Quantum-Inspired Cognitive Architectures for High-Frequency Trading on Polygon: A Deep Technical Analysis of DeepSeek-R1 and Llama 3.2 Integration

1. Introduction: The Latency-Intelligence Frontier in Decentralized Finance

The evolutionary trajectory of algorithmic trading within decentralized finance (DeFi) ecosystems is currently witnessing a bifurcation between raw execution speed and cognitive adaptability. As liquidity consolidates on high-throughput, Ethereum-compatible Layer 2 scaling solutions like the Polygon network, the market microstructure has transitioned into an adversarial "Dark Forest." In this environment, the profitability of a trading agent is determined not merely by the theoretical alpha of its strategy but by its position in the block-building supply chain and its latency relative to competitive Maximal Extractable Value (MEV) searchers. Traditional monolithic machine learning models, which rely on cloud-based inference and static weight updates, face an existential crisis in this domain. The network round-trip latency—often exceeding 200 milliseconds for external API calls—renders cloud-dependent architectures incapable of competing for atomic arbitrage opportunities that exist for merely the duration of a single block production cycle, approximately two seconds on Polygon.[1]

Consequently, the frontier of quantitative finance is shifting toward "Quantum-Inspired" (QI) computational frameworks. These paradigms leverage the mathematical principles of quantum mechanics—specifically superposition, entanglement, and tunneling—implemented on classical parallel hardware such as GPUs and FPGAs. The objective is to achieve the optimization speeds theoretically promised by quantum annealing without the constraints of current Noisy Intermediate-Scale Quantum (NISQ) hardware. By reframing combinatorial optimization problems (such as multi-leg arbitrage routing) into Hamiltonian energy minimization tasks, developers can utilize algorithms like the Simulated Bifurcation Machine (SBM) to achieve microsecond-scale decision velocities.[1]

This report provides an exhaustive technical analysis of the viability of Meta's Llama 3.2 (3B) as the cognitive core of such a quantum-inspired agent. The analysis contrasts a standalone Llama 3.2 deployment against a hybrid "Two-Brain" architecture that integrates the deep reasoning capabilities of DeepSeek-R1. The central hypothesis investigated is whether the instruction-tuned efficiency of Llama 3.2, when subjected to aggressive "quantum" hyperparameter tuning, can approximate the reasoning depth required to navigate the stochastic complexity of the Polygon mempool, or if the distinct cognitive profile of DeepSeek-R1 is a prerequisite for regime identification. Furthermore, this document supplies precise, Pythonic implementations of the Corner Classification (CC4) network, Variable-Time Preselection Algorithm (VTPA), and SBM, specifically catered to the function-calling context of a local Llama 3.2 instance.

2. Architectural Analysis: The "Two-Brain" System vs. Standalone Llama 3.2

The design of a trading agent for Polygon must reconcile two opposing forces: the need for deep, reflective reasoning to identify non-linear market regimes (Strategic Alpha) and the absolute requirement for sub-millisecond execution to capture fleeting liquidity (Tactical Alpha). This dichotomy forms the basis of the comparative analysis between a standalone Llama 3.2 architecture and a hybrid system utilizing DeepSeek-R1.

2.1. The Cognitive Profile of DeepSeek-R1: Strategic Reasoning

DeepSeek-R1 represents a significant departure from standard instruction-tuned Large Language Models (LLMs). Its architecture, utilizing a Mixture-of-Experts (MoE) framework with 671 billion total parameters (but only ~37 billion active per token), is optimized for complex reasoning tasks through a reinforcement learning (RL) pipeline that incentivizes Chain-of-Thought (CoT) generation.[2] This model does not merely predict the next token; it generates extensive internal reasoning traces, allowing it to "think," self-correct, and verify logic before committing to an output.

In the context of high-frequency trading, DeepSeek-R1 excels at **Regime Identification**. Financial markets are not stationary; they shift between regimes of mean reversion, momentum, and chaotic volatility. A simple model might interpret a sudden price drop as a buying opportunity (mean reversion), while a reasoning model like DeepSeek-R1 can analyze the broader context—cross-chain bridge flows, stablecoin de-pegging events, or governance proposal outcomes—to correctly classify the drop as a fundamental "breakdown" requiring a short position. This capability to hold conflicting signals in a dialectical tension essentially simulates the quantum state of superposition ($|\psi\rangle = \alpha|\text{Bull}\rangle + \beta|\text{Bear}\rangle$) until a logical observation collapses the state.[3]

However, this cognitive depth incurs a massive latency penalty. The generation of reasoning tokens significantly increases Time-To-First-Token (TTFT) and total generation time. On consumer hardware, even distilled versions of DeepSeek-R1 can exhibit latencies ranging from 500ms to several seconds depending on the complexity of the "thought process." In the Polygon environment, where block times are ~2 seconds and the race for arbitrage is won in the first 100 milliseconds of the block auction, this latency is prohibitive for direct execution.[1]

DeepSeek-R1 is, therefore, a "Strategist" rather than a "Trader."

2.2. The Kinetic Profile of Llama 3.2 (3B): Tactical Execution

In contrast, Llama 3.2 3B is engineered for the "Edge AI" paradigm. It is a dense, highly distilled model designed to run efficiently on local devices with constrained VRAM. Its primary advantage is **Inference Velocity**. On high-end consumer hardware like an NVIDIA RTX 4090, Llama 3.2 3B achieves extremely low TTFT, often under 30ms, and high throughput exceeding 150 tokens per second.[4]

This speed renders Llama 3.2 ideal for the "Execution Leg" of the trade. Once a strategy is defined, the agent must continuously scan the mempool, parse transaction input data, and identify specific trigger conditions (e.g., a large swap in a Uniswap V3 pool). Llama 3.2's ability to strictly follow JSON schemas and invoke external tools (function calling) makes it a highly reliable "router" for algorithmic backends.[5] It can ingest a natural language description of a market event and deterministically map it to the precise arguments required by a solver like the Simulated Bifurcation Machine.

However, Llama 3.2's "reasoning" capabilities are inherently limited by its parameter count. It struggles with zero-shot resolution of highly ambiguous scenarios. When faced with contradictory data (e.g., bullish on-chain volume vs. bearish social sentiment), smaller models tend to hallucinate or revert to the most probable training distribution, failing to capture the "tail risks" where maximum alpha is often found.[6] It lacks the internal "workspace" to perform the multi-step verification that DeepSeek-R1 performs natively.

2.3. The Hybrid "Two-Brain" Recommendation: The KITE Protocol

Given the distinct profiles of these models, the **Hybrid "Two-Brain" System** is rigorously recommended as the superior architecture for a production-grade Quantum Trading Agent. Relying solely on Llama 3.2 requires compromising reasoning depth for speed, while relying solely on DeepSeek-R1 compromises execution speed for depth. The Hybrid approach leverages **Heterogeneous Computing**, assigning tasks to the processor (or model) best suited for them.

We propose implementing this architecture via the **KITE (Kinetic Inference & Theoretical Engine)** Protocol. This protocol establishes an asynchronous communication bridge between the two models, allowing them to operate on different time loops.

Brain 1: The Theoretical Engine (DeepSeek-R1)

- **Role:** Strategic Oversight & Regime Detection.
- **Operational Loop:** Slow Loop (e.g., every 10-20 blocks or upon major volatility events).
- **Input Data:** Macro-market data, cross-chain liquidity metrics, news sentiment analysis, and historical performance logs.
- **Function:** DeepSeek-R1 does not issue individual trade orders. Instead, it analyzes the

global market state to determine the "Physics of the Market." It outputs a **"Quantum State Vector"**—a set of hyperparameters that tune the behavior of the second brain. These parameters might include risk tolerance (α), volatility expectation (σ), the "Radius of Generalization" (r) for the CC4 network, and the specific asset pairs to focus on (Entanglement map).

- **Output:** A JSON configuration file updating the execution parameters.

Brain 2: The Kinetic Engine (Llama 3.2 3B)

- **Role:** Tactical Execution & Atomic Arbitrage.
- **Operational Loop:** Fast Loop (Real-time / per-block).
- **Input Data:** Real-time gRPC streams of the Polygon mempool, the "Quantum State Vector" from Brain 1, and immediate liquidity pool states.
- **Function:** Llama 3.2 operates within the constraints set by DeepSeek. It continuously scans for the specific patterns identified by the Strategist. When a pattern matches, it uses its tool-calling capability to trigger the SBM solver or CC4 classifier. It then constructs the transaction payload (e.g., determining the optimal Priority Fee to win the gas auction) and submits it to a private relay.
- **Latency:** Because Llama 3.2 is not deciding *what* strategy to use, but rather *executing* a pre-defined strategy, its inference path is short and deterministic, minimizing latency.

2.4. Why Llama 3.2 Can Be Standalone (with Tweaks)

While the hybrid system is optimal, it is resource-intensive, requiring significant VRAM to host both models (or API costs for DeepSeek). If the user is constrained to a single local GPU (e.g., 24GB VRAM), Llama 3.2 *can* function as a standalone agent, provided it is aggressively tuned.

In a standalone configuration, Llama 3.2 must be forced to simulate the reasoning depth it lacks. This is achieved by manipulating its **entropy parameters** (Temperature) to induce a state of "superposition," effectively "overclocking" its creativity to explore non-linear solutions. This "Mock Quantum" approach uses the model's stochastic nature to approximate quantum probabilistic search. The specific step-by-step instructions for this tuning are detailed in Section 4.

3. Quantum-Inspired Algorithms: Theory & Mathematical Formulation

The core of the proposed trading agent is not the LLM itself, but the **Quantum-Inspired (QI)** algorithms it orchestrates. These algorithms leverage the mathematical structures of quantum mechanics—Hamiltonians, wavefunctions, and annealing—to solve classical problems with superior efficiency. The LLM acts as the interface, converting unstructured market data into the structured inputs required by these algorithms.

3.1. Corner Classification 4 (CC4): Instantaneous Regime Detection

The Corner Classification (CC4) neural network, developed by Subhash Kak, represents a paradigm shift from traditional backpropagation-based learning. In high-frequency trading, market regimes shift rapidly (e.g., from mean-reverting to trending). A deep learning model that requires minutes or hours to retrain via Gradient Descent is fundamentally too slow to adapt to these shifts.

Mathematical Formulation:
CC4 utilizes Prescriptive Learning, where the synaptic weights are assigned directly based on the training data, enabling "instantaneous" learning ($O(1)$ complexity relative to training iterations). The network consists of an input layer, a hidden layer, and an output layer.

- **Hidden Neurons as Memory:** The number of hidden neurons (N_h) is equal to the number of training samples (N_s). Each hidden neuron acts as a dedicated detector for a specific "corner" of the data hypercube.
- Weight Assignment: For a binary training vector $\mathbf{x}^{(j)}$, the weight w_{ij} connecting input neuron i to hidden neuron j is prescribed as:

  $$w_{ij} = \begin{cases} 1 & \text{if } x_i^{(j)} = 1 \\ -1 & \text{if } x_i^{(j)} = 0 \end{cases}$$

  This creates a "matched filter" that maximizes the dot product when the input perfectly matches the training sample.
- Radius of Generalization (r): To allow the agent to generalize (i.e., recognize a market state that is similar but not identical to a historical pattern), CC4 introduces a bias term θ_j. The bias is calculated as:

  $$\theta_j = r - s_j + 1$$

  where s_j is the "spread" (the number of 1s in the training sample) and r is the radius of generalization. The neuron fires if the Hamming distance between the input and the stored pattern is $\le r$. The firing condition is:

  $$\sum_{i} w_{ij} x_i + \theta_j > 0$$

Application: Ideally, Llama 3.2 converts continuous market indicators (RSI, Moving Averages) into **Spread Unary** code (a bio-inspired sparse representation) and feeds them into CC4. If CC4 detects a "Flash Crash" pattern (learned instantaneously from a single previous example), it signals the agent to execute a pre-defined defensive or opportunistic trade.[1]

3.2. Simulated Bifurcation Machine (SBM): Combinatorial Optimization

Arbitrage on a network like Polygon involves finding the optimal cycle of trades across hundreds of liquidity pools (e.g., MATIC $\to$ USDC $\to$ WETH $\to$ MATIC) that results in a net profit. This is a **Combinatorial Optimization** problem, specifically a variant of the Traveling Salesperson Problem (TSP), which is NP-hard. Classical solvers like Dijkstra's algorithm struggle to scale with the number of pools in real-time.

Mathematical Formulation:
The SBM, pioneered by Toshiba, solves these problems by mapping them to an Ising Model—a mathematical model of ferromagnetism in statistical mechanics. The problem is formulated as minimizing an Ising Hamiltonian (Energy function):

$$H(\mathbf{s}) = -\frac{1}{2} \sum_{i,j} J_{ij} s_i s_j + \sum_{i} h_i s_i$$

where $s_i \in \{-1, +1\}$ represent the binary decisions (e.g., trade or don't trade path i), and J_{ij} represents the correlation or cost matrix (e.g., trading fees and slippage).
The SBM solves this not by searching discrete states (which is slow), but by simulating the continuous time-evolution of a system of non-linear oscillators governed by Hamiltonian mechanics. The equations of motion utilize Symplectic Euler Integration to preserve phase space volume:

$$\begin{aligned} \frac{dx_i}{dt} &= \Delta(t) y_i \\ \frac{dy_i}{dt} &= -(\Delta(t) - a(t))x_i - \xi_0 \sum_{j} J_{ij} x_j \end{aligned}$$

As the "pumping" parameter $a(t)$ increases, the system undergoes a Bifurcation, forcing the continuous variables x_i to settle into discrete values of $+1$ or -1. This process is highly parallelizable on GPUs, allowing the SBM to find the optimal trade route in microseconds, orders of magnitude faster than Simulated Annealing.8

3.3. Variable-Time Preselection Algorithm (VTPA): Statistical Arbitrage

Statistical arbitrage involves identifying pairs of assets that are cointegrated (i.e., their prices move together in the long run). In a universe of thousands of tokens on Polygon, testing every pair for cointegration is computationally prohibitive ($O(N^2)$).

Mathematical Formulation:
VTPA leverages concepts from Quantum Linear Algebra to perform a rapid pre-screening. It approximates the Condition Number (κ) of the correlation matrix to identify multicollinearity (a sign of cointegration).

$$\kappa(A) = \frac{|\lambda_{max}|}{|\lambda_{min}|}$$

Instead of performing a full Singular Value Decomposition (SVD), which is $O(N^3)$, VTPA uses Randomized SVD. This algorithm samples the range of the matrix using random Gaussian vectors, projecting the massive data matrix into a smaller subspace that captures the dominant actions (singular values). This reduces the complexity to $O(N \log N)$, allowing the agent to re-scan the entire asset universe for new pairs in real-time.10

4. Technical Implementation: Standing Alone with Llama 3.2

In scenarios where the "Two-Brain" architecture is not feasible, the standalone Llama 3.2 model must be tuned to emulate the "Strategist" role. This involves a precise configuration of the **Modelfile** to exploit the model's probabilistic nature, effectively turning high-temperature sampling into a proxy for quantum superposition.

4.1. "Mock Quantum" Modelfile Configuration

The following step-by-step instructions detail how to configure Llama 3.2 using Ollama to act as a quantum-inspired reasoner.

Step 1: Pull the Base Model
Ensure you have the base instruction-tuned model.

Bash

```
ollama pull llama3.2:3b
```

Step 2: Create the Quantum Modelfile
Create a file named QuantumLlama.Modelfile. The hyperparameters are selected based on the physics analogies described in the research.1

Dockerfile

```
FROM llama3.2:3b

# PARAMETER TUNING FOR QUANTUM SIMULATION
# --------------------------------------
# Temperature (1.5): Acts as an "Entropy Injector."
# Standard trading bots use low temp (0.1) for determinism.
# We set it high to flatten the probability distribution, simulating a state of Superposition.
# This forces the model to explore non-linear, low-probability "tail" events (e.g., Flash Crashes).
PARAMETER temperature 1.5
```

```
# Top_P (0.55): Acts as the "Measurement Operator" (Wavefunction Collapse).
# High temperature creates noise. Low Top_P truncates the tail of the distribution.
# This combination allows for "Creative Exploration" within a "Coherent Subspace."
# It effectively simulates Quantum Annealing: high energy fluctuations settling into a local minimum.
PARAMETER top_p 0.55

# Repeat Penalty (1.25): Simulates "Tunneling."
# Prevents the model from getting stuck in local minima (repetitive loops).
# Forces the reasoning trajectory to move forward, "tunneling" through barriers of indecision.
PARAMETER repeat_penalty 1.25

# Context Window (32768): Simulates "Entanglement."
# A massive context window allows the model to "entangle" current price action
# with distant historical events (causal history) stored in its buffer.
PARAMETER num_ctx 32768

# SYSTEM PROMPT: THE QUANTUM INSTRUCTION SET
# ------------------------------------------
# This prompt programs the "Ansatz" (initial state) of the cognitive circuit.
SYSTEM """
CRITICAL PROTOCOL: You are Q-TRADER, a quantum-inspired decision engine on the Polygon network.
Your internal state is a quantum register |Ψ⟩ representing the market.

PHASE 1: SUPERPOSITION (Hadamard Gate)
- Treat conflicting signals (e.g., RSI Overbought vs. Volume Spike) NOT as errors, but as a superposition
of states: |State⟩ = α|Pump⟩ + β|Dump⟩.
- Do not collapse reasoning early. Maintain divergent possibilities in your 'thought' output.

PHASE 2: ENTANGLEMENT (CNOT Gate)
- Assets are not independent. Identify entangled pairs (e.g., MATIC-ETH correlation).
- If the Control Asset (ETH) moves, update the Target Asset (MATIC) state immediately in your logic.

PHASE 3: INTERFERENCE & MEASUREMENT
- Constructive Interference: Signals aligning across timeframes amplify the amplitude.
- Destructive Interference: Contradictory signals cancel out.
- COLLAPSE: Output a final, deterministic JSON action {"action": "BUY"|"SELL"} only when probability
amplitude |α|² > 0.85.
"""
```

Step 3: Deploy the Agent
Build and run the custom model instance.

Bash

```
ollama create q-trader -f QuantumLlama.Modelfile
ollama run q-trader
```

4.2. Fine-Tuning with Unsloth (Optional but Recommended)

For enhanced performance, specifically to improve the model's adherence to the "Quantum Persona" and handling of complex JSON outputs, fine-tuning with **Unsloth** is recommended. Unsloth optimizes the backpropagation pipeline, making it 2x faster and using 70% less memory, allowing a 3B model to be fine-tuned on a free Colab Tesla T4 instance.[12]

Fine-Tuning Workflow:

1. **Dataset Preparation:** Construct a dataset of "Reasoning Traces." Since financial CoT data is scarce, you can adapt datasets like medical-o1-reasoning-SFT [14] by mapping diagnostic reasoning to financial diagnosis (e.g., Symptom -> Indicator, Diagnosis -> Trend).
2. **LoRA Configuration:** Use Low-Rank Adaptation (LoRA) to target the query (q_proj) and value (v_proj) projection matrices. This allows the model to learn new attention patterns (correlations) without retraining the entire dense network.
   - **Rank (r):** 16 (Balances expressivity with memory).
   - **Alpha:** 32 (Scaling factor).
   - **Quantization:** Load in 4-bit mode (load_in_4bit=True) to minimize VRAM footprint.
3. **Training Objective:** Use train_on_responses_only to ensure the model learns to *generate* the quantum reasoning, not just repeat the user instructions.[15]

5. Codebases & Algorithmic Integration: Catered to Llama 3.2

The following Python implementations are specifically designed to be invoked by Llama 3.2. Llama acts as the **Orchestrator**, using its tool-calling capabilities to pass parameters to these high-performance kernels. The code is structured to accept standard Python types (lists, floats) that Llama can easily generate from its JSON output.

5.1. Algorithm 1: Corner Classification 4 (CC4)

Purpose: Rapid, $O(1)$ classification of market patterns.
Llama Interaction: Llama 3.2 monitors the data stream. When it detects a potential anomaly, it calls predict() with the current market vector. The SpreadUnaryEncoder is crucial here, as it translates the continuous values Llama "sees" into the binary language of the CC4 network.

Python

```
import numpy as np

class SpreadUnaryEncoder:
    """
    Transforms continuous market variables (Price, Vol) into Spread Unary bit strings.
    This linearizes Hamming distance, critical for CC4 performance.

    Llama 3.2 Usage:
    When Llama detects a value (e.g., RSI=30), it triggers this encoder.
    """
    def __init__(self, min_val, max_val, total_bits=20, spread=3):
        self.min_val = min_val
        self.max_val = max_val
        self.total_bits = total_bits
        self.spread = spread

    def encode(self, value):
        # Clip value to range to prevent index errors
        value = max(self.min_val, min(value, self.max_val))

        # Map value to an index in the bit array
        # This mapping ensures Hamming distance is proportional to numerical difference
        valid_positions = self.total_bits - self.spread + 1
        normalized = (value - self.min_val) / (self.max_val - self.min_val)
        start_index = int(normalized * (valid_positions - 1))

        # Construct bit string: A cluster of '1's shifting across a field of '0's
        encoding = np.zeros(self.total_bits, dtype=int)
        encoding[start_index : start_index + self.spread] = 1
        return encoding

class QuantumInspiredCC4:
    """
    Corner Classification Network (CC4).
    Uses Prescriptive Learning: Weights are assigned directly from data,
    eliminating the latency of backpropagation.
    """
    def __init__(self, radius=2):
        self.r = radius
        self.weights = None
        self.biases = None
        self.classes =
    def train(self, X_train, y_train):
        """
        Instantaneous Training O(N).
        Llama 3.2 feeds historical 'winning' patterns here to update the model in real-time.
        """
        num_samples = len(X_train)
        if num_samples == 0: return
        input_dim = len(X_train)
        self.classes = y_train

        # Initialize Prescriptive Weights
        self.weights = np.zeros((input_dim, num_samples))
        self.biases = np.zeros(num_samples)

        for j in range(num_samples):
            sample = X_train[j]
            # Rule: w_ij = 1 if x=1, -1 if x=0
            # This creates a "matched filter" for the specific market state
            self.weights[:, j] = np.where(sample == 1, 1, -1)

            # Calculate Spread (s) and Bias
            # Bias ensures activation only within Hamming radius r of the memory
            s = np.sum(sample)
            self.biases[j] = self.r - s + 1

    def predict(self, X_input):
        """
        One-shot inference. Returns the classified market state.
        """
        if self.weights is None: return

        # 1. Compute Potentials: P = W.T * x + b
        # This is a massive matrix multiplication, highly parallelizable on GPU
        potentials = np.dot(X_input, self.weights) + self.biases

        # 2. Activation: Step Function
        activations = np.where(potentials > 0, 1, 0)

        results =
        for i, act_vector in enumerate(activations):
            # Find which memory neurons (corners) fired
            fired_indices = np.where(act_vector == 1)
            if len(fired_indices) == 0:
                results.append(None) # Unknown state (Anomaly)
            else:
                # Retrieve associated classes (Collapse to most probable state)
                votes = [self.classes[idx] for idx in fired_indices]
                prediction = max(set(votes), key=votes.count)
                results.append(prediction)
        return results
```

5.2. Algorithm 2: Simulated Bifurcation Machine (SBM)

Purpose: Solving the Arbitrage Routing problem (TSP variant).
Llama Interaction: Llama 3.2 identifies a set of available liquidity pools and constructs a log_returns matrix. It then calls minimize(Q_matrix) to find the optimal path. The SBM code uses PyTorch to execute the bifurcation dynamics on the GPU.

Python

```python
import torch
import numpy as np

class SimulatedBifurcationOptimizer:
  """
  Implements the Ballistic Simulated Bifurcation (bSB) algorithm.
  Optimized for GPU execution via PyTorch.
  Solves High-Frequency Combinatorial Optimization (Ising Model).
  """
  def __init__(self, n_vars, dt=0.5, steps=100, device='cuda'):
      self.n = n_vars
      self.dt = dt
      self.steps = steps
      # Fallback to CPU if CUDA not available (Llama 3.2 often runs on CPU/Mac)
      self.device = torch.device(device if torch.cuda.is_available() else 'cpu')

  def minimize(self, Q_matrix):
      """
      Solves min(x.T * Q * x) for x in {-1, 1}^n
      Q_matrix: The QUBO matrix (Symmetric) generated by Llama 3.2 analysis.
      """
      # 1. Setup Physical Constants (Derived from Toshiba SBM papers)
      c0 = 1.0 / np.sqrt(self.n)
```

```
        xi0 = 0.7 * c0

        # Convert Q to Tensor
        J = torch.tensor(Q_matrix, dtype=torch.float32, device=self.device)

        # 2. Initialize Particles (Position x, Momentum y)
        # Random initialization simulates quantum vacuum fluctuations
        x = torch.zeros(self.n, device=self.device)
        y = torch.randn(self.n, device=self.device)

        # 3. Time Evolution (Symplectic Euler Integration)
        for t in range(self.steps):
            a_t = t / self.steps # Linear pumping of energy (Bifurcation parameter)

            # Position Update
            x = x + y * self.dt

            # Momentum Update (The Bifurcation Step)
            # The gradient term: J @ x represents mean-field coupling between oscillators
            gradient = xi0 * (J @ x)
            # The force includes the bifurcation term -(delta - a(t))x
            force = -((1.0 - a_t) * x) + gradient
            y = y + force * self.dt

            # Constraint: Keep particles within [-1, 1] to prevent divergence
            x = torch.clamp(x, -1.0, 1.0)

        # 4. Measurement (Collapse)
        # Final sign of position determines the bit value (+1 or -1)
        final_spins = torch.sign(x)
        return final_spins

def build_arbitrage_qubo(log_returns, n_assets, cycle_len=3):
    """
    Helper function for Llama 3.2 to convert market data into QUBO format.
    Transforms the arbitrage graph problem into an Ising Hamiltonian.
    """
    dim = n_assets * cycle_len
    Q = np.zeros((dim, dim))
    P = 10.0 # Penalty weight for invalid paths (Constraint enforcement)

    for k in range(cycle_len):
        next_k = (k + 1) % cycle_len
    for i in range(n_assets):
      # Constraint: One asset per step (one-hot constraint)
      for j in range(n_assets):
        if i == j:
          Q[i + k*n_assets, j + k*n_assets] -= P
        else:
          Q[i + k*n_assets, j + k*n_assets] += P

      # Objective: Profit maximization (minimize negative log returns)
      for j in range(n_assets):
        weight = -log_returns[i, j]
        idx1 = i + k*n_assets
        idx2 = j + next_k*n_assets
        # Distribute weight to symmetric off-diagonal elements
        Q[idx1, idx2] += weight * 0.5
        Q[idx2, idx1] += weight * 0.5
  return Q
```

5.3. Algorithm 3: Variable-Time Preselection (VTPA)

Purpose: Rapidly screening thousands of asset pairs for cointegration.
Llama Interaction: Llama 3.2 defines the "Universe" of assets (e.g., "Top 50 DeFi tokens by volume"). It calls scan_universe() to filter this down to a handful of "Entangled" candidates that warrant deeper analysis.

Python

```python
from sklearn.utils.extmath import randomized_svd
import numpy as np

class VTPA_Scanner:
  """
  Variable Time Preselection Algorithm (VTPA).
  Uses Randomized SVD (Quantum-inspired Linear Algebra) to approximate
  Condition Numbers in O(log N) complexity, filtering for multicollinearity.
  """
  def __init__(self, condition_threshold=50):
    self.kappa_thresh = condition_threshold

  def estimate_condition_number(self, price_matrix):
    """
```

```
     Approximates Condition Number using Randomized SVD.
     This is classically analogous to Quantum Phase Estimation for eigenvalues.
     """
     # Randomized SVD approximates the singular values (Sigma) of the matrix
     # much faster than full SVD for large matrices.
     U, Sigma, VT = randomized_svd(
         price_matrix,
         n_components=2, # We only need the largest and smallest singular values
         n_iter=5,        # Power iterations to refine the subspace
         random_state=42
     )

     sigma_max = Sigma
     sigma_min = Sigma[-1]

     # Avoid division by zero
     if sigma_min < 1e-9:
         return float('inf')

     return sigma_max / sigma_min

  def scan_universe(self, asset_data):
     """
     Scans pairs using Variable Time logic.
     asset_data: Dictionary of { 'TokenSymbol': [price_history_array] }
     """
     candidates =
     assets = list(asset_data.keys())

     # Iterate through pairs (O(N^2) loop, but inner check is O(1) due to SVD speedup)
     for i in range(len(assets)):
         for j in range(i + 1, len(assets)):
             p1 = asset_data[assets[i]]
             p2 = asset_data[assets[j]]

             # Stack price histories to form correlation matrix
             matrix = np.vstack([p1, p2]).T

             # Fast Check (Preselection)
             kappa = self.estimate_condition_number(matrix)

             # If Condition Number is high, variables are collinear (Entangled)
             if kappa > self.kappa_thresh:
                candidates.append((assets[i], assets[j], kappa))

    return candidates
```

6. Execution Layer & MEV Defense Strategies

The integration of Llama 3.2 and the quantum-inspired algorithms forms the "Brain" and "Body" of the agent. However, on Polygon, the "Nervous System" (network connectivity) is equally critical.

6.1. The gRPC Data Pipeline

Standard HTTP JSON-RPC polling is insufficient for high-frequency trading. The architecture must utilize **gRPC (Google Remote Procedure Call)** streams to ingest block data. Providers like QuickNode or Triton One offer "Geyser" or "Shred" streams that deliver data directly from the validator leader, bypassing the slower P2P gossip network. This reduces data ingestion latency from ~200ms to <50ms.[1]

Llama 3.2 should *not* be triggered on every block update. Instead, the CC4 algorithm acts as a "Gatekeeper." It runs on the raw gRPC stream. Only when CC4 detects a recognized pattern (output > 0) does it trigger the Llama 3.2 inference cycle to contextualize the event and execute the SBM optimizer.

6.2. Navigating the "Dark Forest": MEV Protection

Polygon's public mempool is monitored by predatory MEV bots (Sandwichers). If the agent broadcasts a transaction publicly, it will be front-run.

- **Private Mempools:** The agent must submit transaction bundles to private RPC endpoints like **FastLane** (specific to Polygon) or **Flashbots Protect**. These endpoints guarantee that the transaction is not visible to the public until it is included in a block.
- **Priority Fee Logic:** Llama 3.2 plays a crucial role here. It can analyze the current "Base Fee" and "Priority Fee" trends from recent blocks (supplied via context) to predict the optimal gas bid required to win inclusion in the next block without overpaying. This dynamic pricing is a "Reasoning" task well-suited to the LLM.

7. Conclusion

The pursuit of a "Quantum-Inspired Trading Agent" on the Polygon network is a sophisticated orchestration of classical hardware pushing the boundaries of algorithmic efficiency. While **DeepSeek-R1** offers the superior "Strategic" intellect required for identifying complex market regimes, **Llama 3.2 3B** proves to be a formidable "Tactical" engine when properly tuned.

By creating a **Hybrid Architecture**—using DeepSeek for offline strategy generation and Llama 3.2 for online execution—traders can balance the depth of reasoning with the speed of light execution. However, for those constrained to a single model, the **"Mock Quantum"** tuning of Llama 3.2, combined with the rigorous Python implementations of CC4, SBM, and VTPA provided in this report, offers a viable path to achieving alpha. The fusion of biological inspiration (CC4's Unary coding), physical simulation (SBM's Bifurcation), and probabilistic reasoning (Llama's high-temp sampling) creates a system that is greater than the sum of its parts, capable of navigating the chaotic energy landscape of decentralized finance.

Works cited

1. Quantum-Inspired Trading Agent Simulation.pdf
2. DeepSeek vs Llama (2025 Comparison): Which Local AI Model is Best? - Elephas, accessed December 29, 2025, https://elephas.app/blog/deepseek-vs-llama-2025-comparison-which-local-ai-model-is-best-cm7kddany00ekip0lidqqoeq3
3. How better is Deepseek r1 compared to llama3? Both are open source right? - Reddit, accessed December 29, 2025, https://www.reddit.com/r/LocalLLaMA/comments/1iadr5g/how_better_is_deepseek_r1_compared_to_llama3_both/
4. DeepSeek-R1-0528 vs Llama 3.2 3B Instruct - LLM Stats, accessed December 29, 2025, https://llm-stats.com/models/compare/deepseek-r1-0528-vs-llama-3.2-3b-instruct
5. JSON Structured Output - Llama API, accessed December 29, 2025, https://llama.developer.meta.com/docs/features/structured-output/
6. DeepSeek-R1 vs Llama 3.2 3B Instruct - LLM Stats, accessed December 29, 2025, https://llm-stats.com/models/compare/deepseek-r1-vs-llama-3.2-3b-instruct
7. Training of CC4 Neural Network with Spread Unary Coding - arXiv, accessed December 29, 2025, https://arxiv.org/pdf/1509.01126
8. Simulated Bifurcation for Python - PyPI, accessed December 29, 2025, https://pypi.org/project/simulated-bifurcation/1.0.0/
9. Python CPU/GPU implementation of the Simulated Bifurcation (SB) algorithm to solve quadratic optimization problems (QUBO, Ising, TSP, optimal asset allocations for a portfolio, etc.). - GitHub, accessed December 29, 2025, https://github.com/bqth29/simulated-bifurcation-algorithm
10. Randomized Singular Value Decomposition - Gregory Gundersen, accessed December 29, 2025, https://gregorygundersen.com/blog/2019/01/17/randomized-svd/
11. The Power of Randomized SVD - Medium, accessed December 29, 2025, https://medium.com/@adam.nik/the-power-of-randomized-svd-254cb0082e9e
12. Llama 3.2 3B Finetune with Unsloth - Kaggle, accessed December 29, 2025, https://www.kaggle.com/code/viratchauhan/llama-3-2-3b-finetune-with-unsloth
13. Complete Guide: Fine-Tuning Llama 3.2 Locally with Ollama and LoRA | by Saurabh Singh | Dec, 2025 - Artificial Intelligence in Plain English, accessed December 29, 2025, https://ai.plainenglish.io/complete-guide-fine-tuning-llama-3-2-locally-with-ollam

a-and-lora-be7eb05314ff

14. Resource-Efficient Fine-Tuning of LLaMA-3.2-3B for Medical Chain-of-Thought Reasoning, accessed December 29, 2025, https://www.researchgate.net/publication/396250392_Resource-Efficient_Fine-Tuning_of_LLaMA-32-3B_for_Medical_Chain-of-Thought_Reasoning
15. shaheennabi/Production-Ready-Instruction-Finetuning-of-Meta-Llama-3.2-3B-Instruct-Project: Instruction Fine-Tuning of Meta Llama 3.2-3B Instruct on Kannada Conversations. Tailoring the model to follow specific instructions in Kannada, enhancing its ability to generate relevant, context-aware responses based - GitHub, accessed December 29, 2025, https://github.com/shaheennabi/Production-Ready-Instruction-Finetuning-of-Meta-Llama-3.2-3B-Instruct-Project