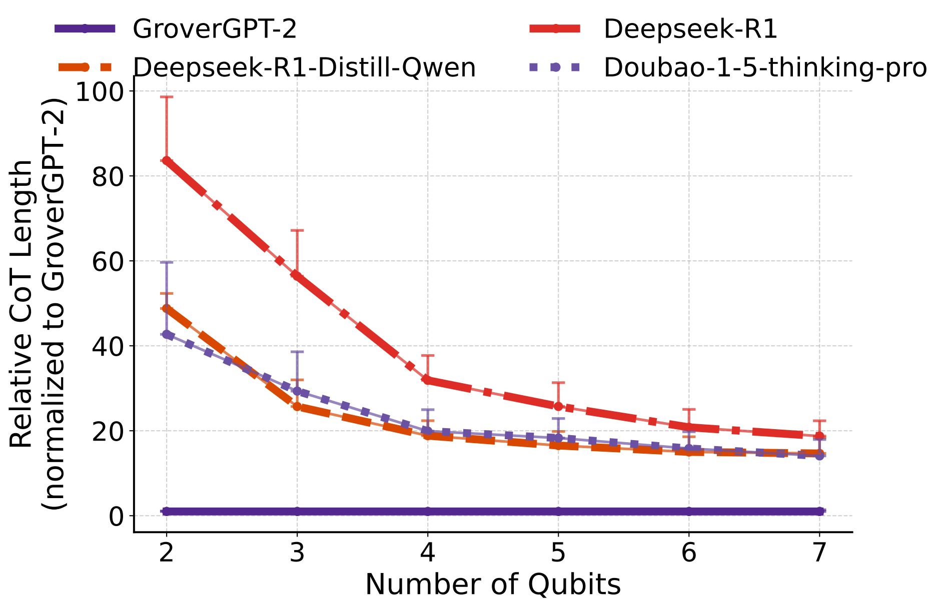
Notably, for both inputs we adopt the analytic solution of the optimal iteration number k_{opt}, which guarantees the maximum success probability for a given (n, t) configuration. This ensures that the output distributions are consistently defined even when the iteration number k is not explicitly present in the oracle-only input for our method.

For the **Full-circuit Input**, we vary the number of marked states in $\{1, 2, 3\}$. The settings for the number of qubits depend on specific experiments. For each configuration, we evaluate on $\max(100, 2^n)$ randomly sampled QASM circuits. The prompting strategy used to guide the baseline LLMs is detailed in *Supplementary Note 11*.

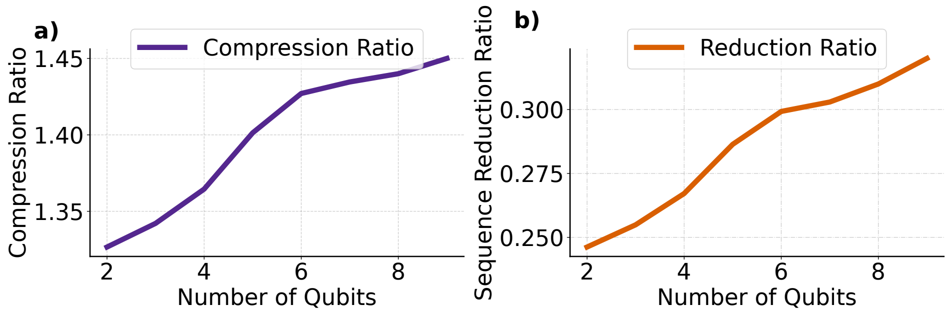
For the **Oracle-only Input**, we use the same set of marked states, and set the number of qubits to $n \in \{2, 3, \dots, 13\}$, allowing us to investigate the model's performance in larger-scale settings. $\max(100, 2^n)$ evaluation samples are also used for each n.

All the experiments adopt a consistent data configuration to train the GroverGPT+ model, as detailed in Section IV B in the main text. Supplementary Note 15 provides a deeper investigation by varying the data configuration.

13. EMPIRICAL STUDY OF THE CHAIN-OF-THOUGHT LENGTH



Supplementary Figure 5. Comparison of the CoT lengths generated by different LLMs for the symbolic analysis of Grover's algorithm across qubit numbers ranging from 2 to 7. The values are normalized with GroverGPT+'s CoT length set as the baseline (1.0) for each qubit count. Solid lines show means; discrete error bars indicate uncertainty (mean $\pm$ std).



Supplementary Figure 6. Compression Ratio (a) and Sequence Reduction Ratio (b) across different number of qubits, showing the efficiency of quantum-native tokenization for token compression.

In this section, we analyze and compare the CoT lengths produced by various LLMs when performing symbolic analysis of Grover's algorithm. The objective is to investigate the computational overhead incurred by different models, particularly since longer CoT sequences can lead to increased latency and inference cost. For this analysis, we collected model-generated CoTs for each circuit and measured their lengths across qubit sizes. Each model's average CoT length was then normalized by GroverGPT+'s CoT length at the same qubit count to provide a consistent relative comparison.

As illustrated in Supplementary Figure 5, GroverGPT+ consistently generates substantially shorter CoT reasoning sequences compared to baseline models. For example, at 2 qubits, GroverGPT+ maintains a normalized CoT length of 1.00, while baseline models often produce sequences that are dozens of times longer—some exceeding 40$\times$ or even 80$\times$ the length. This trend persists across higher qubit counts as well. Even at 7 qubits, GroverGPT+ maintains concise outputs, whereas other models continue to generate significantly longer CoTs, reflecting less focused or more verbose reasoning processes.

We can conclude that even at small qubit sizes, baseline models often require excessively long CoTs to reach a potentially valid answer. This indicates that, without targeted supervision, these models tend to produce verbose and sometimes redundant reasoning chains, reflecting weaker task alignment. Moreover, as evidenced in Figure 3 in the main text, these long CoTs do not guarantee reliable performance, as baseline models still struggle to achieve consistent simulation accuracy, reinforcing that longer reasoning is not necessarily better in this context. By contrast, GroverGPT+'s compact and task-aligned CoTs result from explicit fine-tuning that leverages the structural logic of Grover's algorithm, yielding both faster and more reliable outcomes.

14. EMPIRICAL STUDY OF QUANTUM-NATIVE TOKENIZATION

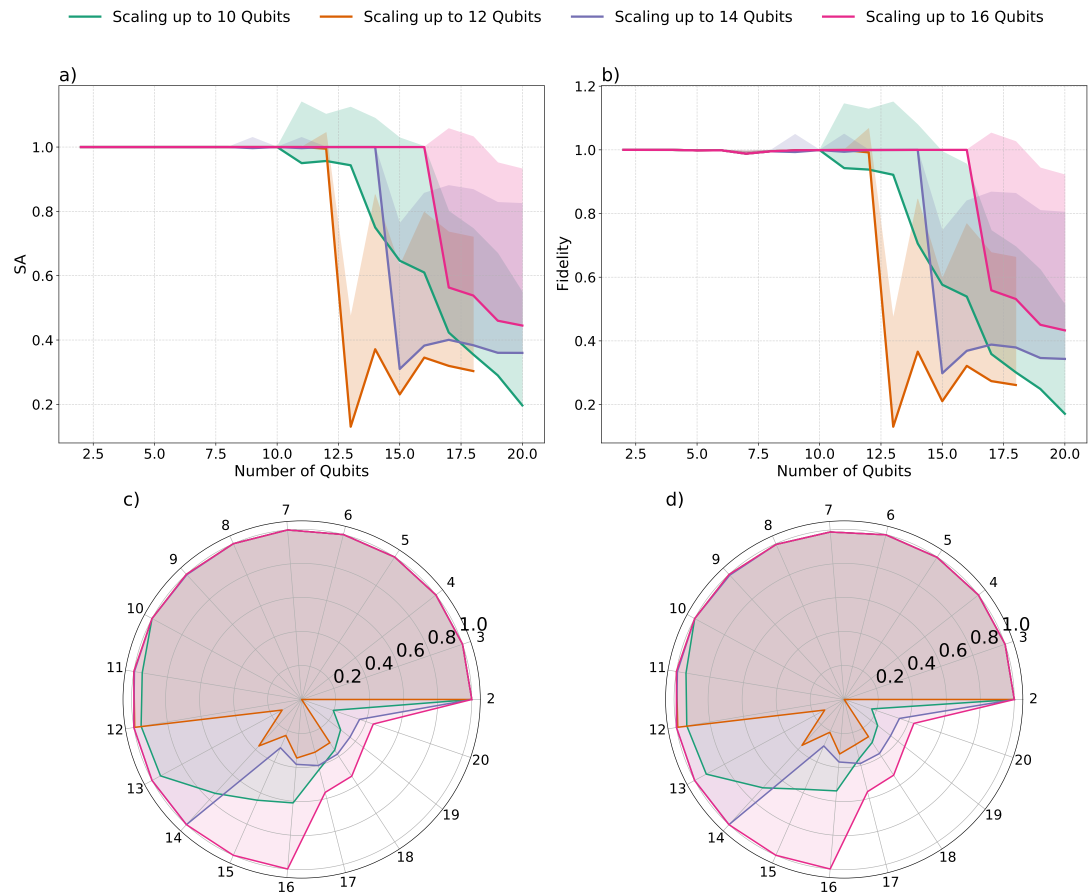
In this study, we investigate how the quantum-native tokenization performs under different number of qubits. This study is conducted following the below steps:

Firstly, we collect all the QASM circuit descriptions for Grover's algorithm under different numbers of qubits. For each given number of qubits n, there are M_n QASM circuit descriptions. Secondly, we tokenize each QASM description using the base tokenizer of the LLaMA-3 model and our designed quantum-native tokenizer to obtain the length of each tokenized sequence. Specifically, for the i-th QASM circuit under n qubits, let $L_{base}^{(n,i)}$ denote the sequence length obtained by the base tokenizer, and let $L_{quantum}^{(n,i)}$ denote the sequence length obtained by the quantum-native tokenizer. Thirdly, we calculate the compression ratio and the sequence reduction ratio for each number of qubits by averaging over all tokenized sequences under the same number of qubits. They can be formally defined as follows:

$$\text{Compression Ratio}_n = \frac{1}{M_n}\sum_{i=1}^{M_n}\frac{L_{base}^{(n,i)}}{L_{quantum}^{(n,i)}} \tag{8}$$

$$\text{Sequence Reduction Ratio}_n = \frac{1}{M_n}\sum_{i=1}^{M_n}\frac{L_{base}^{(n,i)} - L_{quantum}^{(n,i)}}{L_{base}^{(n,i)}} \tag{9}$$

As illustrated in Supplementary Figure 6, both the results measured using the compression ratio and the sequence reduction ratio indicate a higher computational efficiency as the number of qubits increases. Specifically, the compression ratio increases from below 1.35 to above 1.40, indicating that the quantum-native tokenizer consistently produces shorter token sequences compared to the LLaMA-3 base tokenizer, with the advantage becoming more prominent for larger circuits. Correspondingly, the



Supplementary Figure 7. The extrapolation performance of GroverGPT+ under different training data scaling, including the scaling up to 10/12/14/16 qubits.

sequence reduction ratio increases from below 0.26 to above 0.30, meaning that the token length saved by the quantum-native tokenizer becomes more substantial as the circuit complexity grows. This trend can be attributed to the structural design of the quantum-native tokenizer, which recognizes entire gate operations (e.g., `cx q[0],q[1];`) and qubit identifiers as cohesive, semantic units, whereas the LLaMA-3 tokenizer often splits them into meaningless subwords based on natural language rules. As Grover circuits scale with more qubits, the QASM descriptions exhibit increasingly repetitive or patterned structures (e.g., repeated oracle and diffuser subroutines). The quantum-native tokenizer is able to capitalize on these recurring syntactic forms and compress them more efficiently.

These results demonstrate the effectiveness of the proposed tokenizer in reducing sequence length, which directly leads to reduced memory and computational requirements during downstream model fine-tuning or inference. The shorter token sequences not only enable better GPU memory utilization but also reduce the context fragmentation problem inherent in sequences produced by the base tokenizer, particularly beneficial when handling circuits of increasing size and complexity.

15. EXTRAPOLATION PERFORMANCE OF GROVERGPT+

In this study, we investigate how GroverGPT+ performs when extrapolated to larger problem size, specifically as a function of the number of qubits. Specifically, we investigate how the model's performance scales both within the training distribution and when generalizing to unseen problem sizes. Due to the excessive length of full-circuit inputs, which restricts the number of qubits to at most 10, a meaningful evaluation of the extrapolation cannot be reliably observed using data with full-circuit input. Therefore, we focus on the Oracle-only input setting. In particular, we construct four different training sets where the circuits with Oracle-only input cover qubit ranges spanning from 2 to 10, 12, 14 and 16 qubits, respectively. For each setting, we evaluate the model's performance across test circuits ranging from 2 to 20 qubits, measuring its ability to interpolate within the training distribution and extrapolate beyond it. All performance claims are empirically grounded in the tested ranges.

The corresponding results are depicted in Supplementary Figure 7, which reveal the model's performance behavior under different training ranges. For this figure, we use continuous shaded bands as error bars. Our key observations and corresponding analysis are as follows:

- Firstly, the model demonstrates consistently high performance within the training distribution across all experimental conditions. Specifically, as evident in all figures, the evaluation metrics, including the SA and fidelity, reach or closely approach 1.0 within the qubit range included in the training datasets. This confirms that, within the bounds of the trained problem sizes, GroverGPT+ effectively learns to accurately analyze the quantum circuits and infer their outcomes.

- Secondly, when the training dataset is relatively small (e.g., covering qubits from 2 to 10), the model exhibits a noticeable degree of generalization capability to unseen data points. For instance, Supplementary Figure 7 (c, d) show that the model maintains consistently high performance in both SA (approximately 0.95) and fidelity (approximately 0.93) for qubit sizes of 11, 12, and 13, with only minor fluctuations across these larger circuit scales. However, a clear decreasing trend emerges as the circuit complexity grows beyond this range (14 to 20 qubits). Both SA and fidelity show parallel declining trends, suggesting that the generalization capability progressively deteriorates as the problem size increases. Upon inspecting specific output examples (as illustrated in Supplementary Note 15), we observed cases indicating errors in identifying the correct marked states, which likely contributes significantly to the observed decline in both metrics.

- Thirdly, when the dataset size increases further (covering ranges from 2 to 12 and 2 to 14), the high performance within the training range persists. Nevertheless, the generalization ability sharply decreases for larger qubit sizes outside these training distributions (13–20 and 15–20 qubits, respectively). The deterioration patterns for both SA and fidelity also closely resemble each other, suggesting a common underlying issue as above. However, when scaling the training range to 16 qubits, the generalization performance can surpass other ranges, indicating potential advantage of scaling the training size to larger qubit ranges. Similarly, we examine some model outputs, which reveal frequent inaccuracies in marked state reasoning, as exemplified in (as illustrated in Supplementary Note 15).

While inspecting some failed cases similar to Supplementary Note 15, we find that GroverGPT+ performs well on instances involving single or two marked states. However, it fails to correctly identify the marked states when three marked states are required, primarily due to the omission of key reasoning steps in the CoT process. A plausible reason for the decreased generalization could be twofold: on one hand, the inherent increase in the complexity of quantum state representation and inference tasks as the number of qubits and marked states expands beyond the training distribution imposes greater demands on the model's learned representations; on the other hand, the phenomenon of catastrophic forgetting [47, 48] may also contribute, whereby exposure to an enlarged training range (i.e., circuits with higher qubit numbers) potentially disrupts the model's previously acquired capabilities for smaller circuit sizes, thereby impairing its ability to generalize to unseen configurations.

Supplementary Note 15: An example of output that misses key CoT steps

=== Analysis ===

The Oracle entity is extracted below:

... (Correct)

=== Block 1 ===
Operation sequence:
... (Correct)

State construction:
1. x _gate_q_0: Present → 0, then → 0 (Missing)
2. x _gate_q_1: Absent → 1, then → 10 (Missing)
3. x _gate_q_2: Present → 0, then → 010 (Missing)

...
Final state: 00100000 (Incorrect)

...

16. HYPERPARAMETER SETTINGS

In this section, we detail the key hyperparameters used for training GroverGPT+ in our experiments. The model was fine-tuned using LoRA with the configurations in Supplementary Table II:

Supplementary Table II. Hyperparameter Scope for GroverGPT+

Hyperparameters	**Value/Setting**	**Type**
Base Model	Llama-3-8B-Instruct	Fixed*
Optimizer	AdamW	Fixed
Learning Rate η	2×10^{-5}	Fixed
Batch Size B	1	Fixed*
Gradient Accumulation Steps	8	Fixed
Effective Batch Size	8	Derived
Training Epochs E	10	Fixed
Learning Rate Schedule	Cosine	Fixed
Warmup Steps	20	Fixed
Weight Decay	0.0	Fixed
Max Sequence Length	4000	Fixed
FP16 Mixed Precision	Enabled	Fixed
LoRA Target Modules	{*q_proj*, *v_proj*}	Fixed
LoRA Rank r	8 (default)	Fixed†
LoRA Alpha α	32 (default)	Fixed†
Evaluation Strategy	Every 50 steps	Fixed
Evaluation Split	10%	Fixed
Max Gradient Norm	1.0	Fixed
Random Seed	42	Fixed
Dataset Shuffling	Disabled	Fixed

* Effective batch size calculated as $B_{\text{effective}} = B \times$ gradient_accumulation_steps
† Default values from LLaMA-Factory [49] implementation

SUPPLEMENTARY REFERENCES

[1] R. Yang, Y. Gu, Z. Wang, Y. Liang, and T. Li, arXiv preprint arXiv:2410.07961 (2024).
[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need (2023), arXiv:1706.03762 [cs.CL].
[3] A. G. et al., The llama 3 herd of models (2024), arXiv:2407.21783 [cs.AI].
[4] O. et al., Gpt-4 technical report (2024), arXiv:2303.08774 [cs.CL].
[5] D.-A. et al., Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning (2025), arXiv:2501.12948 [cs.CL].
[6] S. S. Nachane, O. Gramopadhye, P. Chanda, G. Ramakrishnan, K. S. Jadhav, Y. Nandwani, D. Raghu, and S. Joshi, Few shot chain-of-thought driven reasoning to prompt llms for open ended medical question answering (2024), arXiv:2403.04890 [cs.CL].
[7] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, Emergent abilities of large language models (2022), arXiv:2206.07682 [cs.CL].
[8] Z. Liang, J. Cheng, R. Yang, H. Ren, Z. Song, D. Wu, X. Qian, T. Li, and Y. Shi, arXiv preprint arXiv:2307.08191 (2023).
[9] K. Nakaji, L. B. Kristensen, J. A. Campos-Gonzalez-Angulo, M. G. Vakili, H. Huang, M. Bagherimehrab, C. Gorgulla, F. Wong, A. McCaskey, J.-S. Kim, *et al.*, arXiv preprint arXiv:2401.09253 (2024).
[10] J. Zhuang and C. Guan, arXiv preprint arXiv:2502.13166 (2025).
[11] M. Cerezo, G. Verdon, H.-Y. Huang, L. Cincio, and P. J. Coles, Nature computational science **2**, 567 (2022).
[12] M. Cerezo, A. Sone, T. Volkoff, L. Cincio, and P. J. Coles, Nature communications **12**, 1791 (2021).
[13] A. Arrasmith, M. Cerezo, P. Czarnik, L. Cincio, and P. J. Coles, Quantum **5**, 558 (2021).
[14] J. Liu, Z. Lin, and L. Jiang, Machine Learning: Science and Technology **5**, 015058 (2024).
[15] I. Tyagin, M. H. Farag, K. Sherbert, K. Shirali, Y. Alexeev, and I. Safro, arXiv preprint arXiv:2504.16350 (2025).
[16] H. Wang, P. Li, M. Chen, J. Cheng, J. Liu, and T. Chen, arXiv preprint arXiv:2501.00135 (2024).
[17] E. J. 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), arXiv:2106.09685 [cs.CL].
[18] P. Li, P. Yadav, J. Yoon, J. Peng, Y.-L. Sung, M. Bansal, and T. Chen, Glider: Global and local instruction-driven expert router (2024), arXiv:2410.07172 [cs.LG].
[19] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer (2023), arXiv:1910.10683 [cs.LG].
[20] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, Code llama: Open foundation models for code (2024), arXiv:2308.12950 [cs.CL].
[21] D.-A. et al., Deepseek-v3 technical report (2025), arXiv:2412.19437 [cs.CL].
[22] D. Kevian, U. Syed, X. Guo, A. Havens, G. Dullerud, P. Seiler, L. Qin, and B. Hu, Capabilities of large language models in control engineering: A benchmark study on gpt-4, claude 3 opus, and gemini 1.0 ultra (2024), arXiv:2404.03647 [math.OC].
[23] X. Zhao, G. Sun, R. Cai, Y. Zhou, P. Li, P. Wang, B. Tan, Y. He, L. Chen, Y. Liang, B. Chen, B. Yuan, H. Wang, A. Li, Z. Wang, and T. Chen, Model-glue: Democratized llm scaling for a large model zoo in the wild (2024), arXiv:2410.05357 [cs.LG].
[24] H. Wang, C. Liu, N. Xi, Z. Qiang, S. Zhao, B. Qin, and T. Liu, Huatuo: Tuning llama model with chinese medical knowledge (2023), arXiv:2304.06975 [cs.CL].
[25] S. Yun, I. Choi, J. Peng, Y. Wu, J. Bao, Q. Zhang, J. Xin, Q. Long, and T. Chen, Flex-moe: Modeling arbitrary modality combination via the flexible mixture-of-experts (2024), arXiv:2410.08245 [cs.LG].
[26] X. Liu, K. Ji, Y. Fu, W. L. Tam, Z. Du, Z. Yang, and J. Tang, P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks (2022), arXiv:2110.07602 [cs.CL].
[27] X. L. Li and P. Liang, Prefix-tuning: Optimizing continuous prompts for generation (2021), arXiv:2101.00190 [cs.CL].
[28] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, Parameter-efficient transfer learning for nlp (2019), arXiv:1902.00751 [cs.LG].
[29] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, Qlora: Efficient finetuning of quantized llms (2023), arXiv:2305.14314 [cs.LG].
[30] H. Alzer, Journal of mathematical analysis and applications **234**, 6 (1999).
[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding (2019), arXiv:1810.04805 [cs.CL].
[32] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, Roberta: A robustly optimized bert pretraining approach (2019), arXiv:1907.11692 [cs.CL].
[33] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, Mixtral of experts (2024), arXiv:2401.04088 [cs.LG].
[34] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, Language models are few-shot learners (2020), arXiv:2005.14165 [cs.CL].

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, Language models are unsupervised multitask learners (2019).
[36] B. Apak, M. Bandic, A. Sarkar, and S. Feld, in *International Conference on Computational Science* (Springer, 2024) pp. 235–251.
[37] V. N. Vapnik and A. Y. Chervonenkis, in *Measures of complexity: festschrift for alexey chervonenkis* (Springer, 2015) pp. 11–30.
[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, Chain-of-thought prompting elicits reasoning in large language models (2023), arXiv:2201.11903 [cs.CL].
[39] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, Large language models are zero-shot reasoners (2023), arXiv:2205.11916 [cs.CL].
[40] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, Self-consistency improves chain of thought reasoning in language models (2023), arXiv:2203.11171 [cs.CL].
[41] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, Tree of thoughts: Deliberate problem solving with large language models (2023), arXiv:2305.10601 [cs.CL].
[42] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, Least-to-most prompting enables complex reasoning in large language models (2023), arXiv:2205.10625 [cs.AI].
[43] X. Yao, R. Ren, Y. Liao, and Y. Liu, arXiv preprint arXiv:2502.04667 (2025).
[44] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, Advances in neural information processing systems **35**, 24824 (2022).
[45] X. Yue, X. Qu, G. Zhang, Y. Fu, W. Huang, H. Sun, Y. Su, and W. Chen, arXiv preprint arXiv:2309.05653 (2023).
[46] L. Yu, W. Jiang, H. Shi, J. Yu, Z. Liu, Y. Zhang, J. T. Kwok, Z. Li, A. Weller, and W. Liu, arXiv preprint arXiv:2309.12284 (2023).
[47] M. McCloskey and N. J. Cohen, in *Psychology of learning and motivation*, Vol. 24 (Elsevier, 1989) pp. 109–165.
[48] R. Ratcliff, Psychological review **97**, 285 (1990).
[49] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, Z. Luo, Z. Feng, and Y. Ma, arXiv preprint arXiv:2403.13372 (2024).