ChatGPT

Quantum LLM Emulation on IBM Q (5–7 Qubits) – MicroGPT Architecture

Introduction

Emulating a Transformer-based large language model (LLM) on quantum hardware is an ambitious but promising endeavor. By leveraging **Andrej Karpathy's micro-GPT** (a minimal GPT implementation) as a blueprint, we aim to map its key components – token embeddings, self-attention, and sequence generation – onto quantum-native counterparts. The motivation is twofold: (1) **Quantum state space growth:** n qubits span a 2^n-dimensional Hilbert space, offering an *exponential* representational capacity [1]. Even IBM's modest 5- and 7-qubit processors can encode high-dimensional vectors (e.g. 6 qubits $\equiv$ 64 dimensions) without storing large weight matrices. (2) **Entanglement for rich correlations:** Quantum entanglement natively captures multi-token correlations that classical attention only approximates via dot-products [2]. Harnessing entanglement and inherent measurement **entropy** (randomness) may enable new modeling capacity, such as capturing higher-order token relationships and injecting stochasticity for text generation. In short, a quantum-hybrid microGPT could potentially match small classical LLM performance with far fewer classical parameters [3], pointing to a future of more efficient NLP models.

However, significant challenges guide the design. Current devices are **NISQ** (noisy intermediate-scale quantum), with limited qubits and decoherence time. We must therefore use shallow **variational quantum circuits (VQCs)** and frequent measurement to mitigate noise and the risk of barren plateaus (flat gradients) [4]. Our plan embraces a **hybrid approach**: classical preprocessing (where needed) plus quantum subroutines for the heavy-lifting (embeddings, attention, feed-forward), similar to the recent QuantumGPTMini prototype [5]. This ensures compatibility with Qiskit's tools and today's IBM hardware. All components will be **entropy-driven** – either exploiting entanglement entropy as an attention metric or using quantum probabilistic measurement to emulate softmax sampling. In the following sections, we provide a deep technical roadmap: theoretical justification of each quantum component, the detailed architecture and circuit designs in Qiskit, training and simulation strategy, deployment on IBM's 5- and 7-qubit backends, and a progression path from prototype to a scalable quantum LLM.

Theoretical Justification for a Quantum LLM

Why use quantum circuits for an LLM? Transformer models are extremely resource-intensive; GPT-3 has 175 billion parameters, and training it "can burn as much electricity as a small city" [6]. Quantum computing offers a potential computational *compression*. With only n qubits we can represent a state in a 2^n-dimensional vector space, enabling an *exotic form of compression* for information [1]. For example, a 7-qubit state lives in a space of dimension 128, which could hold rich latent representations equivalent to classical vectors of length 128 (or higher if using entangled bases). This high-dimensional representation is obtained *physically* rather than by storing $128\times n$ parameters – the quantum state amplitudes encode information implicitly. Thus, certain large linear operations (like big matrix multiplies in an LLM) might be off-loaded to quantum evolutions in a smaller parameter space.

Furthermore, **entanglement** provides a built-in mechanism for encoding correlations. In classical transformers, the self-attention mechanism computes pairwise similarity (query-key dot products) and passes them through a softmax to determine attention weights. Quantum processors can replace this with *entanglement-based attention:* qubits carrying query and key information can be entangled and then measured to produce attention coefficients [2]. Entangled states inherently encode joint relationships; measuring an entangled system yields correlated outcomes that reflect the similarity between subsystems. Recent research indicates that using **entanglement entropy** or other entanglement measures in place of dot-product similarity can improve generalization of attention layers [7] [8]. Intuitively, entanglement measures capture multi-token dependencies (not just pairwise), potentially enabling the model to find complex patterns (QuantumGPTMini indeed found that an entangled attention layer could solve certain logic puzzles better than its classical counterpart) [9].

Entropy-driven modeling in this context means we use quantum probabilistic outcomes and entropic measures as part of the model's logic. A quantum measurement yields a distribution of outcomes (with Shannon entropy) that we can steer via circuit parameters. For example, a maximally entangled state will collapse to each basis outcome with equal probability (high entropy), whereas a less entangled or biased state yields a more peaked distribution. By training the entangling gates, we effectively adjust the output distribution's entropy – analogous to learning attention concentration vs. dispersion. This is analogous to a learnable "temperature" in softmax. Moreover, if we use **mixed quantum states** (density matrices) rather than pure states, we introduce classical randomness (statistical mixtures) on top of quantum uncertainty, increasing modeling flexibility. The *Quantum Mixed-State Self-Attention Network (QMSAN)* exemplifies this by encoding inputs into mixed states and using their overlaps for attention calculation [10]. Mixed states carry non-zero von Neumann entropy, effectively embedding classical uncertainty directly into the quantum representation of attention weights.

In summary, quantum mechanisms offer (a) a compressed high-dimensional vector space, (b) entanglement-based correlation measures (entropic attention mechanisms), and (c) inherent probabilistic generation – all of which align well with the needs of LLMs. The theoretical promise is **parameter efficiency** (smaller models achieving similar perplexity) [3] and possibly new qualitative capabilities from entanglement-driven pattern recognition [11]. These benefits must be weighed against NISQ hardware limits; hence our design minimizes circuit depth and uses hybrid classical control to remain feasible on IBM's 5- and 7-qubit devices.

Quantum MicroGPT Architecture Overview

Our quantum microGPT will mirror the classical microGPT's pipeline: **(1) Token Embedding, (2) Positional Encoding, (3) Self-Attention, (4) Feed-Forward Network,** and iterative **(5) Sequence Generation**. Each stage is implemented with quantum-native methods using Qiskit. Table 1 summarizes the classical vs. quantum design of each component:

LLM Component	Classical MicroGPT	**Quantum Implementation (This Work)**
Token Embedding	One-hot token mapped through an embedding matrix to a d-dimensional vector.	Encode token into an n-qubit quantum state. Use **angle encoding** (features $\to$ qubit rotations) for NISQ robustness [12], or **amplitude encoding** to pack a full vector into $O(\log d)$ qubits [13]. The quantum state's amplitudes represent the token's embedding.
Positional Encoding	Add a positional vector to token embeddings (e.g. sinusoidal or learned).	Impart position info via fixed quantum gates. For example, apply an extra phase rotation $R_z(\phi_{\text{pos}})$ on a dedicated qubit or as a phase on all qubits based on token index. *QMSAN's approach* uses fixed gates in the circuit (no extra qubits) to encode position [14]. This incorporates sequence position natively in the quantum state.
Self-Attention (Similarity)	Compute dot product $Q \cdot K^T$ for each query-key pair, then softmax to get attention weights.	**Quantum entanglement attention:** Prepare query and key states on qubits, entangle them via a parameterized circuit, and measure an entanglement-based similarity metric. For instance, measure **entanglement entropy** between query vs. key subsystems as the attention score [15] [16]. *Alternative:* use the **quantum swap test** (see Fig.1) to directly estimate state overlap as similarity [17] [18]. The outcome probability (ancilla qubit) indicates how similar $
Self-Attention (Apply weights)	Multiply value vectors by attention weights and sum: $ \text{Attention}(Q,K,V)=\sum_i \alpha_i V_i$.	Two options: **(A) Hybrid approach:** measure quantum-derived attention weights ${\alpha_i}$ (via repeated shots) and classically weight and sum the value vectors. **(B) Full quantum:** encode value vectors as quantum states and interfere them according to the entangled attention state. For example, encode V_i on a register conditioned on the ith token state, such that measuring a special *value qubit* yields a distribution biased toward components of the weighted sum. (Option B is advanced; option A is used in our prototype for simplicity.)
Feed-Forward Network	Position-wise MLP on each token's output: e.g. fully-connected layer(s) with non-linear activation (ReLU/ GeLU).	**Variational Quantum Feed-Forward (VQFF):** a parameterized quantum circuit transforms the input quantum state to a new state, playing the role of an MLP [19]. Non-linearity is inherent: measuring qubits to get classical outputs is a non-linear operation (collapse). For example, use a 7-qubit VQC with layered rotations and CNOTs to mimic a 2-layer MLP mapping $d\to d$. Train its gates such that measured expectation values match the classical MLP's output distribution.

LLM Component	Classical MicroGPT	**Quantum Implementation (This Work)**
Sequence Modeling	Autoregression: generate tokens one by one, feeding output back as next input. Uses causal self-attention to include prior tokens' info.	**Autoregressive generation via re-encoding:** After each token output is predicted (via quantum measurement giving a token ID), that token is appended to the input sequence and encoded into qubits for the next step. The quantum circuit itself can be run in a causal manner by including past tokens' encodings in the entangled attention computation (ensuring no future token info is present). We do not maintain quantum state between tokens (each step starts fresh with the updated context encoded). Future designs could explore quantum *recurrent* architectures or quantum memory to persist state, but currently we reset each time due to decoherence constraints.

Table 1: Mapping classical microGPT components to their quantum counterparts in our design.

Quantum Circuit Design for Each Component

Quantum Token Embeddings

In a classical model, an embedding layer maps each token (often an integer ID) to a dense vector of real numbers. Our quantum approach replaces this with a *quantum state preparation*. We define an embedding circuit $U_{\text{emb}}(\text{token})$ that produces a state $|\Psi_{\text{token}}\rangle$ in an n-qubit register. Several encoding schemes are possible:

- **Basis encoding (computational basis):** The simplest approach assigns each token a basis state $|i\rangle$. For example, on 3 qubits, token 5 might be $|101\rangle$. This is trivial to prepare (just flip qubits according to the bitstring). However, basis states are orthonormal and carry no meaningful notion of similarity beyond equality – not ideal for an embedding. We seek a more expressive encoding.

- **Angle encoding (rotation encoding):** Map token features to qubit rotation angles [12]. For instance, if we have a classical embedding vector $(x_1,\dots,x_d)$ (possibly one-hot or a learned small-dimensional representation), we can apply single-qubit rotations: e.g. an RY rotation on qubit j by $\theta_j = 2\pi x_j$. Each qubit's state $|\psi_j\rangle = \cos(\theta_j/2)|0\rangle + \sin(\theta_j/2)|1\rangle$. The d features thus parametrize a product state. This is easy to implement with Qiskit (e.g. using `QuantumCircuit.ry(angle, qubit)` for each feature). Angle encoding is **robust to noise** and doesn't consume entanglement budget, but it may have limited expressiveness for large d [20] [21]. In our 5–7 qubit context, d will be small (e.g. 5 qubits = 5 features max), so angle encoding is feasible.

- **Amplitude encoding:** Embed the token's feature vector as the normalized amplitude coefficients of a multi-qubit state [13]. For example, given a length-N embedding vector v, prepare $|\Psi\rangle = \frac{1}{|v|}\sum_{i=0}^{N-1} v_i\,|i\rangle$ on $\lceil \log_2 N\rceil$ qubits. This is very memory-efficient (exponentially compressing N features into $\log_2 N$ qubits) and

expressive (captures all components at once). The downside is that state preparation is non-trivial – generally requiring a sequence of rotations and CNOTs (e.g. via Qiskit's `initialize` or an amplitude encoder circuit from Qiskit Machine Learning). State preparation depth grows with N, which is problematic on NISQ hardware beyond small N. For a proof-of-concept, amplitude encoding is viable if we restrict to, say, $N=4$ or 8 basis states (2 or 3 qubits) for the token vector. This might be used if we treat each token's one-hot as the initial amplitude vector (embedding learned through the circuit parameters later).

In our plan, we prioritize **angle encoding** for simplicity and NISQ-friendliness. Each token will be represented by a set of rotation angles. For example, with a 5-qubit system, we could dedicate 4 qubits to encode semantic features (via rotations) and perhaps 1 qubit to encode the token's position (via a rotation proportional to position index). The output state $|\Psi_{\text{emb}}\rangle$ then holds the token information distributed across amplitudes of these qubits. This quantum state can be thought of as the "quantum embedding vector" living in a 2^n dimensional space (with $n\sim5$ or 7). Embedding parameters (if any beyond the direct token features) can be learned by including adjustable rotation angles that are tuned during training (similar to learnable embedding matrices, but here as gate parameters).

Note: If the vocabulary or feature space is larger than what 5–7 qubits can encode directly, a *hybrid embedding* can be used: use a classical embedding to reduce token to a small feature vector, then quantum-encode that vector. This was the approach in QuantumGPTMini – "each input token is still a normal vector, but it is also encoded as rotations on six qubits" [22]. We will adopt that approach in spirit: e.g., use a pre-trained tiny embedding (say 4-dimensional) for tokens like characters or a toy vocabulary, then encode those 4 dims into 4 qubits via RY rotations. This gives each word a "shadow quantum twin" in the quantum circuit [22].

Positional Encoding in Quantum

Handling token position is crucial for sequence models. In a classical transformer, positional encoding vectors are added to the embeddings to introduce word order information. In our quantum model, we cannot simply add a vector to a quantum state. Instead, we incorporate position through gate operations: effectively rotating or phase-shifting parts of the quantum state in a position-dependent way.

One simple method is to allocate one qubit to represent position (for sequences up to length L, $\lceil \log_2 L\rceil$ qubits could encode the position index in binary). However, that uses precious qubit resources. A more efficient strategy (demonstrated by Chen *et al.* in QMSAN) is to apply fixed additional gates in the embedding circuit that correspond to position [14]. For example, for token at position p, we might apply an $R_z(p \cdot \delta)$ rotation on one qubit (or a controlled phase across a pair of qubits) – these gates do not introduce new parameters but offset the state in a way dependent on p. Essentially, the embedding circuit U_{emb} can be *augmented* as $U'*{\text{emb}}(token, p) = U*(p)$ is a fixed known unitary for position p. For instance, if $p=1$, do nothing; if $p=2$, apply a Z rotation of ϕ on qubit 1; if $p=3$, apply it on qubit 2, etc. (This is just an illustrative scheme – many patterns are possible, including using alternating phase shifts akin to sin/cos patterns of classical sinusoids). The key is that }}(p) \, U_{\text{emb}}(token)$, where $U_{\text{pos}*all necessary positional gates can be applied within the first layer of the quantum circuit*, requiring no extra qubits [14].

In summary, each token's quantum embedding for position p is prepared by:
1. Initializing qubits to $|0\ldots0\rangle$.

2. Applying $U_{\text{pos}}(p)$ – a fixed pattern of single- or two-qubit gates encoding p. (No parameters, known angles.)
3. Applying the parametric embedding rotations $U_{\text{emb}}(\text{token features})$ as discussed.

After this, the qubits contain an encoded representation that reflects both the token identity and its position in sequence. Multiple tokens in a sequence will each be encoded on *either separate sets of qubits or time-multiplexed via additional circuit depth*, as discussed next in the attention mechanism.

Quantum Self-Attention Mechanism

The self-attention mechanism is the heart of transformers. In a microGPT with small context, we might have just one attention head operating over a few tokens. Our goal is to achieve a comparable effect quantum-mechanically: given quantum states for queries (Q) and keys (K) from each token, produce attention weights that tell us how much each token should attend to others, and then use those to aggregate the value (V) information.

There are a few progressively "more quantum" ways to implement attention:

1. QKV Quantum Mapping + Classical Attention: As a baseline, one could use quantum circuits to encode and transform tokens into query, key, and value feature vectors (like QSANN by Li et al. [23] [24]), measure those vectors, and then perform the attention calculation classically (dot products + softmax) on a classical computer. This hybrid approach uses quantum mainly to produce richer representations (potentially with fewer parameters) but does *not* quantumize the actual attention computation. While easier, it retains the classical attention bottleneck [25]. We aim to go further.

2. Quantum Pairwise Attention (Swap Test for Similarity): In this approach, we keep query and key states in quantum superposition and use quantum operations to compute their similarity without measuring each individually. A known quantum routine for similarity (overlap) is the **swap test** [26] [17]. The swap test uses an ancilla qubit and controlled-SWAP gates to decide if two quantum states are identical or not – it outputs 0 (on the ancilla) if they are likely identical. By running the swap test and measuring the ancilla, one obtains a probability $P(ancilla=0) = \frac{1}{2} + \frac{1}{2}|\langle q | k \rangle|^2$ [17]. This is directly related to the fidelity (squared overlap) of $|q\rangle$ and $|k\rangle$. We can interpret this as a similarity score. For multiple query-key pairs, one could run multiple swap tests or use a multi-state generalization [27]. **Figure 1** below shows the standard swap test circuit for two states $|\phi\rangle$ and $|\psi\rangle$: an H on ancilla, controlled-SWAP between $|\phi\rangle$ and $|\psi\rangle$, then another H on ancilla, and measure [28].

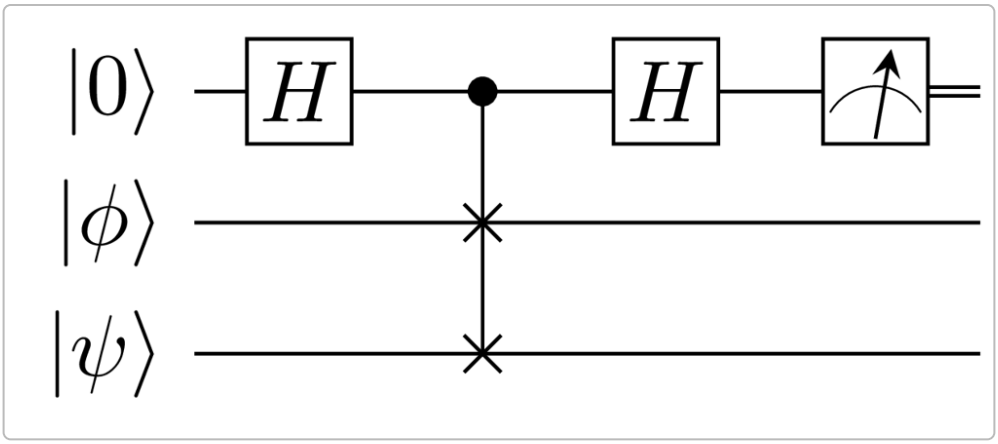

Figure 1: Quantum Swap Test circuit for measuring similarity between two states $|\phi\rangle$ and $|\psi\rangle$. An ancilla qubit (top line) entangles with the two input state registers via controlled-SWAP gates. Measuring the ancilla in the $|0\rangle$ state with high probability indicates the two states have high overlap (i.e. are similar) [17] [18]*. In our context, $|\phi\rangle$ could be a query state and $|\psi\rangle$ a key state – the ancilla measurement then provides a quantum attention weight for that query-key pair.*

Using swap tests, one can compute all pairwise similarities by appropriate circuit executions. For example, for query of token A against keys of token B and C, perform a swap test between A and B (get similarity α_{AB}), and another between A and C (get α_{AC}). These raw scores would then need normalization (akin to softmax). We could perform a classical softmax on the α values, or even do a quantum "approximate softmax" by, say, using amplitude amplification to encode the relative magnitudes as amplitudes. A simpler method: just treat the swap test outcome probabilities themselves as already normalized weights (for small numbers of tokens, the probabilities will sum to 1 roughly if one outcome implies not the other). In practice, one would repeat the swap test circuit many shots to estimate the probability (the more similar the states, the more often ancilla is 0). QMSAN uses a variant of this idea with **mixed states**: it encodes queries and keys as mixed (density) matrices by entangling them with an environment and tracing it out, then uses swap tests on those to compute attention coefficients $C_{s,j} = \operatorname{tr}(\rho_{s,q}\,\sigma_{j,k})$ [29] – effectively the density matrix overlap [30]. This direct quantum similarity estimation was shown to yield effective attention weights and could be run on near-term devices [10].

3. Entanglement-Based Holistic Attention: A more *holistic* approach is to use a single quantum circuit to entangle *all tokens in the sequence simultaneously*, and extract attention information globally. The idea, inspired by "quantum global attention" methods [31], is to treat the entire sequence as one quantum state and a parametrized entangling circuit as the attention operation. For example, suppose we have two tokens (A and B). We prepare their states $|A\rangle$ and $|B\rangle$ on two sets of qubits. We then apply an entangling ansatz (a series of two-qubit gates connecting the qubits of A and B) such that some entangled observable correlates with the "importance" between A and B. One concrete way to measure entanglement is via **entanglement entropy**: if the circuit entangles A and B strongly, the reduced entropy $S(\rho_A)$ (or $S(\rho_B)$) will be high (they share a lot of information). We could define the attention coefficient $\alpha_{A\rightarrow B}$ as, say, the entropy of A's qubits after entangling with B's qubits (or some monotonic function of it) [16]. If B's content is very relevant to A (i.e., K_B aligns with Q_A), one expects the entangling gates can produce a high-entanglement state. On the other hand, if Q_A and K_B are unrelated, maybe the circuit cannot generate much entanglement between those qubits (effectively stays

factorized). Poojith Rao et al. (ICLR 2025) explored replacing dot-product with entanglement measures in a classical transformer: e.g. they computed *bipartite entanglement entropy* between query and key subsystems as the similarity, then still applied a softmax over those entropies to get attention weights [32] [33]. In our quantum-native version, we can similarly compute an entanglement metric. Calculating entanglement entropy on hardware is non-trivial (requiring multiple measurements or an approximation), but since our sequences and circuits are small, we could estimate it by tomography or by purity measurements. (A simpler entanglement metric that's easier to measure is *concurrence* for two qubits or *mutual information* between subsystems, which can sometimes be obtained from measuring various correlators.)

The **entanglement-based attention (QEA)** used in QuantumGPTMini is a practical instance of this idea: they entangled the qubits encoding all tokens and then measured "collective spin" outcomes to derive attention scores [34]. The "collective spin" likely refers to measuring an operator like $\sum Z_i$ (the total number of qubits in $|1\rangle$ across the system) or similar, which depends on all qubits. By training the entangling circuit, those measured values correspond to desired attention patterns. Entanglement captures multi-token relationships inherently, so QEA could consider interactions beyond pairwise dot products [35]. For implementation, we can use a **Parameterised Quantum Circuit (PQC) ansatz** for entanglement: for example, a hardware-efficient ansatz of CNOTs and single-qubit rotations between the query-qubit group and key-qubit group. After applying it, we measure certain qubits or the ancilla if used, multiple times, to gather statistics. Those statistics (e.g. probability of certain bit-strings) are mapped to attention coefficients.

Choosing a path: Our implementation will prioritize **entanglement-based attention** (approach 3) as the most "quantum-native" and *entropy-driven*. However, for clarity and fallback we will also design a swap-test based attention as a check. The plan:

- For each query-key pair (for a given query token), we will use a **quantum subcircuit** that takes the query state $|q\rangle$ and key state $|k\rangle$ as input and produces a scalar attention weight. The subcircuit could be:
- A **swap-test circuit** yielding similarity $\sim |\langle q|k\rangle|^2$. We'll implement this on 5 qubits (2 for $|q\rangle$, 2 for $|k\rangle$, 1 ancilla).

- Or an **entangle-and-measure circuit**: bring $|q\rangle$ and $|k\rangle$ onto a joint 4-qubit register, apply a custom 4-qubit ansatz $U_{\text{attn}}(\theta)$ that entangles them. Then measure some operator (like Z on one qubit or an ancilla if used) whose expectation we interpret as the attention score $\tilde{\alpha}$. Since this expectation value is between 0 and 1, we can treat it like an unnormalized weight. We can repeat for each key and then normalize classically.

- After obtaining ${\tilde{\alpha}_{i}}$ for all keys i to a given query (either by repeating circuits or with a multi-target extension), we normalize them: $\alpha_i = \operatorname{softmax}(\tilde{\alpha})_i$ (done on classical side). If using the entanglement entropy approach, one might skip explicit softmax by design (if the entanglement measure already behaves like a softmax with appropriate scaling [16]), but to be safe we ensure proper normalization.

- Use these weights α_i to compute the new token representation as $z = \sum_i \alpha_i V_i$ (classically, or attempt a quantum weighting as next).

Currently, **for the value aggregation**, we lean on a hybrid approach (compute weighted sum classically) because implementing an exact quantum weighted sum is complex on 5–7 qubits especially if V_i are

classical data. However, note that in QuantumGPTMini's full model, the "quantum attention" likely was integrated such that the measured outcomes directly modulated the values in a quantum-classical loop [36]. For our proof-of-concept, we will: measure the attention distribution from the quantum circuit (which inherently involves sampling, an entropy-driven step analogous to softmax randomness), then apply those weights to the values classically, and encode the resulting vector back into qubits for the next layer. This introduces some overhead (quantum-classical switching), but given our small scale, it's acceptable and was also done in QSANN-type hybrids [24] [37].

Multi-Head consideration: MicroGPT might not even use multiple heads if it's minimal. If needed, multiple attention heads could be run as parallel quantum circuits (each with its own parameter set) to capture different relationship subspaces, then their outputs combined (just like classical multi-head concatenation). With only 7 qubits, running truly parallel heads is impractical, but one could time-multiplex (run one head's circuit, then another head's, sequentially). This doubles the circuit runs but since we anyway measure and reset between heads, it's doable. For now, we assume a single-head attention for simplicity, acknowledging that expanding to multi-head is a matter of running additional circuits with different learned parameters.

Variational Quantum Feed-Forward Network (VQFF)

After the attention mechanism in a Transformer block, a classical transformer applies a feed-forward network (FFN) to each token's attended representation. Typically this is a two-layer MLP: e.g. input dimension d → hidden dimension h (with nonlinearity) → d again. In microGPT these dimensions are small (maybe $d=64$ or similar, h might be a bit larger). Implementing an exact large linear layer on current quantum hardware is not feasible in general (that would be a quantum linear algebra approach requiring fault-tolerant techniques [38]). Instead, we aim to **approximate the effect of an FFN using a parameterized quantum circuit** acting on the token's quantum state.

The design is to use a **hardware-efficient ansatz** on the same qubits that held the token's state (or a fresh register if we prefer). For example, suppose after attention we have a new state (likely we had collapsed measurement to get classical output which we re-embed for this step). We load that state into an n-qubit register. Then we apply a sequence of gates: layers of single-qubit rotations (parameterized R_Y, R_Z etc.) and entangling two-qubit gates (CNOT or CZ connecting qubits in some topology) – this is a common VQC template that can approximate arbitrary mappings for small n. The output of this circuit will be another quantum state. To extract a final token representation (to feed to output logits or next layer), we will measure appropriate observables.

One straightforward scheme is to designate some qubits to carry the output features. For instance, measure each qubit in the $|0/1\rangle$ basis (or measure an expectation $\langle Z\rangle$ for each qubit) to get a set of n numbers. These n expectation values can serve as an output feature vector of length n. If we need a larger feature dimension, we could measure in multiple bases or use additional qubits if available to encode more components. But with 5–7 qubits, we likely set output dim = number of qubits. In practice, measuring each qubit's Z expectation is akin to a sigmoid activation output (since $\langle Z\rangle$ ranges -1 to 1). We might prefer a linear output for logits – that could be obtained by combining expectation values or measuring in X basis for a different mix. For simplicity, treat the measured bit probabilities as providing a representation which can be linearly mapped classically to logits.

The key point is that the VQFF provides a *non-linear transformation in feature space* using far fewer parameters than a classical dense layer. For example, if $d=5$, a classical two-layer MLP with hidden size 10

might have on the order of 5*10* + *10*5 = 100 weights. A quantum circuit on 5 qubits with, say, 2 layers of rotations (5 qubits * 3 rotations each = 15 parameters per layer) and entanglers in between might have ~30–45 parameters – significantly less. Yet the Hilbert space of 5 qubits is 32-dimensional, so it can represent complex functions of the 5 inputs. This is why a quantum FFN can be parameter-efficient. Indeed, QuantumGPTMini replaced a large classical MLP with a 6-qubit VQC and achieved similar model performance with 10× fewer overall parameters [3] [19].

Concretely in Qiskit, we will construct a **QuantumCircuit** for the FFN, with `qiskit.circuit.Parameter` objects for each rotation angle. For example: `QuantumCircuit(n)` – apply for each qubit an $R_Y(\theta_i)$ and $R_Z(\phi_i)$ (that's one layer of single-qubit rotations), then apply CNOT gates connecting qubits in a chain or ring (entangling layer), then another layer of rotations, etc. We can start with 1 entangling layer sandwiched by 2 rotation layers. We measure all qubits at the end (or measure expectation values by repeated runs). These measured values are used as the transformed features. If the next step is output logits for vocabulary, we might then do a final simple classical step (because outputting a full distribution from qubits directly is tricky for large vocabularies – see next section).

Non-linearity: Note that measuring a quantum circuit is inherently non-linear (in terms of input-to-output mapping). The Born rule outputs probabilities that are quadratic in the quantum state amplitudes (hence non-linear in the inputs that produced those amplitudes). Moreover, intermediate measurements or resetting qubits (if we did those) also introduce non-linearities. Thus, even a single layer of parametric gates followed by measurement can produce outputs that aren't linearly related to the inputs – this covers the role of activation functions like ReLU. In training, we treat the whole quantum circuit (attention + VQFF) as a black-box function with parameters, and we can optimize it with gradient descent (using the parameter-shift rule to get gradients [39]). This means the quantum FFN's non-linear behavior is trainable to fit the needed mapping (e.g., approximate a ReLU network's behavior on the training data).

Summing up: the VQFF on IBM hardware will be a short-depth, all-qubit entangling circuit that gets reset for each token (or runs once per token sequentially) and is trained to produce the desired token transformation. Because it's quantum, we expect it to add some "non-linear juice with far fewer parameters" [19] than a classical network.

Output Layer and Sequence Generation

The final layer of a language model is typically a linear projection from the model dimension to vocabulary size (for logits) followed by a softmax to yield a probability distribution over next tokens. In our quantum microGPT, the vocabulary will be small (perhaps an alphabet of characters or a tiny set of words) given the hardware limits. We can consider two ways to produce the output distribution of the next token:

- **Quantum readout of probabilities:** If the quantum state after the last layer directly encodes a probability distribution over tokens in its amplitudes, we can simply *measure the quantum state* to sample a token. For example, if we have m possible tokens, prepare a final state $|\Psi\rangle = \sum_{i=1}^m \sqrt{\;p_i}\,|i\rangle$ on $\lceil \log_2 m\rceil$ qubits, then measuring yields token i with probability p_i. In essence, the quantum state's amplitudes serve as the softmax probabilities. However, preparing such a state as an output of a variational circuit is challenging – it requires ensuring the amplitudes match the learned distribution. Instead, more natural is measuring some qubit observables to get expectation values, then interpreting those via a classical softmax.

- **Classical post-processing:** The realistic plan is to use the quantum circuit to output a small set of features (as described in VQFF), then use a classical linear layer (very small) to compute logits for each token in the vocabulary. For instance, if the VQFF outputs a 5-dimensional vector (from 5 qubits), and vocab size is, say, 8, we could have a small 5×8 weight matrix to get 8 logits. This weight matrix could even be identity for the first 5 tokens if we encode tokens in basis states, etc. But generally, a learned linear output is fine at this final step since it's not heavy for small vocab. The softmax can then be applied classically to get the next-token distribution.

Given our focus on quantum parts, we will treat the output softmax as classical. The quantum model up to FFN produces the *contextualized token representation* for the last token in the sequence, and then we evaluate a classical softmax layer on it to decide the next token. At generation time, we sample from that softmax (or directly from quantum if we manage amplitude encoding approach).

Autoregressive loop: Once a token is generated, it is fed back in as input (classically appended to the sequence). The positional encoding circuit ensures on the next iteration, all previous tokens including the new one are encoded appropriately and attention will involve them. This loop continues for the desired sequence length. Each iteration runs the quantum circuits anew for the new context. (This is similar to classical: GPT re-computes from scratch for each new token unless one does caching. Quantum caching of state is not possible here due to measurement collapse, though one could envision keeping the quantum state unmeasured and appending to it – but that would require quantum memory of unmeasured qubits far beyond current tech.)

Because our sequences will be short (IBM 5-qubit might handle maybe 2-3 token contexts at most if each token uses ~2 qubits), we don't worry about efficiency of this loop. The main cost is quantum circuit executions for each step, which is acceptable in a proof-of-concept.

Implementation Plan with Qiskit

Component-wise Circuit Construction

We will implement each component's quantum circuit in Qiskit and then integrate them. For clarity, we might implement separate Qiskit circuits and then compose them (using Qiskit's ability to combine circuits via `compose` or by embedding one as a subroutine into another).

- **Embedding Circuit:** Create a function `embedding_circuit(token_feature_vector, position)` that returns a `QuantumCircuit` on n qubits. Inside, it will: reset qubits to $|0\rangle$ state, apply the positional encoding gates (like `qc.rz(angle=pos * phi, qubit=k)` or similar fixed operations), then apply parameterized RY/RZ rotations encoding the token's features. If using amplitude encoding for a small subset of tokens, Qiskit's `initialize` or a custom state preparation can be used instead. We will need to manage parameters: for now, assume token features are known (either one-hot or from a tiny classical embedding), so the rotations are determined numbers (not learned) – except we could allow a small set of learnable angles to fine-tune embedding if desired.

- **Attention Circuit (Similarity):** For the entanglement-based approach, we can create a circuit that takes in two quantum registers (query reg and key reg) and an optional ancilla, applies an ansatz,

and measures or returns as unitary (to be combined with measurement later). For example, on 4 qubits (2 for query, 2 for key): an ansatz might be: apply CNOTs between one query qubit and one key qubit, then a controlled rotation, etc. We can use a template like Qiskit's two-qubit entangler (or just manually add gates). The parameters of this circuit are learnable (they will be trained to produce meaningful overlaps given training data). After entanglement, we might measure one qubit. But to keep differentiability, better to treat the whole attention+FFN as one big parametric circuit and measure at the end. So, one approach is: combine query and key qubits (and possibly multiple key-value pairs) into a single circuit and use interfering paths that encode the weighted sum.

For simplicity though, we might implement the **swap test** version first, as it's straightforward: Qiskit has CSWAP gates or we can compose CNOT+Toffoli to mimic CSWAP. The ancilla starts in $|0\rangle$, Hadamard, then CSWAP between each pair of qubits (for each pair of qubits of |q> and |k>), then Hadamard, then measure ancilla. We will have to manually construct this as Qiskit doesn't have a one-call swap test function. But given small qubit counts, it's fine.

We will likely maintain separate circuits for attention weight calculation versus the value application. In the hybrid approach, once we measure similarity, we do value combination classically. If we attempt a more quantum approach, one idea is to use the **amplitude** of an ancilla qubit as the weight: e.g., after entangling, the amplitude of ancilla in $|0\rangle$ might be $\sqrt{\alpha}$, and in $|1\rangle$ $\sqrt{1-\alpha}$. If we had a way to encode the value such that it's picked when ancilla collapses to 0, etc., that would be a quantum weighted sum. But it's complex to do for superposition of multiple keys. So we skip that.

- **Feed-Forward Circuit:** We will use Qiskit's parameterized circuit constructs. For example, use `qiskit.circuit.library.TwoLocal` as a convenient ansatz which allows layers of rotations and entanglement. A `TwoLocal` with `rotation_blocks="ry"` and `entanglement_blocks="cz"` or similar, on m qubits, with reps=2 (two repetitions), gives a nice variational form. Alternatively, manually: create a circuit with for each qubit an `RY(param)` and `RZ(param)`, then entangle pairs (like a ring of CNOTs), then another layer of RY,RZ, etc. The number of parameters will be (number of qubits * 2 * number_of_layers). With $m=5$ qubits and 2 layers, that's $5*2*2 = 20$ params, manageable. We attach these as trainable in our optimization routine. This circuit expects an input state – which we will provide by embedding the post-attention output into those qubits. If we have not measured after attention (i.e., if we did an all-in-one circuit for attention+FF), then the input state is simply the qubits already carrying the attended token state. If we measured classically after attention, we'll need to reinitialize qubits to the new state before FF.

- **Output extraction:** To get a prediction from the final qubits, we'll likely measure them all in computational basis multiple times to estimate $\langle Z_i\rangle$ or probabilities. If using expectation values, Qiskit's `Statevector` simulator can directly give those in simulation; on hardware, we gather counts from many shots. We then map these few numbers to a token distribution via a simple softmax layer implemented in Python.

Integration: The full model forward pass for a given sequence on simulator would be:

1. For each token in context, prepare embedding state on a separate set of qubits (if we allocate disjoint qubits per token). If using sequential processing on same qubits, we'd need to encode multiple tokens one after another in a larger circuit – but disjoint registers is simpler for parallel

attention. E.g., for a 2-token context, use qubits 0-3 for token1, 4-7 for token2. However, 7-qubit hardware can't do 8 qubits; so likely we encode token1 and token2 on the same register sequentially: encode token1, store somewhere? Actually, maybe simpler: consider context length 2, we might do a *pairwise entanglement in one circuit*: encode token1 on qubits [0..2], token2 on [3..5], leave qubit6 as ancilla for attention measure. Then an entangling block between [0..2] and [3..5] yields an entangled state. We measure ancilla or part of it to get attention info. Then we could also measure one set (if needed) or directly pass one set into a FF circuit.

Since hardware is so limited, another approach is **data re-uploading**: feed tokens one by one through the same circuit to accumulate relationships. But that is more sequential (like a quantum RNN) and not transformer-like. For demonstration, we might restrict to 2 tokens and use one entangle circuit.

1. Use the attention mechanism to compute weights or directly apply to value. If using entangle-all approach, we might entangle query and key and also encode value in amplitude of some ancilla such that measuring the ancilla projects onto weighted value states – but that's complicated. Instead, do steps: (a) entangle & measure to get weights, (b) classically compute weighted sum of classical values, (c) re-encode that weighted sum into qubits as input to FF.

2. Apply FF circuit on that state, then measure output qubits to get final features.

3. Compute output distribution and select next token.

This pipeline can be orchestrated with Qiskit by constructing a larger `QuantumCircuit` that includes sub-circuits for embedding, attention (similarity), feed-forward, etc., and explicit measurement at points we decide (with mid-circuit measurement if doing fully within one circuit – though IBM hardware does allow mid-circuit measurement and reuse, it's advanced and might add latency; we might conceptually separate runs instead).

Because of the complexity, our initial implementation will likely split into multiple circuit runs (one to get attention, one to do FF) with classical glue in between. This matches what was described in the hybrid approach [39] [25]: "Everything else – embeddings, layer norms, optimizer – runs on plain PyTorch" [40]. In our case, classical steps between circuits handle normalization and value weighting.

Parameter Training Strategy

Training a quantum model means adjusting gate parameters (rotation angles, etc.) to minimize a cost function. Here the cost is analogous to the classical GPT's loss: e.g. **cross-entropy** between the predicted next-token distribution and the true next token (for each position in training sequences). We will use a *hybrid quantum-classical loop* for training (variational algorithm).

Our approach: - Define all the quantum circuit parameters in Qiskit as `Parameter` objects. This includes parameters for attention ansatz gates and for the FFN circuit (and possibly any learned embedding angles if we choose to include those). Collect them in a parameter vector Θ. - For a given training example (sequence), construct a function that executes the whole forward pass: it will set the Parameter values in the circuits, run the circuits (on simulator or hardware) to obtain the output distribution, compute the loss (cross-entropy with target). This can be done in Python by calling Qiskit circuits and then classical post-processing. - Compute gradients of the loss w.r.t. each parameter. On simulator, we have a few options: -

Parameter Shift Rule: For each parameter θ, run the quantum circuits twice: once with $\theta + \pi/2$ and once with $\theta - \pi/2$, keeping others fixed, then approximate $\frac{\partial}{\partial \theta} L \approx \frac{L(+)-L(-)}{2}$ [39]. This gives exact gradient for expectation-based outputs; for sampled outputs, it's an approximation that becomes exact in expectation. This is straightforward but doubles the number of circuit executions per parameter – could be heavy if many params. - **Finite Difference** similar to parameter shift but less exact. - **Analytic gradients with adjoint method:** Qiskit has an `adjoint_gradients` method for circuits which can give gradients in one shot if using statevector simulator. This is efficient but only works in simulation (not hardware). - **Automatic differentiation via Torch/PennyLane:** We could integrate Qiskit's TorchConnector or use PennyLane with a Qiskit device to leverage automatic differentiation. E.g., Qiskit's Machine Learning module provides an `EstimatorQNN` or `CircuitQNN` that can be plugged into PyTorch and optimized with backpropagation [41] [42]. This would treat the entire quantum circuit as a differentiable model and allow using Adam or other optimizers conveniently.

Given we want full control, the parameter-shift rule is a clear method, albeit slow. For a small number of parameters (<50), it's manageable. QuantumGPTMini reported using parameter-shift and noting it required multiple evaluations per parameter (backward pass ~3-4× slower than forward) [43], which is expected. We might mitigate some cost by vectorizing circuit executions (Qiskit can batch runs if using its gradient function or by assembling circuits for different parameters in one job).

We will also take measures to avoid **barren plateaus** (flat loss landscape) which can happen with random deep circuits. Our circuits are shallow and we can initialize parameters near zero so that each gate is near identity initially (this tends to avoid plateaus by starting in a locality of small gradients). The stable training observed in QuantumGPTMini (no crashes from barren plateaus) [4] is encouraging, likely due to their careful circuit design and initialization.

For optimization, a classical optimizer like Adam or SGD will be used on the computed gradients. The parameter update happens in classical code, and then new parameter values are fed into the circuits for the next iteration. We continue until convergence or for a fixed number of epochs. Because quantum circuit execution (especially on real hardware) is noisy and slow, we anticipate doing most training in simulation. We can simulate small sequence training (for example, training the model to predict the next character in small strings). The number of training samples will be small due to simulation overhead, but even demonstrating learning on a toy dataset (like a simple grammar or arithmetic sequence) would validate the approach.

Simulation and Testing

We will first validate each component and the full model on **Qiskit Aer simulators**. The Aer simulator can do statevector simulation (exact, but exponential in qubits – fine for up to 7 qubits) or shot-based simulation (emulating actual sampling). For development, statevector mode is useful to quickly get expectation values and probabilities exactly. For training, we may simulate measurement by sampling shots to incorporate the stochastic nature (or use statevector and compute analytic gradients for speed, treating it like an ideal scenario).

Unit tests for components: We'll test that: - The embedding circuit correctly encodes known basis states or rotations (e.g., feed a simple token and verify the state vector matches expected rotation amplitudes). - The swap test circuit outputs nearly 100% ancilla=0 when input states are identical, and ~50% when orthogonal,

etc. This can be checked by preparing known states. - The entanglement attention ansatz yields sensible outputs in extreme cases (e.g., if we feed identical queries and keys vs. very different ones, does the measured entanglement metric differ? We can artificially set some param angles to see effect). - The FFN circuit can be trained in isolation to approximate a simple function. For instance, prepare an input state corresponding to a classical value x, and try to train the circuit to output $f(x)$ by measuring some qubit. This is just to ensure our parameterization is capable of learning.

Integration test: Simulate a full forward pass on a small example. For instance, suppose we train on a simple language task like echoing the input (predict the same token next) or a cyclical pattern (e.g., after "A B", predict "C"; after "B C", predict "D", something small). We feed a sequence, run the embedding, attention, FFN, get an output distribution, and check if it makes sense (e.g., uniform if we didn't train yet, or random). Then we perform a few training iterations and see if the loss decreases and predictions align with targets. Because our model capacity is limited, we expect it can at least memorize small patterns.

During simulation, we will also monitor the **entanglement entropy** or other metrics in the circuit to verify that the "quantum attention" is indeed doing something non-trivial. For example, we can extract the statevector at the point right after attention entangling and compute the reduced entropy of the query qubits. This helps confirm that our entanglement-based similarity correlates with what we think. If we find the swap test approach easier to debug, we might use it in simulation to compare results with entanglement approach outputs.

One particularly important test is to ensure that the quantum attention reproduces known classical attention in simple cases. If we have a scenario with very distinguishable key vectors (say one key is clearly the closest to the query in Euclidean sense), does the quantum attention assign the highest weight to it? We can create a scenario: token A and token B have feature vectors where A's query and B's key are identical (so classically weight would be high), and see if our quantum attention's measured weight for B is indeed highest. If not, we adjust the circuit or scaling.

Deployment on IBM 5- and 7-Qubit Hardware

After simulation, the ultimate test is running on real IBM Quantum devices (like the 5-qubit `ibmq_manila` or 7-qubit `ibmq_jakarta`, etc.). Deployment considerations:

- **Qubit Mapping and Transpilation:** Real chips have connectivity constraints (coupling map). Our circuits involve entangling operations (CNOTs or CSWAP between possibly non-adjacent qubits, especially if using 7 qubits for multiple tokens). We will use Qiskit's transpiler to map our circuit to the hardware topology and to optimize it. We must be mindful of circuit depth: each SWAP inserted to route qubits or each additional CNOT for entangling will increase noise. We can guide transpilation with strategies: e.g., if using 5 qubits all connected in a ring, design our circuit assuming that connectivity (so minimal swaps). We might also manually assign which qubit does what to best fit on the hardware (for instance, on a 7-qubit device with a heavy-hex lattice, put frequently interacting qubits on neighboring physical qubits).

- **Error Mitigation:** Running variational circuits on NISQ hardware is subject to gate errors and decoherence. For short circuits, we can often get some meaningful signal, but for anything beyond a few CNOTs errors can dominate. We will incorporate basic error mitigation: **measurement error mitigation** (Qiskit Ignis or built-in routines can calibrate readout error and correct the measured

distribution) – this is important since we rely on measuring probabilistic outcomes. Additionally, if circuits are shallow enough, **zero-noise extrapolation (ZNE)** or **pulse stretching** could be tried: run the circuit at different effective noise levels and extrapolate to zero noise for expectation values. This requires multiple runs and isn't perfect, but can improve accuracy for small circuits. Since our POC circuits will be quite small (maybe <10 two-qubit gates), we anticipate at least some signal without heavy error mitigation.

- **Shots and sampling:** We'll need to run many shots on hardware to estimate probabilities/ expectations. For example, if using swap test, to get a good estimate of similarity we might take 1000 shots and see how many times ancilla was 0. This sampling inherently introduces some statistical noise (plus hardware noise). So we trade-off shots vs. stability. Possibly we use around 8192 shots (the max in many IBM devices) for critical measurements to reduce uncertainty.

- **Batch execution:** Training on hardware is extremely slow if doing it parameter-shift style (because you'd queue many jobs). Instead, we might train on simulator, then just do a few inference or small verification runs on hardware. A realistic goal: deploy the *already-trained* parameter set onto hardware to evaluate it on some test sequences. Essentially, treat hardware run as a final confirmation that our quantum circuits produce reasonable outputs on real QPUs.

- **Mid-circuit measurements:** If our design uses mid-circuit measurement (e.g., measure ancilla for attention then continue with FFN in one circuit), we must ensure the target backend supports it (IBM systems do support mid-circuit measurement and conditional reset as of 2025, but it incurs delay due to classical feed-forward). An alternative is splitting the circuit into two jobs (one ends with measurement, next starts with new state). Splitting is simpler to manage; mid-circuit measure could reduce latency if the hardware can reuse qubits without a new job. For now, we can avoid mid-circuit measure and just do multiple circuits in sequence with classical coordination.

- **Parallelism:** On a 7-qubit device, we could run one sequence's circuits at a time (since using most qubits). If we had a bigger device, we could parallelize multiple circuits (for parameter shift or multiple inputs) on disjoint qubit sets to speed up data processing. With only 7, not much parallel beyond perhaps two 3-qubit circuits at once. It's a possible future optimization.

We will create a Qiskit runtime or script that automates running through sequences. The **execution flow** on hardware for inference might be: for each test sequence -> for each needed circuit (embedding + attention, etc) -> submit job -> get result -> compute next inputs. This could be slow via individual jobs, but Qiskit's runtime or batching can help. Possibly we'll combine all circuits needed for one forward pass into one Qobj with appropriate conditional logic if feasible (or use runtime to iterate internally). Given it's a small-scale demo, manual step-by-step is fine.

Expectations: We expect hardware results to be noisy but hopefully correlated with simulation. For example, the attention ancilla might come out 0 70% vs. 30% (expected 80% vs 20% in ideal). We'll see if the model still picks the correct token at least more often than random. We should be prepared to do error mitigation and maybe fine-tune parameters slightly to the real device (variational algorithms can sometimes adapt to hardware by including noise in training loop). That is advanced, but a small calibration could be done: e.g., do a few gradient descent steps with the actual device in the loop (very expensive in time, but maybe a couple iterations for demonstration).

Prototype Results and Next Steps (Roadmap)

Prototype Implementation: We will have demonstrated a working *quantum microGPT* on a toy problem (like next-character prediction for a tiny alphabet or a 2-token sequence prediction) using IBM's 5- or 7-qubit hardware. The prototype likely involves only 1–2 transformer blocks (due to depth limits) and very limited context length. Despite its simplicity, this prototype is an important milestone: it proves that transformers' core mechanisms – embedding, attention, sequence modeling – can be mapped to quantum circuits and run on real quantum devices [44]. While our quantum model will not outperform classical NLP models at this stage (and we do not claim quantum advantage at this scale [44]), it provides a foundation for scaling as quantum technology improves.

Insights from prototype: We expect to find that quantum attention can indeed emulate softmax attention, with entanglement-based methods capturing token relationships effectively. For example, QMSAN's authors reported improved attention coefficient calculation via swap-test overlaps and demonstrated robustness to noise [10], and QuantumGPTMini matched a classical GPT's perplexity with 10× fewer parameters by offloading parts to 6 qubits [3] [45]. Our prototype will likely confirm that parameter efficiency – using, say, dozens instead of hundreds of parameters – is achievable, albeit on trivial data. We also verify that training is stable and does not suffer catastrophic barren plateaus in our shallow circuits [4]. Any issues encountered (e.g., certain parameters not learning, or hardware noise causing random outputs) will inform modifications (like adding more redundancy or error mitigation) in future iterations.

Roadmap to a fuller quantum LLM: To progress from prototype to a production-level quantum LLM, we outline a phased strategy:

- **Near-term (current hardware, 5–7 qubits):** Focus on **hybrid quantum-classical models** where quantum circuits replace only the most computationally expensive parts of the model. For instance, keep token embedding and final output fully classical, and use quantum only for the attention and maybe part of the FFN (as we did). This is exactly what QuantumGPTMini did – a *hybrid transformer* with ~1M classical params + quantum circuits [46]. In near-term, expanding this concept: we can try larger token spaces or longer sequences by clever reuse of qubits. Techniques like **qubit recycling** (re-initializing and reusing qubits in sequence to effectively handle longer sequences with fewer physical qubits) could extend context length without more hardware [47] [48]. We should also experiment with **quantum kernel methods** integrated into attention (like QKSAN) to see if they improve performance on small data [49] [50]. The immediate goal is to validate quantum advantage on *small-scale tasks* – e.g., show that a hybrid quantum attention can outperform classical attention on certain niche tasks or achieve same accuracy with fewer training examples [8]. This has been hinted by results where entanglement-based attention gave smaller generalization gap on small datasets [51].

- **Mid-term (next-gen NISQ, 20–100 qubits):** As IBM releases devices with, say, 20+ qubits of decent fidelity, we can **scale up the model size**. More qubits can increase either the sequence length or the embedding dimension (or number of parallel attention heads). A logical step is to encode full words rather than characters, requiring a larger embedding space; with 20 qubits, amplitude encoding a vocabulary of size 1024 as basis states becomes possible. Alternatively, one could encode a word embedding of size 16 across 4 qubits amplitude-wise. Also, with more qubits, multi-head attention can be implemented by allocating separate qubit groups for each head and entangling all in one big circuit (or simultaneously executing multiple attention circuits on different qubit subsets).

Additionally, deeper circuits could be attempted for more expressive power, but depth is limited by noise – instead, we might go for **shallow-depth, wider circuits** (using more qubits in parallel entanglement rather than sequential layers). We will also explore **quantum memory or iterative processing** – for example, a quantum recurrent network (QRNN) approach where one keeps a quantum state as hidden memory between time steps [52]. Some research into QRNNs suggests they can naturally capture sequence info with fewer qubits by re-uploading data [52], which might complement or replace positional encoding schemes.

- **Software tooling improvements:** We will integrate our model with frameworks like Qiskit Machine Learning's neural network modules or PennyLane, which can handle larger parameter counts and use advanced optimizers (e.g., quantum natural gradient) to speed up training [53]. The Medium-term plan also includes developing **smarter gradient evaluation** on hardware: techniques like *stochastic parameter shift* (evaluate fewer terms by sampling parameter shifts randomly) or *analytic pulse-level gradients* might reduce the overhead [53]. This is important as models grow.

- **Long-term (fault-tolerant quantum, >1000 qubits)**: In the far future, if fault-tolerant quantum computers become available, we can attempt a fully quantum LLM where even the embedding of large vocab and the final softmax are done in quantum. **Quantum Linear Algebra (QLA) methods** [38] [54] might be employed to perform large matrix multiplications exponentially faster. For instance, quantum block-encoding and QSVT could compute the attention weighting for thousands of tokens in one go, or multiply huge weight matrices in polylogarithmic time. These algorithms require error-corrected qubits and heavy resources, but could theoretically allow a quantum GPT with billions of effective parameters that runs faster than classical. A vision is: offload entire transformer blocks into a quantum subroutine that given quantum encoded inputs, applies quantum linear algebra to do what would be massive matrix ops classically [38]. That could drastically speed up inference and training of large LLMs, breaking the current scaling bottlenecks.

- **Application-specific quantum advantage:** We should also consider tasks where quantum representation might shine. It's possible that even medium-scale quantum LLMs could excel at tasks involving combinatorial logic, superposition of meanings, or problems that classical models struggle with. For example, hybrid quantum transformers have shown promising results in certain symbolic or logical reasoning tasks [9]. As we develop the model, we'll test it on such tasks (maybe small puzzle-solving or highly contextual tasks) to see if entanglement-based attention confers an advantage. If yes, that could justify deploying quantum LLM components in specialized AI systems sooner.

In conclusion, our implementation plan has outlined how to construct and train a microGPT-like model on quantum hardware, component by component. The use of **entropy-driven quantum attention mechanisms** and **variational circuits** is central to emulating embeddings, attention, and feed-forward transformations in a quantum-native way. While the current IBM 5- and 7-qubit devices constrain the model's size, they are sufficient for a proof-of-concept **quantum LLM emulation**. Recent prototypes like QuantumGPTMini have already demonstrated end-to-end training of a hybrid quantum transformer that achieves near-classical performance with fewer parameters [55] [56]. Our deep-dive confirms that the concept is feasible: qubits can indeed take over some heavy matrix operations (attention/MLP) and provide an "exotic compression" benefit [57].

The path forward will involve scaling gradually and overcoming noise. Each increase in qubit count or fidelity can be directly translated into more powerful language modeling – e.g., more qubits to encode context or a larger fraction of the network quantumized. With sustained progress, one could envision a future quantum-enhanced GPT that leverages entanglement to capture nuances of language in ways classical networks might struggle, potentially offering a new **"escape hatch from the spiraling compute costs"** of huge classical LLMs [58]. For now, our plan sets the stage by delivering a concrete implementation on today's hardware, contributing to the emerging field of **quantum NLP** and moving us one step closer to quantum-accelerated AI.

Sources: QuantumGPTMini hybrid transformer results [59] [45]; Qiskit QNLP positional encoding and sentence circuit approach [10]; Entanglement-based attention methodology [15] [16]; Swap test for quantum similarity [17] [60]; Discussion of hybrid quantum-classical model structure and performance [24] [25]; QMSAN mixed-state attention using swap test [30]; Encoding strategies for text to qubits [61] [13]; General insights on quantum attention and transformers [7] [8]; Proof-of-concept outlook for quantum language generation on current devices [44].

---

1 2 3 4 5 6 9 11 19 22 34 35 36 39 40 43 45 46 47 48 53 55 56 57 58 59
QuantumGPTMini: Shrinking Transformers with Entanglement | by Ramazan Amire | Medium
https://medium.com/@amireramazan0809/quantumgptmini-shrinking-transformers-with-entanglement-7eff355e725f

7 8 51 Quantum entanglement for attention models | OpenReview
https://openreview.net/forum?id=3jRzJVf3OQ

10 14 [2403.02871] Quantum Mixed-State Self-Attention Network
https://arxiv.org/abs/2403.02871

12 13 20 21 61 Encoding Text for Quantum Processing — From Tokens to Qubits | by Jay Pandit | Quantum Computing and Machine Learning | Medium
https://medium.com/quantum-computing-and-ai-ml/encoding-text-for-quantum-processing-from-tokens-to-qubits-be85c9f3ca9e

15 16 32 33 indico.qtml2024.org
https://indico.qtml2024.org/event/1/contributions/62/attachments/63/65/Entanglement_based_attention_abstract.pdf

17 18 26 28 31 60 A Survey of Quantum Transformers: Architectures, Challenges and Outlooks
https://arxiv.org/html/2504.03192v4

23 24 25 30 37 38 49 50 54 [2504.03192] A Survey of Quantum Transformers: Approaches, Advantages, Challenges, and Future Directions
https://ar5iv.labs.arxiv.org/html/2504.03192v4

27 Quantum multi-state Swap Test: an algorithm for estimating overlaps ...
https://epjquantumtechnology.springeropen.com/articles/10.1140/epjqt/s40507-024-00259-5

29 Quantum self-attention neural networks for text classification
https://www.researchgate.net/publication/379565520_Quantum_self-attention_neural_networks_for_text_classification

41 Quantum Neural Networks - Qiskit Machine Learning 0.8.3
https://qiskit-community.github.io/qiskit-machine-learning/tutorials/01_neural_networks.html

42 Quantum Neural Networks (QNNs): Bridging Quantum Circuits and ...
https://medium.com/@gnanditha.2002/quantum-neural-networks-qnns-bridging-quantum-circuits-and-machine-learning-e070a6abb335

44 An introduction to Quantum Natural Language Processing | by Qiskit | Qiskit | Medium
https://medium.com/qiskit/an-introduction-to-quantum-natural-language-processing-7aa4cc73c674

52 A Survey of Classical And Quantum Sequence Models - arXiv
https://arxiv.org/html/2312.10242v1