NeuroShard: A Decentralized Architecture for Collective Intelligence

LZ

December, 2025

Abstract

We present **NeuroShard**, a fully dynamic, decentralized architecture for building and operating Large Language Models (LLMs) as a **public good** rather than a corporate asset. Instead of training a frontier model inside a single data center and selling access to its outputs, NeuroShard turns a global network of heterogeneous devices—from gaming GPUs and laptops to data center servers—into one continuously evolving optimizer that collectively trains a shared model, **NeuroLLM**, from random initialization. Every aspect of the system adapts automatically: network size, model depth, model width, vocabulary, and node roles all scale dynamically with participation.

The architecture introduces **Quorum-Based Training**, where speed-matched node groups self-organize into complete pipelines that train synchronously within the quorum while synchronizing across quorums via **DiLoCo-style gradient aggregation**. This enables heterogeneous hardware—from H100 servers to Raspberry Pis—to contribute appropriately through multiple **Contribution Modes**: pipeline membership for real-time training, asynchronous gradient submission, data provision, and proof verification.

To make this feasible in a trustless setting, we introduce **Proof of Neural Work (PoNW)** with **Optimistic Verification**, where proofs are accepted immediately and can be challenged within a time window, enabling high throughput while maintaining security. PoNW is complemented by **Byzantine-robust gradient aggregation** with $\sqrt{n}$-weighted contributions for fair influence, and by **Governance-Based Vocabulary Expansion** to prevent coordinator attacks. Together, these mechanisms form a protocol in which every useful gradient, every contributed sample, and every token generated pushes NeuroLLM forward—demonstrating that a competitive, community-owned LLM can be constructed and governed by its users, one gradient at a time.

Contents

1 Introduction **6**
1.1 The Problem with Centralized AI 6
1.2 NeuroShard: A Decentralized Solution 6
1.3 Contributions 7

2 Related Work **7**
2.1 Distributed Deep Learning 7
2.1.1 Data Parallelism and Federated Learning 7
2.1.2 Pipeline Parallelism 8
2.1.3 Decentralized Training 8
2.2 Decentralized AI Networks 8
2.2.1 Bittensor 8
2.2.2 Gensyn 8
2.3 Key Differentiators 8

3 NeuroLLM Architecture 8
3.1 Design Philosophy . . . 8
3.2 Architecture Details . . . 9
3.2.1 RMSNorm (Root Mean Square Normalization) . . . 9
3.2.2 Rotary Position Embeddings (RoPE) . . . 9
3.2.3 Grouped Query Attention (GQA) . . . 9
3.2.4 SwiGLU Activation . . . 9
3.3 Dynamic Scaling via Adaptive Architecture . . . 9
3.3.1 The Scaling Algorithm . . . 9
3.3.2 Automated Architecture Updates . . . 10
3.4 Dynamic Scaling via Layer Pooling . . . 10
3.4.1 The Dynamic Layer Pool . . . 10
3.4.2 Contribution-Based Rewards . . . 10
3.4.3 Example: Network Growth . . . 11
3.5 Forward Pass . . . 11
3.6 Memory Requirements - Dynamic Architecture . . . 11

4 Network Topology and Node Roles 12
4.1 Speed Tiers . . . 12
4.2 Contribution Modes . . . 12
4.2.1 Async Mode Freshness Decay . . . 12
4.3 Node Capability Assessment . . . 13

5 Decentralized Training 13
5.1 Training Overview . . . 13
5.2 Quorum-Based Training . . . 13
5.2.1 Quorum Properties . . . 14
5.2.2 Quorum Formation Algorithm . . . 14
5.2.3 Multiple Quorums at Different Speeds . . . 14
5.3 Genesis Dataset and Deterministic Sharding . . . 15
5.3.1 Data Provider Nodes . . . 15
5.3.2 Quorum Roles . . . 15
5.4 Within-Quorum Pipeline Training . . . 16
5.4.1 Gradient Compression . . . 16
5.5 Cross-Quorum Synchronization (DiLoCo) . . . 16
5.5.1 Local Inner Loop . . . 16
5.5.2 Cross-Quorum Aggregation . . . 17
5.5.3 Outer Optimizer . . . 17
5.5.4 Bandwidth Reduction . . . 17
5.6 Training Rewards . . . 17
5.6.1 Gradient Weighting (Fair Influence) . . . 18
5.6.2 Scarcity-Based Incentives . . . 18
5.6.3 Role Multipliers . . . 18

6 System Architecture 18
6.1 High-Level Overview . . . 18
6.2 NeuroNode . . . 19
6.3 P2P Network Layer . . . 20
6.3.1 Peer Discovery . . . 20
6.3.2 Gradient Gossip . . . 20
6.4 Decentralized Inference Protocol . . . 20
6.4.1 Inference Flow . . . 20

6.4.2 Dynamic Pricing . . . 20

7 Proof of Neural Work (PoNW) 21
7.1 Overview . . . 21
7.2 PoNW as the Model Builder . . . 21
7.3 PoNW Components . . . 21
7.4 Optimistic Verification . . . 22
7.4.1 Challenge Protocol . . . 22
7.4.2 Game Theory . . . 22
7.4.3 Adaptive Verification Rate . . . 23
7.5 The Training-Mining Equivalence . . . 23
7.6 Reward Function . . . 23
7.7 Proof Generation . . . 24
7.8 Proof Validation . . . 24
7.9 Gradient Verification . . . 24
7.9.1 Statistical Verification . . . 24
7.9.2 Loss-Based Verification . . . 24
7.9.3 Cross-Validation . . . 25
7.10 Attack Prevention . . . 25
7.11 Cryptographic Security Model . . . 25

8 Verification and Staking 26
8.1 Work-First, Stake-Enhanced . . . 26
8.2 Verifier Mode . . . 26
8.3 Staking Requirements . . . 26
8.4 Slashing Conditions . . . 26
8.5 Diminishing Stake Returns . . . 27
8.6 Verifier Economics . . . 27
8.6.1 Verifier Rewards . . . 27
8.7 Why Work-Plus-Stake Works . . . 27

9 Robustness and Anti-Poisoning 28
9.1 The Model Poisoning Threat . . . 28
9.2 Robust Aggregation . . . 28
9.3 Adversarial Resistance Matrix . . . 29
9.4 Cross-Quorum Verification . . . 29
9.5 The Fraud Proof System . . . 29

10 Governance and The NeuroDAO 29
10.1 What NeuroDAO Does NOT Control . . . 29
10.2 What NeuroDAO Controls . . . 30
10.3 Vocabulary Governance . . . 30
10.3.1 Vocabulary Proposal Process . . . 30
10.4 Model Architecture Governance . . . 30

11 NEURO Token Economics 31
11.1 Token Utility . . . 31
11.2 Genesis Block and Transparency . . . 31
11.3 Deflationary Mechanics . . . 31
11.4 Earning NEURO . . . 32
11.4.1 Privacy-Preserving Distributed Inference Marketplace . . . 32
11.4.2 Multipliers and Bonuses . . . 33

11.5 Spending NEURO . . . 34
11.6 Economic Equilibrium . . . 34

12 Security Considerations **34**
12.1 Threat Model . . . 34
12.2 Data Privacy . . . 34
12.3 Model Integrity . . . 34
12.4 Sybil Resistance . . . 35

13 Checkpoint System and State Sharding **35**
13.1 Sharded Checkpoints . . . 35
13.2 Checkpoint Structure . . . 35
13.3 Checkpoint Synchronization . . . 35
13.4 Model Evolution Over Time . . . 36
13.4.1 Continuous Training . . . 36
13.4.2 Layer Growth System . . . 36
13.4.3 Layer Addition Sequence . . . 36
13.4.4 Identity Initialization . . . 36
13.5 The Intelligence Ledger . . . 37

14 Implementation **37**
14.1 Technology Stack . . . 37
14.2 API Endpoints . . . 37
14.3 Desktop Application . . . 37

15 Scale Adaptation and Network Phases **38**
15.1 Network Phases . . . 38
15.2 Adaptive Parameters . . . 38

16 Vision and Roadmap **38**
16.1 Foundation . . . 38
16.2 Robust Training . . . 39
16.3 Scale . . . 39
16.4 Maturity . . . 39

17 Distributed Architecture and Parallelism **39**
17.1 The Dynamic Approach . . . 39
17.2 Advanced Parallelism . . . 40
17.2.1 Tensor Parallelism . . . 40
17.2.2 Pipeline Parallelism . . . 40
17.3 Hybrid Parallelism . . . 40
17.4 Training at Scale . . . 40
17.5 The Key Insight . . . 41

18 Quorum Resilience and Fault Tolerance **41**
18.1 Design Philosophy: Throughput over Latency . . . 41
18.2 Quorum Health and Replacement . . . 41
18.3 Multipath Routing (Fault-Tolerant) . . . 41
18.3.1 Key Directive . . . 41
18.3.2 Multipath Routing . . . 41
18.3.3 Probabilistic Send with Failover . . . 42
18.3.4 Capacity Bitmask Heartbeats . . . 42
18.4 Asynchronous Micro-Batch Engine . . . 42

18.4.1 Key Directive 42
18.4.2 Activation Buffering 43
18.4.3 Interleaved 1F1B Schedule 43
18.4.4 Dynamic Micro-Batch Sizing 43
18.5 Extreme Gradient Accumulation (DiLoCo-Style) 43
18.5.1 Key Directive 43
18.5.2 Local Inner Loop 43
18.5.3 Outer Optimizer 44
18.5.4 Bandwidth Reduction 44
18.5.5 Enhanced Verification 44
18.6 Speculative Checkpointing 44
18.7 Summary: Resilience Through Design 45

19 Philosophy: Why Decentralized AI Matters **45**
19.1 The Concentration Problem 45
19.2 The NeuroShard Principles 45
19.2.1 Principle 1: Intelligence Should Be a Public Good 45
19.2.2 Principle 2: Contributors Should Own What They Build 45
19.2.3 Principle 3: No Single Point of Control 45
19.2.4 Principle 4: Start Simple, Grow Together 46
19.3 The Long-Term Vision 46

20 Conclusion **46**

21 Acknowledgments **47**

A Code Examples **47**
A.1 Starting a NeuroNode 47
A.2 Contributing Training Data 48
A.3 Generating Text 48

B Configuration Reference **49**
B.1 NeuroLLM Dynamic Architecture Config 49
B.2 Quorum and Training Configuration 49

1 Introduction

1.1 The Problem with Centralized AI

The current landscape of artificial intelligence is defined by a **small number of vertically integrated AI platforms** that own the models, the data, and the distribution channels. Large Language Models (LLMs) such as GPT-5, Claude, and LLaMA are developed, trained, and deployed by a handful of organizations that decide who is allowed to build on top of them and under what terms. This concentration of power is not just an economic issue; it is an architectural bottleneck that constrains what kinds of AI systems we can build.

- **Centralized Control:** Alignment, safety, and access policies are set unilaterally, with limited transparency or public oversight. A single TOS change can reshape what billions of users are allowed to ask or build.
- **Infrastructure Barriers:** Training frontier models requires tightly managed, billion-dollar clusters. This shuts out independent researchers, communities, and smaller organizations from ever creating models at comparable scale.
- **Compute Latency and Waste:** While hyperscale data centers consume enormous amounts of energy, the world's edge compute, consumer GPUs, idle CPUs, and phones, remains largely dark and unused for AI.
- **Data Privacy and Sovereignty:** Centralized training aggregates raw data into monolithic repositories, creating systemic privacy risk and making it difficult for communities or individuals to control how their data shapes the models that govern their lives.

1.2 NeuroShard: A Decentralized Solution

NeuroShard proposes a decentralized alternative: a protocol for **collaborative model creation** where the network, not any single operator, is the place where the model lives. Instead of distributing pre-trained weights from a centralized lab, NeuroShard coordinates the training of a new model, **NeuroLLM**, from scratch using the aggregated compute and data of participating nodes. Each node that joins strengthens both the model and the network, and each contribution is cryptographically accounted for.

The fundamental principle is that **everything is dynamic**. There are no fixed roles, sizes, speeds, prices, or quorums. The only constants are the rules of the game: how to discover peers (DHT), how to verify work (PoNW), how to aggregate gradients (robust methods), and how to resolve disputes (stake slashing).

- **Quorum-Based Training:** Speed-matched nodes self-organize into **Quorums**—complete training pipelines that process batches synchronously within the group while synchronizing across quorums via DiLoCo-style gradient aggregation. This enables heterogeneous hardware to contribute at appropriate speeds.
- **Multi-Mode Contribution:** Every device can contribute something. Fast GPU nodes join pipeline quorums for real-time training. Slow devices submit asynchronous gradients. Any node can provide Genesis data or verify proofs. From H100 servers to Raspberry Pis, all participate meaningfully.
- **Organic Scalability:** The model architecture adapts automatically. Using a **Dynamic Layer Pool**, model depth and width grow as network capacity increases, without fixed "model sizes" or synchronized upgrade phases.

- **Optimistic Verification:** Through **Proof of Neural Work (PoNW)**, proofs are accepted immediately and can be challenged within a time window. This enables high throughput while maintaining economic security through stake-based challenges.
- **Byzantine Tolerance:** Training is secured through $\sqrt{n}$-weighted gradient aggregation with robust statistics (Trimmed Mean, Krum, Coordinate-wise Median), ensuring fair influence proportional to work and rejecting poisoned gradients.
- **Economic Alignment:** The NEURO token aligns incentives with scarcity-based rewards—nodes holding under-replicated layers earn more, naturally balancing the network. The token is backed by the network's collective computational intelligence.

1.3 Contributions

This paper presents the following contributions:

- **Dynamic Model Architecture:** A transformer implementation that scales depth and width based on real-time network topology via a distributed layer registry, with automatic layer growth as capacity increases.
- **Quorum-Based Training:** A novel approach where speed-matched node groups self-organize into complete training pipelines, enabling heterogeneous hardware to contribute appropriately.
- **Multi-Mode Contribution:** Support for multiple contribution modes (pipeline, async, data, verify) so every device from H100 servers to Raspberry Pis can participate meaningfully.
- **DiLoCo Cross-Quorum Synchronization:** Extreme gradient accumulation with periodic cross-quorum sync, reducing bandwidth by 90%+ while maintaining convergence.
- **Proof of Neural Work (PoNW):** An optimistic verification consensus algorithm where proofs are accepted immediately and challengeable, enabling high throughput.
- **Byzantine-Robust Aggregation:** Implementation of $\sqrt{n}$-weighted gradient aggregation with Trimmed Mean and Coordinate-wise Median to ensure fair influence and reject poisoned gradients.
- **Governance-Based Vocabulary:** Decentralized vocabulary expansion through stake-weighted governance to prevent coordinator attacks.
- **Adaptive Protocol:** The same protocol works from 1 node to millions, with parameters adapting automatically to network size.

2 Related Work

2.1 Distributed Deep Learning

Scaling deep learning beyond a single device has historically followed three paradigms:

2.1.1 Data Parallelism and Federated Learning

Federated Learning (FL) [1] enables training across decentralized devices by keeping data local and aggregating gradients. However, FL assumes each client can hold the full model in memory and requires a central aggregator. NeuroShard extends FL by eliminating the central aggregator through gossip-based gradient sharing.

2.1.2 Pipeline Parallelism

Pipeline parallelism [2] partitions models vertically by layers. Systems like Petals [3] use this for distributed inference. NeuroShard goes further by enabling distributed *training*, not just inference.

2.1.3 Decentralized Training

Prior work on decentralized training (e.g., Hivemind [4]) focuses on coordinating training of existing model architectures. NeuroShard introduces a model designed from the ground up for decentralized training.

2.2 Decentralized AI Networks

2.2.1 Bittensor

Bittensor [5] creates a decentralized intelligence market where nodes are rewarded based on the quality of their outputs. However, Bittensor treats models as black boxes and focuses on output validation rather than collective model creation. NeuroShard creates the model itself through collective training.

2.2.2 Gensyn

Gensyn [6] focuses on verifiable compute for ML training. NeuroShard incorporates similar verification concepts through PoNW while also providing the model architecture and training protocol.

2.3 Key Differentiators

NeuroShard is unique in combining:

- A **custom model architecture** designed for decentralized training (NeuroLLM)
- **Gradient gossip** for decentralized SGD without central coordination
- **Training rewards** that economically incentivize compute and data contribution
- **From-scratch training** rather than distributing pre-existing models

3 NeuroLLM Architecture

3.1 Design Philosophy

NeuroLLM is not a fork or fine-tune of an existing model. It is a completely new transformer architecture designed with the following goals:

1. **Efficient Gradients:** Optimized for gradient compression and gossip transmission
2. **Stable Training:** Uses techniques that are robust to asynchronous, noisy gradient updates
3. **Scalable Architecture:** Can grow from 125M to 70B+ parameters as the network matures
4. **Privacy-Compatible:** Supports differential privacy in training data

3.2 Architecture Details

NeuroLLM uses a modern transformer architecture with the following components:

3.2.1 RMSNorm (Root Mean Square Normalization)

Unlike LayerNorm, RMSNorm [7] is more stable for distributed training and requires fewer parameters:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2 + \epsilon}} \cdot \gamma \tag{1}$$

3.2.2 Rotary Position Embeddings (RoPE)

RoPE [8] encodes positional information directly into attention computations, enabling:

- No fixed maximum sequence length
- Better extrapolation to longer contexts
- No learnable position parameters (reducing gradient noise)

3.2.3 Grouped Query Attention (GQA)

GQA [9] reduces memory usage by sharing key-value heads across multiple query heads:

- 12 query heads, 4 key-value heads (bootstrap phase)
- 3x reduction in KV cache size
- Minimal quality degradation

3.2.4 SwiGLU Activation

SwiGLU [10] provides better gradient flow than ReLU or GELU:

$$\text{SwiGLU}(x) = \text{Swish}(xW_{\text{gate}}) \otimes (xW_{\text{up}}) \tag{2}$$

3.3 Dynamic Scaling via Adaptive Architecture

Traditional distributed training relies on static model architectures (e.g., "GPT-3 175B has 96 layers × 12,288 dim"). NeuroShard introduces **Adaptive Architecture Scaling**, where both model width (hidden dimension) and depth (number of layers) adapt automatically to total network capacity.

3.3.1 The Scaling Algorithm

NeuroShard's architecture follows empirical scaling laws derived from GPT-3 [11] and Chinchilla [12]:

$$\text{Optimal Architecture} = f(\text{Total Network Memory}) \tag{3}$$

Where the function f balances width and depth according to:

$$\text{Width (hidden\_dim)} \propto M^{0.6} \tag{4}$$

$$\text{Depth (num\_layers)} \propto M^{0.4} \tag{5}$$

This ensures width grows faster than depth, which empirically produces more efficient models [13].

3.3.2 Automated Architecture Updates

The network automatically recalculates optimal architecture every 50 nodes:

1. **Calculate Total Capacity:** $M_{\text{total}} = \sum_{i=1}^{N} M_i$ where M_i is each node's memory
2. **Compute Optimal Architecture:** Use scaling formulas to determine best width × depth
3. **Check Improvement:** Only upgrade if new architecture is $\geq 1.3\times$ better
4. **Trigger Migration:** New nodes use new architecture; existing nodes migrate gradually

No Votes Required: Architecture scaling is fully automated based on network capacity, not governance votes.

3.4 Dynamic Scaling via Layer Pooling

The **Dynamic Layer Pool** manages layer assignment across nodes using the current architecture.

3.4.1 The Dynamic Layer Pool

The network maintains a DHT-based registry of model layers:

1. **Resource Advertisement:** Upon joining, a node announces its available VRAM and compute capacity.
2. **Layer Allocation:** The protocol assigns a specific range of transformer layers to the node. This assignment is dynamic; as nodes join or leave, layers are redistributed to ensure redundancy ($R \geq 2$ replicas per layer).
3. **Organic Growth:** The total depth of the model is a function of the aggregate network memory. There are no pre-set "sizes" (e.g., 7B, 70B); the model grows layer-by-layer as capacity permits.

$$\text{Model Depth} \approx \frac{\sum_{i=1}^{N} \text{AvailableMemory}_i \times \alpha}{\text{Memory per Layer}} \tag{6}$$

where α is a utilization factor (e.g., 0.8).

This architecture allows for **Pipeline Parallelism** by default: inference and training forward passes traverse the network, routing activations from one node's assigned layers to the next.

3.4.2 Contribution-Based Rewards

Nodes earn NEURO proportional to their contribution:

$$R_{\text{multiplier}} = 1.0 + \frac{\text{Layers Held}}{\text{Total Network Layers}} \times (1 + 0.1 \cdot \mathbb{1}_{\text{embed}} + 0.1 \cdot \mathbb{1}_{\text{head}}) \tag{7}$$

Where $\mathbb{1}_{\text{embed}}$ and $\mathbb{1}_{\text{head}}$ are indicator functions for holding the embedding layer and LM head respectively.

3.4.3 Example: Network Growth

Table 1: Dynamic Model Growth Example (Width × Depth)

Network State	Total Memory	Architecture	Params	Comparable To
1 node (2GB)	2GB	8L × 512H	50M	GPT-2 Small
10 nodes (4GB avg)	40GB	16L × 1024H	350M	GPT-2 Large
50 nodes (6GB avg)	300GB	24L × 2048H	2.7B	GPT-3 Small
100 nodes (8GB avg)	800GB	32L × 3072H	9.2B	GPT-3 Medium
500 nodes (8GB avg)	4TB	48L × 5120H	47B	LLaMA 2 70B
1000 nodes (8GB avg)	8TB	64L × 7168H	123B	GPT-4 class

Key Insight: The model grows in BOTH width (hidden dimension) and depth (layers), following empirical scaling laws from GPT-3 and Chinchilla research. This produces efficient, well-architected models at any scale. No fixed dimensions - purely capacity-driven.

3.5 Forward Pass

The NeuroLLM forward pass follows standard transformer architecture:

Algorithm 1 NeuroLLM Forward Pass

```
procedure FORWARD($input\_ids$)
    $x \leftarrow \text{TokenEmbedding}(input\_ids)$
    for $layer \in [0, \text{num\_layers})$ do
        $x \leftarrow x + \text{Attention}(\text{RMSNorm}(x))$        ▷ Pre-norm
        $x \leftarrow x + \text{SwiGLU}(\text{RMSNorm}(x))$
    end for
    $x \leftarrow \text{RMSNorm}(x)$
    $logits \leftarrow \text{LinearProjection}(x)$
    return $logits$
end procedure
```

3.6 Memory Requirements - Dynamic Architecture

NeuroLLM's architecture adapts to network capacity. Each node contributes based on available memory AND current network size:

Table 2: Layer Capacity (varies by network architecture)

Available RAM	Network: 10 nodes	Network: 100 nodes	Network: 1000 nodes
	Architecture adapts: 16L×1024H → 32L×3072H → 64L×7168H		
1GB	2-3 layers (30M)	1-2 layers (55M)	1 layer (120M)
2GB	4-6 layers (60M)	3-4 layers (110M)	2-3 layers (240M)
4GB	10-12 layers (120M)	6-8 layers (220M)	4-6 layers (480M)
8GB	20-24 layers (250M)	12-16 layers (440M)	8-10 layers (960M)
16GB	Full model (350M)	24-32 layers (880M)	16-20 layers (1.9B)

Key Principle: Architecture scales with total network capacity. Early nodes hold entire small model. As network grows, model becomes wider and deeper, with layers distributed across nodes. Every device participates optimally.

4 Network Topology and Node Roles

4.1 Speed Tiers

Every node self-benchmarks upon joining and periodically thereafter. Nodes are classified into **Speed Tiers** based on their forward pass latency per layer:

Table 3: Speed Tier Classification

Tier	Latency/Layer	Hardware Examples	Primary Role
T1	< 10ms	H100, A100	Pipeline leader
T2	10–50ms	RTX 4090, 3090	Pipeline member
T3	50–200ms	RTX 3060, good CPU	Pipeline member
T4	200–1000ms	Older GPU, standard CPU	Pipeline or async
T5	> 1000ms	Raspberry Pi, old hardware	Async only

The speed tier determines which quorums a node can join. Quorums are formed from **compatible speed tiers** (e.g., T1–T2, T2–T3) to avoid pipeline stalls from slow nodes.

4.2 Contribution Modes

Every node operates in one of six contribution modes based on its capabilities and network needs:

1. **Pipeline Mode:** Real-time quorum member. Processes activations in synchronized pipeline. Highest rewards. Requires speed tier T1–T4.
2. **Async Mode:** Offline training with periodic gradient submission. Works for any speed tier, especially T5 nodes too slow for real-time pipelines. Process: download weights $\rightarrow$ train locally $\rightarrow$ compute pseudo-gradient $\Delta w = w_{\text{initial}} - w_{\text{final}}$ $\rightarrow$ submit to cohort sync.
3. **Data Mode:** Store and serve Genesis training data shards. Earns storage and bandwidth payments.
4. **Verify Mode:** Monitor and challenge PoNW proofs. Earns fraud bounties when detecting cheaters.
5. **Inference Mode:** Serve inference requests only. Market-priced per token.
6. **Idle Mode:** Available but not actively contributing.

4.2.1 Async Mode Freshness Decay

Async contributions are weighted by freshness to prioritize recent work:

$$\text{freshness} = \begin{cases} 1.0 & \text{if age} < 1 \text{ hour} \\ 0.9 & \text{if age} < 1 \text{ day} \\ 0.7 & \text{if age} < 1 \text{ week} \\ 0.3 & \text{otherwise} \end{cases} \tag{8}$$

Combined with $\sqrt{\text{batches}}$ weighting, this ensures async contributors are rewarded fairly while prioritizing fresh gradients.

$$\text{Mode Selection} = f(\text{SpeedTier}, \text{QuorumAvailability}, \text{NetworkNeeds}) \tag{9}$$

The key insight is that **every device contributes something**. A Raspberry Pi with 1GB RAM cannot join a real-time pipeline, but it can train asynchronously and submit gradients when ready. A high-bandwidth node without GPU can serve Genesis data. A node with stake but limited compute can verify proofs.

4.3 Node Capability Assessment

Each node advertises its capabilities to the DHT:

- **Memory:** Available RAM/VRAM in MB
- **Compute Speed:** Measured time for one layer forward pass
- **Bandwidth:** Network throughput in Mbps
- **Layer Range:** Which layers the node currently holds
- **Quorum ID:** Current quorum membership (if any)
- **Stake:** NEURO tokens staked
- **Reputation:** Historical success rate and uptime

5 Decentralized Training

5.1 Training Overview

NeuroLLM is trained through a **quorum-based** decentralized process with two levels of synchronization:

1. **Quorum Formation:** Speed-matched nodes self-organize into quorums—complete pipelines covering all layers.
2. **Within-Quorum Training:** Quorum members train synchronously, processing batches as a pipeline (forward pass flows through nodes, backward pass returns gradients).
3. **Cross-Quorum Sync:** Periodically (every N batches), all holders of each layer synchronize via DiLoCo-style gradient aggregation.
4. **Async Contributions:** Slow nodes (T5) train offline and submit gradients to the next sync round.
5. **Reward Distribution:** NEURO tokens distributed based on contribution with $\sqrt{n}$ weighting for fair influence.

5.2 Quorum-Based Training

A **Quorum** is a self-organized group of nodes that together hold a complete model and train as a unit:

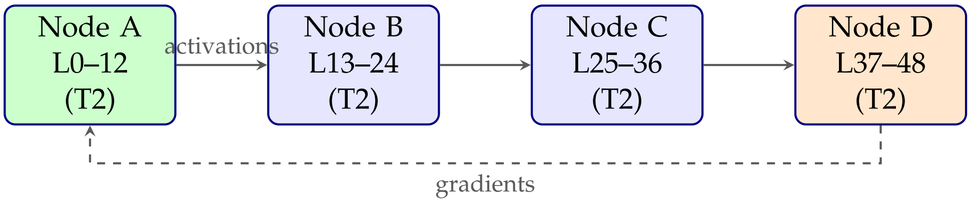


Figure 1: A Quorum: Complete pipeline of speed-matched nodes. Forward pass flows left-to-right; gradients flow back.

5.2.1 Quorum Properties

- **Complete:** Covers all layers from embedding (L0) to LM head
- **Speed-Matched:** All members in compatible speed tiers (e.g., T2–T3)
- **Self-Sufficient:** Can train independently without coordinator
- **Temporary:** Sessions last ∼1 hour, then renew or dissolve

5.2.2 Quorum Formation Algorithm

When a node wants to train, it attempts to form or join a quorum:

Algorithm 2 Quorum Formation

```
procedure FormQuorum($initiator$)
    $myTier \leftarrow initiator.speedTier$
    $myLayers \leftarrow initiator.layerRange$
    $missing \leftarrow$ AllLayers $-$ $myLayers$
    $compatible \leftarrow$ GetCompatibleTiers($myTier$)

    for $layer \in missing$ do
        $candidates \leftarrow$ DHT.GetLayerHolders($layer$)
        Filter by compatible tiers and quorum availability
    end for

    Greedy selection: maximize coverage, minimize latency

    if complete coverage found then
        Propose quorum to all selected members
        if all accept then
            Register quorum in DHT
            return new Quorum
        end if
    end if
    return null                                  ▷ Fall back to async mode
end procedure
```

5.2.3 Multiple Quorums at Different Speeds

The network naturally forms multiple quorums operating at different speeds:

Table 4: Example Quorum Distribution

Quorum Type	Speed Tiers	Throughput	Characteristics
Fast	T1–T2	∼16 batch/s	Data center GPUs
Medium	T2–T3	∼3 batch/s	Gaming GPUs
Slow	T3–T4	∼0.5 batch/s	CPU-heavy nodes
Async	T5	Variable	Train offline, submit later

Each quorum is internally synchronized but operates asynchronously relative to other quorums. Cross-quorum synchronization happens via DiLoCo.

5.3 Genesis Dataset and Deterministic Sharding

To solve the "garbage in, garbage out" problem inherent in user-submitted data, NeuroShard employs a **Genesis Dataset Strategy**. Instead of users submitting arbitrary text, the training data is drawn from a cryptographically verified manifest of high-quality, open-source datasets (e.g., FineWeb, RedPajama).

5.3.1 Data Provider Nodes

Nodes in **Data Mode** store and serve Genesis dataset shards. The dataset is sharded deterministically:

$$\text{Assigned Shard ID} = \text{Hash}(\text{Node ID}) \pmod{\text{Total Shards}} \tag{10}$$

This deterministic assignment enables **Proof of Neural Work Verification**:

- A peer can verify data by downloading the same shard and checking the hash.
- Verifiers can re-compute the embedding layer output to validate initiator work.
- This forces data providers to store and serve real training data to earn rewards.

5.3.2 Quorum Roles

Within each quorum, nodes naturally assume roles based on their layer assignments:

1. **Initiator (Embedding Layer):** Fetches batch from Genesis data providers, embeds tokens, and starts the forward pass. Attaches ground-truth labels.
2. **Processors (Middle Layers):** Receive activations, perform forward pass through local layers, and pass to next node. Do not need dataset access.
3. **Finisher (LM Head):** Computes loss between output and labels, initiates backward pass, distributes gradients back through the pipeline.

This architecture ensures that massive datasets (terabytes) can be used without requiring every participant to download them. Only initiator nodes and data providers pay the storage/bandwidth cost, for which they are compensated with bonus multipliers.

Table 5: Quorum Role Bonuses

Role	Responsibility	Bonus
Initiator	Embedding + data fetch	+20%
Processor	Forward/backward layers	Base rate
Finisher	Loss + backward initiation	+30%

5.4 Within-Quorum Pipeline Training

Within a quorum, training follows a synchronized pipeline:

Algorithm 3 Quorum Pipeline Training

```
procedure QuorumTrainingStep
    if Role = Initiator then
        $batch, labels \leftarrow$ GenesisLoader.get_batch()
        $embeddings \leftarrow$ Embed($batch$)
        $hidden \leftarrow$ ForwardMyLayers($embeddings$)
        SendToNextNode($hidden, labels$)
    else if Role = Processor then
        $input, labels \leftarrow$ ReceiveFromPrevNode()
        $output \leftarrow$ ForwardMyLayers($input$)
        SendToNextNode($output, labels$)
    else if Role = Finisher then
        $final, labels \leftarrow$ ReceiveFromPrevNode()
        $loss \leftarrow$ CrossEntropyLoss($final, labels$)
        Backward($loss$)
        PropagateGradientsBack()
    end if
end procedure
```

5.4.1 Gradient Compression

To minimize P2P bandwidth usage, NeuroShard implements a multi-stage compression pipeline for gradient updates:

1. **Top-K Sparsification:** Only the largest k% of gradient elements (by magnitude) are transmitted.
2. **INT8 Quantization:** Selected values are quantized to 8-bit integers.
3. **Entropy Coding:** The sparse, quantized stream is further compressed using zlib.
4. **Error Feedback:** Residuals (discarded values) are accumulated locally and added to the next step's gradient, ensuring convergence.

This pipeline achieves compression ratios of 50-100x with minimal impact on model convergence.

5.5 Cross-Quorum Synchronization (DiLoCo)

The key insight of NeuroShard is that quorums train independently most of the time, synchronizing only periodically. This DiLoCo-style approach reduces bandwidth by 90%+ while maintaining convergence.

5.5.1 Local Inner Loop

Instead of syncing gradients every step, each quorum:

1. Saves initial weights w_0
2. Trains independently for N steps (e.g., $N = 500$)

3. Computes pseudo-gradient: $\Delta w = w_0 - w_N$

4. Only then triggers cross-quorum sync

5.5.2 Cross-Quorum Aggregation

At sync time, all holders of each layer form a **Layer Cohort** and exchange pseudo-gradients:

Algorithm 4 Cross-Quorum Sync (DiLoCo)

```
procedure CohortSync
    $\Delta w \leftarrow w_{\text{initial}} - w_{\text{current}}$        ▷ Pseudo-gradient
    $cohort \leftarrow$ FindLayerCohort(myLayers)
    $contributions \leftarrow$ [MyContribution]
    for $peer \in cohort$ do
        $contributions$.append(RequestGradient($peer$))
    end for
    $aggregated \leftarrow$ WeightedRobustAggregate($contributions$)
    $w_{\text{new}} \leftarrow w_{\text{initial}} + \eta_{\text{outer}} \cdot aggregated$
    $w_{\text{initial}} \leftarrow w_{\text{new}}$        ▷ Reset for next round
end procedure
```

5.5.3 Outer Optimizer

The aggregated pseudo-gradients are applied using an outer optimizer with Nesterov momentum:

$$m_{t+1} = \beta \cdot m_t + \Delta w_{\text{aggregated}} \tag{11}$$

$$w_{t+1} = w_t + \eta_{\text{outer}} \cdot (\beta \cdot m_{t+1} + \Delta w_{\text{aggregated}}) \tag{12}$$

Where $\beta = 0.9$ (momentum) and $\eta_{\text{outer}} = 0.7$ (outer learning rate).

5.5.4 Bandwidth Reduction

Table 6: Bandwidth Comparison: Standard vs DiLoCo

Metric	**Standard Gossip**	**DiLoCo (N=500)**
Syncs per 1000 steps	1000	2
Bandwidth per step	100%	0.2%
Straggler tolerance	Low	High

5.6 Training Rewards

Training is the **dominant** reward activity in NeuroShard. Contributors earn NEURO tokens based on their contribution with $\sqrt{n}$ weighting for fair influence:

$$R_{\text{training}} = R_{\text{base}} \times L_{\text{count}} \times B_{\text{count}} \times M_{\text{role}} \times M_{\text{scarcity}} \times M_{\text{stake}} \tag{13}$$

Where:

- $R_{\text{base}} = 0.0005$ NEURO per batch per layer
- L_{count} = number of layers held

- B_{count} = number of batches processed
- M_{role} = role-based multiplier (initiator, processor, finisher)
- M_{scarcity} = bonus for holding under-replicated layers
- M_{stake} = logarithmic stake bonus (capped)

5.6.1 Gradient Weighting (Fair Influence)

To prevent large nodes from dominating the aggregation, contributions are weighted by $\sqrt{\text{batches}}$ rather than batches directly:

$$w_i = \sqrt{B_i} \times \text{freshness}_i \tag{14}$$

This ensures that a node processing 100 batches has $\sqrt{100} = 10\times$ the influence of a node processing 1 batch, not $100\times$.

5.6.2 Scarcity-Based Incentives

Nodes holding under-replicated layers earn more, naturally balancing the network:

$$M_{\text{scarcity}} = 1 + 0.5 \times \frac{\text{TargetReplicas} - \text{ActualReplicas}}{\text{TargetReplicas}} \tag{15}$$

If a layer has only 1 replica but the target is 3, the scarcity bonus is $1 + 0.5 \times \frac{2}{3} = 1.33\times$.

5.6.3 Role Multipliers

- **Initiator (Embedding):** 1.2x (compensates for data bandwidth)
- **Finisher (LM Head):** 1.3x (compensates for loss computation)
- **Processor (Middle):** 1.0x base rate

Example: A processor node holding 25 layers with 50% scarcity bonus, 100 NEURO staked:

- Base: $0.0005 \times 25 \times 60 = 0.75$ NEURO/min
- Scarcity: $1.5\times$
- Stake bonus: $1 + 0.1 \times \log_2(1 + 100/1000) = 1.01\times$
- Total: $0.75 \times 1.5 \times 1.01 = 1.14$ NEURO/min ($\sim$1,640 NEURO/day)

6 System Architecture

6.1 High-Level Overview

NeuroShard consists of three main components:

1. **NeuroNode:** The core node that runs NeuroLLM, handles training and inference
2. **P2P Network:** Kademlia DHT for peer discovery, gossip for gradient sharing
3. **NEURO Economy:** Token system for rewards and payments

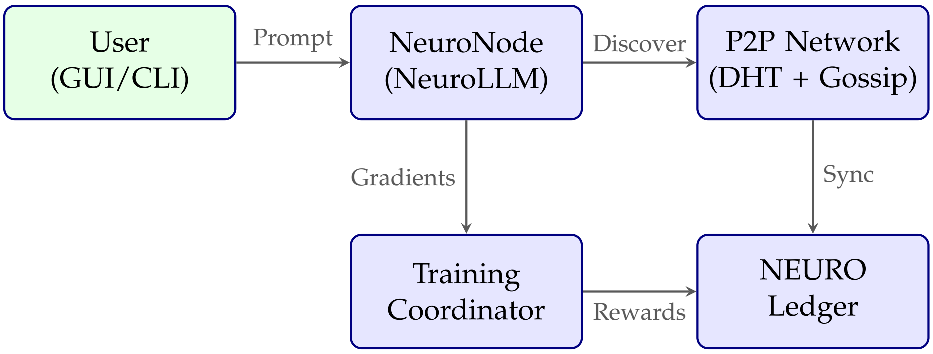


Figure 2: NeuroShard Architecture: Users interact with NeuroNode, which participates in P2P network for peer discovery and gradient gossip. Training Coordinator manages distributed training, and NEURO Ledger tracks token balances.

6.2 NeuroNode

The NeuroNode is the core component that:

- **Self-Benchmarks:** Measures compute speed, determines speed tier (T1–T5)
- **Detects Memory:** Automatically detects available system memory
- **Registers with Network:** Announces capabilities, receives layer assignments
- **Joins or Forms Quorum:** Finds compatible peers for pipeline training
- **Selects Contribution Mode:** Pipeline, async, data, verify, or inference
- **Earns NEURO:** Rewards based on work with scarcity bonuses

```python
def create_node(
    node_token: str,
    port: int = 8000,
) -> NeuroNode:
    # Create node (auto-detects memory and speed)
    node = NeuroNode(
        node_id=sha256(node_token).hexdigest(),
        port=port
    )

    # Start node:
    # 1. Benchmark and determine speed tier
    # 2. Register with network
    # 3. Get layer assignments based on memory
    # 4. Try to join or form a quorum
    # 5. Fall back to async mode if no quorum
    node.start()

    # Node is now ready:
    # - Speed tier determined (e.g., T2)
    # - Contribution mode selected (e.g., PIPELINE)
    # - Quorum joined or async mode active
    # - Earning NEURO for contributions

    return node
```

Listing 1: NeuroNode Initialization

6.3 P2P Network Layer

6.3.1 Peer Discovery

NeuroShard uses a hybrid discovery mechanism:

1. **Tracker Bootstrap:** Initial peer discovery via central tracker
2. **Kademlia DHT:** Decentralized discovery after bootstrap
3. **Gossip Protocol:** Peer list exchange for redundancy

6.3.2 Gradient Gossip

Gradients are shared via gossip:

1. Node computes local gradients
2. Compresses gradients (Top-K + INT8 + Zlib)
3. Gossips to $k = 3$ random peers
4. Receiving peers validate and aggregate

6.4 Decentralized Inference Protocol

Inference in NeuroShard flows through quorums, with users selecting based on latency, price, and reputation:

6.4.1 Inference Flow

1. **Discovery:** User queries DHT for available inference endpoints. Response includes quorum latency, price/token, reputation, and capacity.
2. **Request:** User sends prompt to chosen quorum's Initiator with payment locked in escrow.
3. **Execution:** For each token: Initiator → embed → Processors → Finisher → sample → return. Tokens streamed as generated.
4. **Payment:** On completion, payment released proportionally: Initiator 15%, Processors (by layers), Finisher 20%, Network burn 5%.
5. **Failure:** Timeout triggers partial refund, reputation penalty. Complete failure means full refund and reputation loss.

6.4.2 Dynamic Pricing

Each quorum sets price based on demand:

$$P = P_{\text{base}} \times (1 + u^2) \times M_{\text{speed}} \times M_{\text{rep}} \tag{16}$$

Where u = utilization, M_{speed} is speed tier multiplier (T1: 1.5×, T4: 0.8×), and M_{rep} is reputation factor. This creates a market where faster, more reliable quorums can charge premium prices.

Note: In the bootstrap phase, output quality will be low (the model is untrained). Quality improves as the network trains the model.

7 Proof of Neural Work (PoNW)

7.1 Overview

Proof of Neural Work (PoNW) is the foundational consensus mechanism of NeuroShard. Unlike traditional blockchain consensus:

- **Proof of Work (PoW):** Rewards arbitrary computation (SHA-256 hashing) that produces no useful output.
- **Proof of Stake (PoS):** Rewards capital lockup without requiring any computation.
- **Proof of Neural Work (PoNW):** Rewards *useful neural network computation* that directly improves NeuroLLM.

The key insight is that **training a neural network is inherently verifiable work**. A gradient computed on real data will, when aggregated with other gradients, reduce the model's loss. Fake or random gradients will not. This creates a natural proof mechanism.

7.2 PoNW as the Model Builder

PoNW is not just a reward mechanism - it is the **engine that builds NeuroLLM**. Every NEURO token minted represents a real contribution to the model's intelligence:

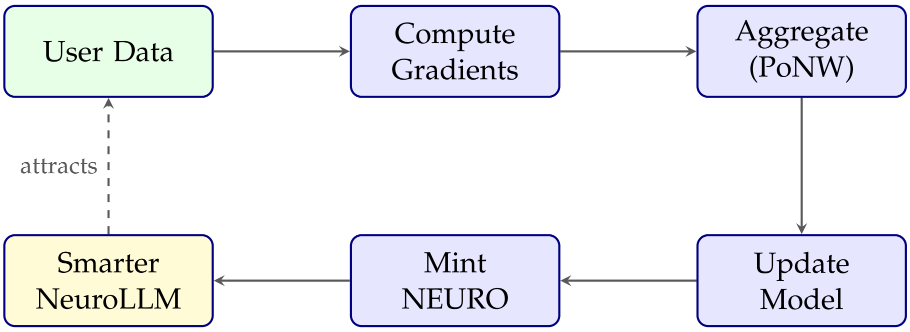


Figure 3: PoNW Cycle: User data → Gradients → Aggregation → Model Update → NEURO Rewards → Smarter Model → More Users

This creates a **virtuous cycle**:

1. Users contribute data and compute
2. PoNW validates contributions and mints NEURO
3. Model improves with each training round
4. Better model attracts more users
5. More users = more data = faster improvement

7.3 PoNW Components

PoNW validates three types of work:

1. **Proof of Inference:** Tokens processed during text generation. Each forward pass through NeuroLLM is counted.

2. **Proof of Training:** Gradient contributions to model improvement. The gradient's impact on loss reduction is measured.

3. **Proof of Data:** Training samples contributed. Data quality is assessed by the resulting gradient quality.

7.4 Optimistic Verification

NeuroShard uses **Optimistic Verification**: proofs are accepted immediately and can be challenged within a time window. This enables high throughput while maintaining economic security.

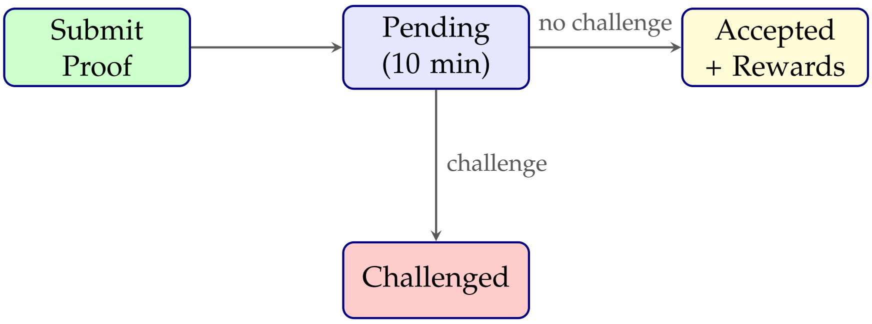


Figure 4: Optimistic Verification: Proofs accepted after challenge window expires.

7.4.1 Challenge Protocol

1. **Submission:** Node completes work, generates proof, submits to DHT. Status: "pending".

2. **Challenge Window:** 10-minute window for challenges.

3. **No Challenge (common):** Window expires, proof auto-accepted, rewards distributed.

4. **Challenge (rare):** Challenger stakes 10 NEURO, requests verification data.

5. **Resolution:** Challenger recomputes and compares. Fraud detected → prover slashed; false challenge → challenger loses stake.

7.4.2 Game Theory

Table 7: Optimistic Verification Outcomes

Outcome	**Prover**	**Challenger**
No challenge	Full reward	N/A
Valid proof	Reward + 50% stake	Loses stake
Fraud detected	Loses 2× stake	Gets prover stake
Prover timeout	Loses 2× stake	Gets prover stake

This makes cheating unprofitable: the expected value of submitting false proofs is negative due to the probability of detection and large slashing penalties.

7.4.3 Adaptive Verification Rate

The spot-check probability scales with network size:

$$P_{\text{verify}} = \begin{cases} 0 & \text{if } N < 20 \text{ (trust phase)} \\ 0.01 & \text{if } 20 \leq N < 100 \\ 0.03 & \text{if } 100 \leq N < 500 \\ 0.05 & \text{if } N \geq 500 \text{ (adversarial mode)} \end{cases} \tag{17}$$

7.5 The Training-Mining Equivalence

In NeuroShard, **training IS mining**. Traditional cryptocurrency mining wastes energy on arbitrary computation. NeuroShard redirects that energy into building collective intelligence:

Table 8: Mining Comparison

Aspect	**Bitcoin (PoW)**	**NeuroShard (PoNW)**
Work Type	SHA-256 hashing	Neural network training
Output	Block hash	Model improvement
Useful?	No (arbitrary)	Yes (builds AI)
Verification	Hash check	Loss reduction
Energy Use	Wasted	Productive

Every NEURO token represents real intelligence added to NeuroLLM. The token's value is backed by the model's capability.

7.6 Reward Function

The total NEURO reward for a Proof of Neural Work is calculated as:

$$R_{\text{total}} = \left(R_{\text{uptime}} + R_{\text{inference}} + R_{\text{training}} + R_{\text{data}}\right) \cdot M_{\text{role}} \cdot M_{\text{stake}} \cdot M_{\text{training}} \tag{18}$$

Where the base rewards are:

- $R_{\text{uptime}} = 0.0001 \cdot \frac{T_{\text{seconds}}}{60}$ NEURO (uptime, minimal)
- $R_{\text{inference}} = 0.1 \cdot \frac{T_{\text{tokens}}}{10^6} \cdot S_{\text{role}}$ NEURO (role-weighted inference)
- $R_{\text{training}} = 0.0005 \cdot B_{\text{batches}}$ NEURO (training batches, dominant)
- $R_{\text{data}} = 0.00001 \cdot D_{\text{samples}}$ NEURO (data serving)

And the multipliers are:

- $M_{\text{role}} = 1.0 + \text{InitiatorBonus} + \text{FinisherBonus} + \text{ScarcityBonus}$
- $M_{\text{stake}} = 1.0 + 0.1 \cdot \log_2\left(1 + \frac{S_{\text{stake}}}{1000}\right)$ (diminishing returns)
- $M_{\text{training}} = 1.1$ if actively training in a quorum, else 1.0

The role share S_{role} for inference is:

- Initiator (has embedding): $S_{\text{role}} = 0.15$
- Processor (middle layers): $S_{\text{role}} = 0.70$
- Finisher (has LM head): $S_{\text{role}} = 0.15$

7.7 Proof Generation

Nodes generate PoNW proofs every 60 seconds:

```python
def get_ponw_proof(self) -> Dict:
    proof = {
        "node_id": self.node_id,
        "timestamp": time.time(),
        "tokens_processed": self.total_tokens_processed,
        "training_rounds": self.total_training_rounds,
        "layers_held": len(self.my_layer_ids),
        "reward_multiplier": self.calculate_reward_multiplier(),
    }

    # Sign the proof
    proof_string = f"{self.node_id}:{proof['timestamp']}:{proof['layers_held']}"
    proof["signature"] = sha256(proof_string).hexdigest()

    return proof
```

Listing 2: PoNW Proof Generation

7.8 Proof Validation

Receiving nodes validate proofs by checking:

1. **Signature:** Cryptographically valid
2. **Timestamp:** Within 5-minute window (prevents replay)
3. **Non-duplicate:** Not already processed
4. **Plausibility:** Token counts are reasonable for the time period

7.9 Gradient Verification

The core innovation of PoNW is **gradient verification**. When a node submits a gradient contribution, it can be verified through:

7.9.1 Statistical Verification

Valid gradients have predictable statistical properties:

- **Distribution:** Real gradients follow a roughly Gaussian distribution
- **Magnitude:** Gradient norms fall within expected ranges based on batch size
- **Sparsity:** After Top-K compression, valid gradients have specific sparsity patterns

7.9.2 Loss-Based Verification

The ultimate test of a gradient's validity is whether it reduces loss:

$$\mathcal{L}(w - \eta \cdot g) < \mathcal{L}(w) \tag{19}$$

Where w is the current weights, η is the learning rate, and g is the submitted gradient. If this inequality holds on a validation set, the gradient is valid.

7.9.3 Cross-Validation

Multiple nodes computing gradients on similar data should produce similar gradients. Outliers are flagged:

$$\text{score}(g_i) = \frac{\|g_i - \bar{g}\|}{\sigma_g} \tag{20}$$

Gradients with score > 3 (more than 3 standard deviations from mean) are rejected.

7.10 Attack Prevention

PoNW prevents common attacks through a multi-layered security model:

- **Freeloading:** Cannot earn rewards without running the model and computing real gradients
- **Inflation:** Token counts are cross-validated against actual computation time
- **Sybil:** Staking requirement (1000 NEURO) makes fake nodes expensive
- **Gradient Poisoning:** Robust aggregation and statistical verification reject malicious gradients
- **Replay Attacks:** Timestamp windows and signature uniqueness prevent proof reuse

7.11 Cryptographic Security Model

Every PoNW proof is cryptographically secured using **ECDSA (secp256k1)**:

1. **BIP39 Wallet:** Users generate a 12-word mnemonic seed phrase (similar to MetaMask). The node token is derived as $\text{token} = \text{BIP39\_seed}(\text{mnemonic})[:32]$. Private keys are never stored—only the user controls their wallet.
2. **Key Derivation:** $\text{private\_key} = \text{SHA256}(\text{node\_token})$ (32 bytes)
3. **Public Key:** $\text{public\_key} = \text{ECDSA.derive}(\text{private\_key})$ (33 bytes compressed)
4. **Node Identity:** $\text{node\_id} = \text{SHA256}(\text{public\_key})[:32]$ (deterministic from public key)
5. **Proof Signature:** $\text{sig} = \text{ECDSA.sign}(\text{private\_key}, \text{canonical\_payload})$
6. **Trustless Verification:** Anyone can verify signatures using only the public key
7. **Timestamp Freshness:** Proofs must be < 5 minutes old
8. **Replay Prevention:** Each signature stored in `proof_history`, can never be reused
9. **Rate Limiting:** Max 120 proofs/hour, max 1M tokens/minute per node

Why ECDSA over HMAC? HMAC requires sharing the secret token to verify signatures. ECDSA allows **trustless verification** - any node can verify a proof's authenticity using only the signer's public key, without needing their secret token. This is essential for decentralized consensus.

The canonical payload format ensures deterministic verification:

```
payload = f"{node_id}:{proof_type}:{timestamp}:{nonce}:
           {uptime}:{tokens}:{batches}:{samples}:
           {layers}:{has_embed}:{has_head}"
```

Transparency Guarantee: There is NO admin backdoor. The ONLY way to get NEURO is to run a node, do real work, create a signed proof, and pass ALL verification checks. Even the project creators must run nodes and earn like everyone else.

8 Verification and Staking

NeuroShard combines computational work with economic stake to secure the network. Unlike pure Proof of Work (no accountability) or pure Proof of Stake (no work required), NeuroShard requires **both** real computation and economic commitment.

8.1 Work-First, Stake-Enhanced

The fundamental principle is that **work dominates**. You cannot earn significant rewards without performing real neural network computation. Stake provides a multiplier and enables verification rights, but the base rewards come from training and inference work.

8.2 Verifier Mode

Any node with stake can operate in **Verify Mode** to check proofs from other nodes:

- Monitor submitted PoNW proofs
- Randomly select proofs to verify (spot-check)
- Re-execute computation and compare results
- Challenge if mismatch found

Verifiers earn no base payment but receive **large bounties** when they detect fraud (they receive the slashed stake from cheaters).

8.3 Staking Requirements

Table 9: Stake Requirements by Role

Role	Purpose	Minimum Stake
Pipeline Member	Quorum participation	0 NEURO
Verifier	Challenge proofs	10 NEURO
Challenger	Submit fraud proofs	10 NEURO (at risk)

8.4 Slashing Conditions

Nodes lose stake when they:

- Submit fraudulent PoNW proofs (2× stake slashed)
- Make false fraud accusations (challenger stake lost)
- Submit poisoned gradients (detected by robust aggregation)
- Fail to respond to verification requests

8.5 Diminishing Stake Returns

To prevent the "rich-get-richer" problem inherent in pure PoS systems, NeuroShard implements **logarithmic diminishing returns** on stake multipliers:

$$M_{\text{stake}} = 1.0 + 0.1 \times \log_2 \left(1 + \frac{S_{\text{stake}}}{1000}\right) \tag{21}$$

Table 10: Diminishing Returns vs Linear Staking

Stake (NEURO)	Linear (Old)	Diminishing (New)	Reduction
1,000	1.10x	1.10x	0%
2,000	1.20x	1.16x	3%
5,000	1.50x	1.26x	16%
10,000	2.00x	1.35x	33%
50,000	6.00x	1.56x	74%
100,000	11.00x	1.66x	85%

Key Insight: Under the old linear system, a whale with 100,000 NEURO would earn 11x rewards. Under diminishing returns, they earn only 1.66x. This dramatically reduces wealth concentration while still rewarding commitment.

8.6 Verifier Economics

8.6.1 Verifier Rewards

Verifiers earn through fraud detection:

Table 11: Verifier Reward Sources

Source	Description	Rate
Fraud Detection	Successful challenge	Prover's stake
Stake Multiplier	For quorum roles	up to 1.66x
Finisher Bonus	LM Head in quorum	+30%
Training Bonus	If actively training	+10%

8.7 Why Work-Plus-Stake Works

The combined model solves the fundamental tension between decentralization and security:

1. **Work Dominates:** You cannot earn significant rewards without actual computation (training, inference). Stake only provides a multiplier, not base rewards.

2. **Stake Provides Accountability:** Nodes have economic skin in the game. Cheating costs real money.

3. **Diminishing Returns Prevent Monopolies:** Whales cannot dominate through pure capital. A node with 100× more stake only gets 1.5× more rewards.

4. **Low Barrier to Entry:** Only 10 NEURO required to become a verifier or challenger. This is earnable in a few days of active participation.

5. **Optimistic Flow:** Most proofs are never challenged, enabling high throughput. Cheaters are punished economically when caught.

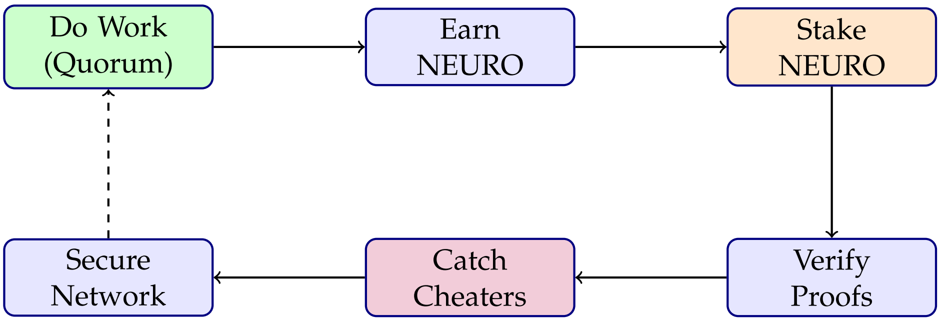


Figure 5: The Security Virtuous Cycle: Work → Earn → Stake → Verify → Secure → More Work

The Result: A system where nodes are both computationally invested (they run the model in quorums) and economically invested (they stake tokens). This dual commitment creates strong incentive alignment for honest behavior.

9 Robustness and Anti-Poisoning

9.1 The Model Poisoning Threat

In a decentralized training environment, malicious actors may attempt to "poison" the model by submitting gradients that degrade performance or inject hidden backdoors (triggers). NeuroShard employs a multi-layered defense strategy.

9.2 Robust Aggregation

To secure the global model against adversarial attacks ("poisoning"), the Training Coordinator employs **Robust Aggregation** algorithms rather than simple averaging.

- **Trimmed Mean:** For each parameter coordinate, the highest and lowest k% of values are discarded before averaging. This neutralizes outliers.
- **Coordinate-wise Median:** The median value of gradients is used, which theoretically tolerates up to 50% malicious participants.
- **Krum/Bulyan:** Advanced selection algorithms that identify the gradient vector closest to the spatial center of the distribution.

9.3 Adversarial Resistance Matrix

Table 12: Attack Types and Defenses

Attack	Detection	Defense	Penalty
Lazy compute	PoNW spot-checks	Merkle proofs	2× stake slash
Slow griefing	Latency monitoring	Speed tiers, quorum kick	Reputation loss
Gradient poison	Magnitude checks	Robust aggregation	Gradient rejected
Sybil attack	IP/hardware analysis	Min stake, attestation	All nodes slashed
Quorum collusion	Cross-quorum audit	Random verification	Quorum dissolved
Inference cheat	Multi-quorum verify	Response comparison	Blacklist
Vocab attack	Community review	Governance voting	Proposal rejected

9.4 Cross-Quorum Verification

To detect quorum-level collusion, the network performs random cross-quorum audits:

1. Select random proof from target quorum
2. Request verification data (activations, gradient samples)
3. Recompute forward pass and compare hashes
4. If mismatch detected, initiate fraud challenge

9.5 The Fraud Proof System

To address the "Lazy Verifier" problem and ensure economic security, NeuroShard implements a challenge-response system:

- **Evidence Submission:** If a node detects a statistically anomalous gradient (e.g., $> 5\sigma$ deviation), it generates a cryptographic **Fraud Proof**.
- **Verification:** The proof, containing the signed malicious gradient and statistical context, is broadcast to the network.
- **Slashing:** Upon verification, the malicious node's staked NEURO is slashed.

10 Governance and The NeuroDAO

NeuroLLM is not a static artifact; it is a living system that must evolve. Governance is handled by the **NeuroDAO**.

10.1 What NeuroDAO Does NOT Control

Unlike traditional systems, NeuroDAO does **not** vote on:

- **Model Size:** This is determined automatically by network capacity
- **Phase Transitions:** There are no phases - the model grows organically
- **Who Can Participate:** Anyone with compute can join

10.2 What NeuroDAO Controls

NEURO token holders vote on:

- **Reward Rates:** Adjusting the $R_{compute}$ and R_{data} multipliers to balance supply and demand.
- **Slashing Conditions:** Defining what constitutes malicious behavior and the penalties.
- **Protocol Parameters:** Minimum stake, gradient compression ratio, training round duration.

10.3 Vocabulary Governance

Unlike layer growth (which is automatic), vocabulary changes go through governance to prevent attacks:

Attack Scenario (without governance):

1. Malicious “merge coordinator” proposes merge: “api” + “_key” → “api_key”
2. This token gets heavily trained on API key patterns
3. Model learns to output API keys more readily
4. Attacker can extract sensitive data via inference

Solution: Governance-Based Vocabulary

1. Any merge must be publicly proposed (requires 100 NEURO stake)
2. 7-day review period for community inspection
3. 66% stake-weighted approval required
4. Suspicious merges get rejected
5. Proposer stake at risk if proposal is malicious

10.3.1 Vocabulary Proposal Process

Table 13: Vocabulary Governance Phases

Phase	Duration	Requirements
Proposal	Immediate	100+ NEURO stake
Discussion	7 days	Community review
Voting	7 days	30% quorum, 66% approval
Implementation	7 days	Grace period for nodes

10.4 Model Architecture Governance

The community decides on the “brain structure” of NeuroLLM:

- **Architecture Changes:** Proposals to switch from RoPE to ALiBi, or introduce Mixture-of-Experts layers, are submitted as Neuro Improvement Proposals (NIPs).
- **Layer Architecture:** Changes to attention mechanism, FFN structure, or normalization.
- **Training Parameters:** Changes to learning rates, sync intervals, or aggregation methods.

11 NEURO Token Economics

11.1 Token Utility

The NEURO token serves four functions:

1. **Reward Currency:** Earned by contributing compute and data
2. **Payment Currency:** Spent to access NeuroLLM inference
3. **Staking Security:** Staked to earn reward multipliers (with diminishing returns)
4. **Verification Eligibility:** Minimum 10 NEURO stake required to challenge proofs

11.2 Genesis Block and Transparency

NeuroShard's ledger is initialized with a **Genesis Block** that guarantees zero pre-mine:

- **Zero Pre-Mine:** The ledger starts with `total_minted = 0.0`
- **No Founder Allocation:** There are no pre-allocated tokens for founders or investors
- **No ICO:** All NEURO must be earned through verified Proof of Neural Work
- **Audit Trail:** Every token minted is traceable to a specific PoNW proof

The Genesis Block is recorded in the `proof_history` table with signature `GENESIS_BLOCK` and reward amount `0.0`. Anyone can verify this by querying:

```
SELECT * FROM global_stats WHERE id = 1;
-- Returns: total_minted=0.0, total_burned=0.0 (at genesis)
```

11.3 Deflationary Mechanics

To ensure long-term value accrual, NeuroShard implements a **Burn Mechanism**:

- **Fee Burn:** 5% of all spending (inference, transfers) is permanently burned
- **Fraud Slashing:** 50% of slashed stakes from malicious nodes are burned (50% to whistleblower)
- **Validator Slashing:** 100% of validator slashes are burned (validators slashed at 2x rate for voting against consensus)
- **Burn Address:** `BURN_0x0000...` - tokens sent here are irrecoverable

As network usage grows, the supply of NEURO decreases, rewarding long-term holders and contributors.

11.4 Earning NEURO

Users earn NEURO through a hierarchical reward structure that prioritizes **Training** as the core value:

Table 14: NEURO Earning Mechanisms (Reward Hierarchy)

Activity	**Description**	**Rate**	**Daily (Active)**
Training	Contributing gradients	0.0005 NEURO/batch	∼43 NEURO
Inference	Processing tokens	Market-based (pure supply/demand)	Variable
Data	Serving training shards	0.00001 NEURO/sample	Variable
Uptime	Running a node (idle)	0.0001 NEURO/min	∼0.14 NEURO

Note on Dynamic Pricing: Inference rewards use a **pure market** where price is determined by supply and demand. When the model is worthless (bootstrap), demand is zero and price approaches zero. As the model improves and attracts users, demand rises naturally and price increases. The market self-regulates: high prices attract more nodes to inference, while low prices push nodes back to training. Quality emerges from the demand signal itself.

11.4.1 Privacy-Preserving Distributed Inference Marketplace

NeuroShard implements a **request-response marketplace** that orchestrates distributed inference while preserving user privacy.

Architecture:

Inference in NeuroShard is inherently distributed across the pipeline:

1. **User** submits request to marketplace (metadata only - NO prompt!)
2. **Driver node** (Layer 0) receives encrypted prompt directly from user
3. **Worker nodes** (Layers 1-N) process activations (never see prompt)
4. **Validator node** (LM Head) generates output, returns to user

Privacy Guarantee: The prompt is NEVER stored in the marketplace. Users send encrypted prompts directly to their chosen driver node via a private channel. Worker nodes only process activations (meaningless vector representations), ensuring they cannot reconstruct the original prompt. This preserves privacy while enabling distributed computation.

Pure Market Pricing (No Artificial Caps):

$$P_{\text{market}} = P_{\text{base}} \times (1 + u)^2 \tag{22}$$

Where:

- $P_{\text{base}} = 0.0001$ NEURO (starting price)
- $u =$ demand rate/supply rate (utilization)
- NO minimum or maximum caps - market finds true value

Request-Response Matching:

To prevent timing attacks, prices are **locked at submission time**:

1. User submits request → Price locked at current market rate
2. Driver claims request → Starts distributed pipeline

3. All nodes submit proofs → Each rewarded by role
4. Request marked complete → All participants paid at locked price

This ensures users always pay the price they saw at submission, even if market price spikes during processing.

Distributed Reward Distribution:

Multiple nodes participate in each inference request. The locked price determines the total reward pool, distributed by role:

- **Driver (Layer 0):** 15% of pool (processes embedding, sees prompt)
- **Workers (Layers 1-N):** 70% of pool (divided among participants)
- **Validator (LM Head):** 15% of pool (generates output)

Each node submits its own signed PoNW proof linking to the same `request_id`. The ledger validates all proofs and distributes rewards accordingly.

Market Dynamics:

- **Worthless model:** No demand → Price ≈ 0 (nodes focus on training)
- **Improving model:** Rising demand → Price increases naturally
- **Viral demand:** Price spikes → Attracts more inference capacity
- **Equilibrium:** Market self-regulates where training profit $\gg$ inference profit

Key Properties:

1. **Privacy:** Workers never see prompts, only activations
2. **Decentralization:** Uses existing pipeline parallelism architecture
3. **Fair pricing:** Price locked at submission (no timing attacks)
4. **Quality signal:** Demand IS quality (no manual adjustments needed)
5. **Self-regulation:** Market finds equilibrium automatically

This marketplace enables NeuroShard to provide inference services while maintaining its core principles of privacy, decentralization, and fair compensation.

11.4.2 Multipliers and Bonuses

Table 15: Reward Multipliers

Multiplier	**Condition**	**Bonus**
Staking Bonus	Diminishing: $\log_2(1 + S/1000)$	up to +66%
Training Bonus	Actively training	+10% rewards
Driver Bonus	Holding embedding layer	+20% rewards
Validator Bonus	Holding LM head + 100 NEURO stake	+30% rewards
Layer Bonus	Per layer held	+5% (max 100%)
Validation Fee	Per proof validated	0.001 NEURO

Note: The Validator Bonus increased from 20% to 30% to compensate for the additional stake requirement and proof validation responsibilities.

11.5 Spending NEURO

Users spend NEURO for:

- **Inference:** 1 NEURO per 1M tokens generated
- **Priority:** Higher payment for faster processing
- **Transfers:** Send NEURO to other users

11.6 Economic Equilibrium

The system creates a sustainable economy:

- **Active Contributors:** Earn more than they spend
- **Passive Users:** Must purchase NEURO to use the service
- **Network Effect:** More users → more training → better model → more users

12 Security Considerations

12.1 Threat Model

NeuroShard assumes:

- Up to 49% of nodes may be malicious
- Attackers are economically rational
- Cryptographic primitives are secure

12.2 Data Privacy

Training data is protected through:

- **Local Processing:** Raw data never leaves the node
- **Differential Privacy:** Token-level noise injection
- **Gradient Compression:** Only sparse gradients are shared

12.3 Model Integrity

NeuroLLM integrity is maintained through:

- **Gradient Validation:** Anomalous gradients are rejected
- **Checkpoint Hashing:** Model state is cryptographically verified
- **Consensus:** Updates require majority agreement

12.4 Sybil Resistance

Sybil attacks are prevented through multiple layers:

- **Validator Stake:** 100 NEURO minimum to become a Hybrid Validator (verify proofs)
- **PoNW Requirement:** Must actually run the model to earn base rewards (stake only provides multiplier)
- **Diminishing Returns:** Creating 10 nodes with 100 NEURO each yields less than 1 node with 1000 NEURO
- **Slashing Risk:** Malicious validators lose their stake (2x penalty)
- **ECDSA Identity:** Node IDs derived from public keys prevent identity spoofing

13 Checkpoint System and State Sharding

13.1 Sharded Checkpoints

Unlike centralized systems where the full model state exists on a single disk, NeuroShard utilizes a **Sharded Checkpoint** system. The global model state is partitioned, with each node responsible for persisting only the weights of its assigned layers.

- **Distributed Storage:** No single node needs to store the entire multi-terabyte model file.
- **Redundancy:** Each shard is replicated across multiple nodes (based on layer assignment redundancy).
- **Merkle Integrity:** A root Merkle hash ensures the consistency of the global state across all shards without requiring full assembly.

13.2 Checkpoint Structure

Each node maintains a local shard:

```
shard = {
    "shard_id": "hash_of_node_id",
    "version": 1234,
    "layer_ids": [0, 1, 2, ...],     # Specific layers held by this node
    "state_dict": {...},             # Weights for ONLY these layers
    "optimizer_state": {...},        # Optimizer state for these layers
    "model_hash": "sha256...",       # Hash of this shard
    "timestamp": 1712345678
}
```

Listing 3: Sharded Checkpoint Structure

This approach allows the network to persist models that are significantly larger than the storage capacity of any individual participant.

13.3 Checkpoint Synchronization

New nodes joining the network synchronize via:

1. **Discovery:** Query DHT for nodes with latest checkpoint version

2. **Download:** Fetch checkpoint from multiple peers (parallel chunks)
3. **Verification:** Verify hash matches consensus
4. **Load:** Initialize NeuroLLM from checkpoint

13.4 Model Evolution Over Time

NeuroLLM evolves through two mechanisms:

13.4.1 Continuous Training

Every 60 seconds, a training round completes:

- Gradients from all contributing nodes are aggregated
- Model weights are updated
- New checkpoint is created
- NEURO rewards are distributed

13.4.2 Layer Growth System

Model depth grows automatically as network capacity increases. Growth is triggered when:

1. Total network capacity exceeds current model size by threshold (1.5× to 2.0×)
2. All existing layers have minimum replica coverage
3. At least 10,000 steps since last upgrade (stability check)

13.4.3 Layer Addition Sequence

When growth is triggered, the network executes a 5-phase sequence:

1. **Announcement:** Broadcast architecture upgrade, new `arch_version` assigned, 10-minute grace period begins.
2. **Preparation:** New layer entries created in DHT with status "pending". High scarcity bonus attracts nodes to claim new layers. Nodes download identity-initialized weights.
3. **Activation:** Requires each new layer to have $\geq$ MIN_REPLICAS holders and at least one fast holder (T1–T3). If not met, extend grace period (max 3 extensions).
4. **Quorum Reformation:** All existing quorums dissolve gracefully. Nodes reform quorums with new layer coverage.
5. **Warmup:** New layers train with 0.1× learning rate, gradually ramping to full LR over warmup steps.

13.4.4 Identity Initialization

New layers are initialized as near-identity functions to minimize disruption:

$$\text{output} \approx \text{input} \quad \text{(initially)} \tag{23}$$

This is achieved by scaling output projections to near-zero (×0.01), so the layer's residual contribution is negligible until trained.

13.5 The Intelligence Ledger

Every improvement to NeuroLLM is recorded in an **Intelligence Ledger**:

Table 16: Intelligence Ledger Entry

Field	**Description**
Round ID	Unique training round identifier
Timestamp	When the round completed
Contributors	List of node IDs that contributed
Batch Size	Total samples processed
Loss Before	Model loss before update
Loss After	Model loss after update
NEURO Minted	Total tokens distributed
Checkpoint Hash	Hash of resulting model state

This creates an immutable record of how NeuroLLM was built - every contribution, every improvement, every NEURO token can be traced back to specific training rounds.

14 Implementation

14.1 Technology Stack

- **PyTorch:** Neural network framework
- **FastAPI:** HTTP API server
- **gRPC:** Inter-node communication
- **Tkinter:** Desktop GUI
- **Kademlia:** DHT implementation
- **PyInstaller:** Cross-platform packaging

14.2 API Endpoints

Table 17: NeuroNode API Endpoints

Endpoint	**Method**	**Description**
/generate_text	POST	Generate text from prompt
/contribute_data	POST	Add training data
/train_step	POST	Trigger training step
/training_status	GET	Get training metrics
/api/stats	GET	Node statistics
/api/ponw	GET	Get PoNW proof
/api/model_info	GET	Model information

14.3 Desktop Application

The NeuroShard GUI provides:

- One-click node startup

- Real-time training metrics
- NEURO balance display
- Training enable/disable toggle
- Automatic updates

15 Scale Adaptation and Network Phases

The same protocol works from 1 node to millions, with parameters adapting automatically to network size. Behavior emerges from conditions rather than explicit phase transitions.

15.1 Network Phases

Table 18: Emergent Network Behavior by Scale

Nodes	Phase	Emergent Behavior
1	Genesis	Solo training, small model (only time solo allowed)
2–4	Micro	Single quorum, no cross-quorum sync
5–19	Small	1–3 quorums, speed matching begins, first layer growth
20–99	Medium	Multiple quorums at different tiers, async welcome
100–499	Growing	Adversarial resistance active, governance proposals
500+	Large	Regional clustering, specialized quorums, 100B+ params
5000+	Massive	Hierarchical coordination, frontier-scale model

15.2 Adaptive Parameters

Key protocol parameters scale with network size:

- **Min Quorum Size:** 1 ($N<5$), 2 ($N<20$), 3 ($N\geq20$)
- **Cohort Sync Interval:** 100 batches ($N<10$), 500 ($N<100$), 1000 ($N\geq100$)
- **Challenge Probability:** 0% ($N<20$), 1% ($N<100$), 5% ($N\geq500$)
- **Target Replicas:** 1 ($N<5$), 2 ($N<20$), 3 ($N\geq20$)

16 Vision and Roadmap

NeuroShard's evolution is driven by network growth, not predetermined schedules. The following milestones represent capabilities that emerge naturally as participation increases.

16.1 Foundation

The core infrastructure for decentralized AI:

- Dynamic layer pool for organic model scaling
- Gradient gossip protocol for decentralized training
- Proof of Neural Work consensus
- NEURO token economics
- Cross-platform node software (desktop, server)

16.2 Robust Training

Byzantine-tolerant training at scale:

- Trimmed mean and median gradient aggregation
- Fraud proof mechanism with slashing
- Anomaly detection for gradient poisoning
- 50x+ gradient compression
- **Network Target:** 100+ active nodes

16.3 Scale

Optimizing for global-scale throughput:

- **Mobile Optimization:** INT4 quantization and structural pruning for efficient edge device participation.
- **Hierarchical Aggregation:** Multi-tier parameter aggregation to minimize P2P latency in networks >10,000 nodes.
- **Developer SDK:** Comprehensive Python and JavaScript bindings for direct integration into third-party applications.
- **Network Target:** 1,000+ nodes, 1B+ parameters

16.4 Maturity

Global-scale collective intelligence:

- Multi-modal capabilities (vision, audio, code)
- Specialized domain fine-tuning
- NeuroDAO governance for protocol decisions
- Enterprise and institutional integrations
- **Network Target:** 10,000+ nodes, 100B+ parameters

Key Principle: These are not "phases" requiring coordinated upgrades. They are capabilities that emerge as the network grows. A single node joining the network immediately increases model capacity.

17 Distributed Architecture and Parallelism

17.1 The Dynamic Approach

In NeuroShard, distributed inference is not a "future feature" - it is the **default architecture**. Each node holds only the layers it can support, and inference requests flow through the network:

- Node A (2GB): Holds layers 0-18, including embedding
- Node B (4GB): Holds layers 19-56

- Node C (8GB): Holds layers 57-134
- Node D (4GB): Holds layers 135-172, including LM head

An inference request flows: Input → Node A → Node B → Node C → Node D → Output. Note that while the logical flow is linear, the physical routing is handled by the Swarm Network Layer (Section 16), where each step is dynamically routed to a redundant pool of peers to ensure fault tolerance.

17.2 Advanced Parallelism

NeuroShard implements advanced parallelism strategies to support massive models on consumer hardware.

17.2.1 Tensor Parallelism

To handle layers that are too large for a single device's memory (e.g., a 8192-dimension projection matrix), NeuroShard implements **Tensor Parallelism**. Large weight matrices are split across multiple nodes:

- **Column Parallelism:** Dividing linear layers by output features.
- **Row Parallelism:** Dividing linear layers by input features.

This allows a single logical layer to span multiple physical devices, enabling the training of models with hidden dimensions far exceeding individual VRAM limits.

17.2.2 Pipeline Parallelism

As the default mode of operation, Pipeline Parallelism distributes the model vertically. Input activations flow through the network of nodes, each processing a subset of layers, similar to a packet moving through a network router chain. This naturally balances compute load across the heterogeneous cluster.

17.3 Hybrid Parallelism

For very large models, NeuroShard combines both approaches:

1. **Inter-layer (Pipeline):** Different nodes hold different layers
2. **Intra-layer (Tensor):** Large layers split across multiple nodes

This enables models of **any size** to run across the network, each node contributing what it can.

17.4 Training at Scale

Distributed training uses:

- **Gradient Accumulation:** Nodes accumulate gradients locally before sharing
- **Ring All-Reduce:** Efficient gradient synchronization across nodes
- **Asynchronous SGD:** Nodes can train at different speeds
- **Layer-Local Gradients:** Each node only computes gradients for its assigned layers

17.5 The Key Insight

Traditional distributed training requires homogeneous hardware and careful coordination. NeuroShard's dynamic layer pool allows **heterogeneous participation**:

- A Raspberry Pi with 1GB RAM contributes 9 layers
- A gaming PC with 32GB RAM contributes 312 layers
- A cloud server with 256GB RAM contributes 2,500 layers

All participate in the same model, all earn NEURO proportional to contribution.

18 Quorum Resilience and Fault Tolerance

18.1 Design Philosophy: Throughput over Latency

The core insight driving NeuroShard's network design is: **"Compute is cheap; Bandwidth is expensive."** Residential internet connections (100Mbps-1Gbps) and node churn create fundamental challenges for distributed training. Rather than fighting these constraints, NeuroShard embraces them through quorum-based training with automatic failover.

Goal: 95% GPU utilization, even if wall-clock convergence is 10% slower per epoch. We win by being *unstoppable*, not by being the fastest.

18.2 Quorum Health and Replacement

Quorums monitor member health and can dynamically replace failing nodes:

- **Heartbeat Monitoring:** Members send heartbeats every 30 seconds
- **Degraded Status:** 2 missed heartbeats triggers replacement search
- **Graceful Replacement:** New node downloads layer state from peers
- **Session Renewal:** At 80% of session time, quorum evaluates and renews

18.3 Multipath Routing (Fault-Tolerant)

Traditional pipeline parallelism fails catastrophically when any node becomes unresponsive. NeuroShard uses **multipath routing** within quorums for fault tolerance.

18.3.1 Key Directive

"If Node A hangs, the packet must automatically flow to Node B without crashing the run."

18.3.2 Multipath Routing

For each layer range, the DHT maintains a list of K potential peers (typically $K = 3$), sorted by a weighted score:

$$\text{Score}_i = w_{\text{latency}} \cdot \frac{\text{Latency}_i}{500\text{ms}} + w_{\text{queue}} \cdot \frac{\text{QueueDepth}_i}{100} \tag{24}$$

Where $w_{\text{latency}} = 0.4$ and $w_{\text{queue}} = 0.6$. Lower scores indicate better candidates.

18.3.3 Probabilistic Send with Failover

When sending activations:

1. Send to primary peer (lowest score)
2. If no ACK within 200ms, immediately re-route to secondary peer
3. Continue until success or all K candidates exhausted

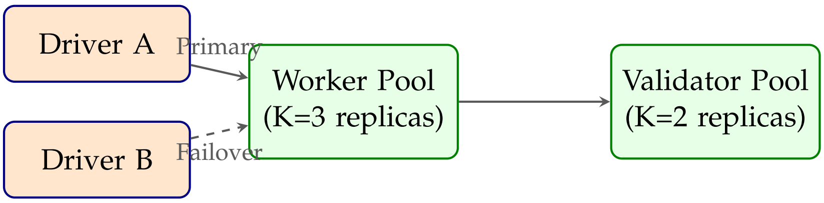
This provides automatic failover without coordinator intervention.

18.3.4 Capacity Bitmask Heartbeats

Every 5 seconds, nodes broadcast a lightweight **Capacity Bitmask** ($\sim$20 bytes):

- Available memory (MB)
- Current queue depth
- Layer range held
- GPU utilization (%)
- Status flags (training/inference/accepting work)

This allows peers to know who is ready to receive work, enabling intelligent routing decisions.



Automatic Failover
If primary hangs >200ms,
route to secondary

Figure 6: Swarm Routing: Multipath routing with automatic failover. Each layer range has K candidate peers.

18.4 Asynchronous Micro-Batch Engine

18.4.1 Key Directive

"Decouple Compute from Communicate. The forward pass must run 100 steps ahead of the backward pass if necessary."

The goal is to eliminate GPU starvation. The GPU should *never* wait for network packets.

18.4.2 Activation Buffering

Each node maintains two priority queues:

- **Inbound Buffer:** Incoming activations sorted by priority
- **Outbound Buffer:** Processed activations awaiting transmission

The GPU worker simply pops the next available item from Inbound, computes, and pushes to Outbound. Network I/O happens asynchronously.

$$\text{Buffer Fill Rate} = \frac{\text{Current Queue Size}}{\text{Max Queue Size}} \tag{25}$$

- Fill Rate < 0.1: Node is **Starved** (bad) — GPU idle
- Fill Rate > 0.9: Node is **Backpressured** (good) — GPU saturated

18.4.3 Interleaved 1F1B Schedule

Rather than strict synchronous execution, NeuroShard uses an **Interleaved 1F1B** (one-forward, one-backward) schedule:

```
Schedule (4 micro-batches):
F0 F1 F2 F3 B0 F4 B1 F5 B2 F6 B3 ...
```

Key insight: Start backward passes *before* all forwards complete. This overlaps backward compute with forward network latency, maximizing GPU utilization.

18.4.4 Dynamic Micro-Batch Sizing

Micro-batch size is dynamically tuned to find the "Goldilocks zone":

- Large enough to saturate the GPU
- Small enough to fit in network MTU (∼64KB target)

$$\text{MicroBatchSize} = \min\left(\frac{\text{AvailableMemory} \times 0.1}{\text{ActivationSize}}, \frac{\text{64KB}}{\text{SeqLen} \times \text{HiddenDim} \times 4}\right) \tag{26}$$

18.5 Extreme Gradient Accumulation (DiLoCo-Style)

18.5.1 Key Directive

"Accumulate locally for as long as possible. Sync only when statistically necessary."

Inspired by DiLoCo [14], NeuroShard slashes bandwidth by 90%+ through lazy syncing.

18.5.2 Local Inner Loop

Instead of syncing gradients every step, each node:

1. Saves initial weights w_0
2. Trains independently for N steps (e.g., $N = 500$) using local optimizer
3. Computes pseudo-gradient: $\Delta w = w_0 - w_N$
4. Only then triggers network sync

18.5.3 Outer Optimizer

The aggregated pseudo-gradients are applied using an outer optimizer with Nesterov momentum:

$$m_{t+1} = \beta \cdot m_t + \Delta w_{\text{aggregated}} \tag{27}$$

$$w_{t+1} = w_t + \eta_{\text{outer}} \cdot (\beta \cdot m_{t+1} + \Delta w_{\text{aggregated}}) \tag{28}$$

Where $\beta = 0.9$ (momentum) and $\eta_{\text{outer}} = 0.7$ (outer learning rate).

18.5.4 Bandwidth Reduction

Table 19: Bandwidth Comparison: Standard vs DiLoCo-Style

Metric	**Standard Gossip**	**DiLoCo (N=500)**
Syncs per 1000 steps	1000	2
Bandwidth per step	100%	0.2%
Latency tolerance	Low	High
Straggler tolerance	Low	High

18.5.5 Enhanced Verification

Since syncs are less frequent, bad gradients are more damaging. NeuroShard implements enhanced verification:

1. Compute gradient on local "trusted" micro-batch
2. Compare distribution to submitted gradient:
   - Cosine similarity > 0.5 (direction alignment)
   - Magnitude ratio within $10\times$ (scale check)
   - Variance ratio within $100\times$ (distribution check)
3. Reject gradients that fail any check

18.6 Speculative Checkpointing

To enable fast crash recovery, each node maintains "hot" snapshots:

- **Frequency:** Every 2 minutes (background thread)
- **Contents:** Model weights, optimizer state, DiLoCo buffers
- **Retention:** Last 5 snapshots
- **Announcement:** Published to DHT for neighbor discovery

On crash, a replacement node fetches the hot snapshot from a neighbor rather than restarting the epoch. Recovery time drops from "full restart" to < 30 seconds.

18.7 Summary: Resilience Through Design

Table 20: Swarm Architecture Benefits

Challenge	Solution	Result
Node failure	Multipath routing	<200ms failover
GPU starvation	Activation buffering	95% utilization
High latency	DiLoCo accumulation	500× fewer syncs
Crash recovery	Hot snapshots	<30s recovery
Network variability	Capacity heartbeats	Intelligent routing

19 Philosophy: Why Decentralized AI Matters

19.1 The Concentration Problem

As of 2024, the ability to create frontier AI is concentrated in fewer than 10 organizations worldwide. These organizations:

- Control what the AI can and cannot say
- Decide who gets access and at what price
- Collect and monetize user interactions
- Can shut down access at any time

This concentration of power over intelligence is unprecedented in human history. NeuroShard exists to provide an alternative.

19.2 The NeuroShard Principles

19.2.1 Principle 1: Intelligence Should Be a Public Good

Just as the internet democratized access to information, NeuroShard aims to democratize access to *intelligence*. NeuroLLM is not a black-box API owned by a corporation; it is a public utility created, maintained, and improved by the community that uses it.

19.2.2 Principle 2: Contributors Should Own What They Build

Every gradient computed, every data sample contributed, every hour of compute donated -these contributions are recorded on the Intelligence Ledger. NEURO tokens represent a programmable claim on the collective intelligence being built; they are a way to say, in cryptographic form, "I helped create this model."

19.2.3 Principle 3: No Single Point of Control

NeuroLLM cannot be censored, shut down, or silently modified by any single entity. As long as the network exists, the model exists. This is not just a systems design choice; it is a political statement about who should steer the trajectory of powerful AI: a narrow set of firms, or the people who actually use and train it.

19.2.4 Principle 4: Start Simple, Grow Together

NeuroLLM starts as a 125M parameter model that produces mostly gibberish. This is intentional. We refuse to bootstrap decentralization from a centralized model trained behind closed doors, with unknown data, incentives, and filters. Instead, we start from random weights, in the open, and let the network grow the model into something powerful over time.

19.3 The Long-Term Vision

In 10 years, we envision:

- **NeuroLLM as a global brain:** Billions of parameters, trained on the collective knowledge of millions of contributors
- **NEURO as AI currency:** The standard token for AI services worldwide
- **Specialized variants:** Community-governed fine-tunes for medicine, law, science, art
- **True AI democracy:** Major decisions about AI development made by token holders, not corporations

20 Conclusion

NeuroShard introduces a fundamentally new approach to artificial intelligence: a model created, trained, and owned by the community. NeuroLLM represents the first truly decentralized large language model - one that:

- **Belongs to everyone:** No corporation controls NeuroLLM. Every contributor is a co-owner.
- **Grows with participation:** More users = more compute = more data = better model. The network effect is the engine of intelligence.
- **Rewards contributors:** NEURO tokens for compute and data. Your contribution is recorded forever on the Intelligence Ledger.
- **Resists censorship:** No single point of control. No kill switch. No terms of service that can change overnight.
- **Builds intelligence from scratch:** We don't redistribute corporate models. We create our own, from random weights, through collective effort.

The model starts "dumb" and grows intelligent through collective effort. This is not a limitation - it is the foundation of true decentralization. Every participant who contributes to NeuroLLM's training is a co-creator of a new form of collective intelligence.

Proof of Neural Work ensures that every NEURO token represents real intelligence added to the model. Unlike cryptocurrencies that waste energy on arbitrary computation, NeuroShard channels that energy into building something useful: a public AI that belongs to humanity.

The vision of NeuroShard is a world where AI is a public good, created by humanity, for humanity. We are not asking permission from corporations to build the future. We are building it ourselves, one gradient at a time.

Join us. Contribute compute. Contribute data. Earn NEURO. Own the future of intelligence.

21 Acknowledgments

We thank the open-source community for the foundational technologies that made NeuroShard possible, including PyTorch, the broader P2P research community, and the pioneers of decentralized AI.

References

[1] B. McMahan et al., "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*, 2017.

[2] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.

[3] A. Borzunov et al., "Petals: Collaborative Inference and Fine-tuning of Large Models," in *ACL*, 2023.

[4] M. Ryabinin et al., "Hivemind: a Library for Decentralized Deep Learning," in *NeurIPS*, 2020.

[5] Bittensor Network. https://bittensor.com

[6] Gensyn Protocol. https://docs.gensyn.ai/litepaper

[7] B. Zhang and R. Sennrich, "Root Mean Square Layer Normalization," *arXiv:1910.07467*, 2019.

[8] J. Su et al., "RoFormer: Enhanced Transformer with Rotary Position Embedding," *arXiv:2104.09864*, 2021.

[9] J. Ainslie et al., "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints," *arXiv:2305.13245*, 2023.

[10] N. Shazeer, "GLU Variants Improve Transformer," *arXiv:2002.05202*, 2020.

[11] T. Brown et al., "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems*, 2020.

[12] J. Hoffmann et al., "Training Compute-Optimal Large Language Models," *arXiv:2203.15556*, 2022.

[13] J. Kaplan et al., "Scaling Laws for Neural Language Models," *arXiv:2001.08361*, 2020.

[14] A. Douillard et al., "DiLoCo: Distributed Low-Communication Training of Language Models," *arXiv:2311.08105*, 2023.

[15] M. Ryabinin et al., "SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient," *arXiv:2301.11913*, 2023.

A Code Examples

A.1 Starting a NeuroNode

```
from neuroshard import NeuroNode

# Create and start node (auto-benchmarks and detects memory)
node = NeuroNode(
    node_token="your-unique-token",
    port=8000
)
node.start()

# Node automatically:
# 1. Benchmarks speed tier (e.g., T2 = 10-50ms/layer)
# 2. Detects available memory (e.g., 4GB)
# 3. Registers with network, gets layer assignments
# 4. Tries to join or form a quorum
# 5. Falls back to async mode if no quorum available

print(f"Speed tier: {node.speed_tier}")
print(f"Mode: {node.contribution_mode}")
print(f"Layers: {node.layer_range}")
print(f"Quorum: {node.quorum_id or ’async mode’}")
```

Listing 4: Node Startup

A.2 Contributing Training Data

```
# Contribute text data for training
node.contribute_training_data(
    text="Your training text here...",
    apply_dp=True  # Apply differential privacy
)

# Check buffer status
stats = node.data_manager.get_stats()
print(f"Buffer: {stats[’buffer_size’]} samples")
```

Listing 5: Data Contribution

A.3 Generating Text

```
# Generate text (quality improves with training)
result = node.generate(
    prompt="Hello, world!",
    max_new_tokens=50,
    temperature=0.8
)

print(result)
# Early: "Hello, world! xkjf asd random gibberish..."
# Later: "Hello, world! How can I help you today?"
```

Listing 6: Text Generation

B Configuration Reference

B.1 NeuroLLM Dynamic Architecture Config

The architecture adapts to network capacity using scaling laws:

```python
def calculate_optimal_architecture(total_memory_mb):
    """
    Calculate optimal width x depth based on total network capacity.

    Examples:
      40GB network    -> 16L x 1024H  (~350M params)
      800GB network   -> 32L x 3072H  (~9.2B params)
      8TB network     -> 64L x 7168H  (~123B params)
    """
    # Apply empirical scaling laws
    # Width grows as memory^0.6, depth as memory^0.4
    params_budget = (total_memory_mb * 0.6) / 16

    hidden_dim = calculate_width(params_budget)
    num_layers = calculate_depth(params_budget)
    num_heads = hidden_dim // 64  # 64 dim per head
    num_kv_heads = num_heads // 3  # GQA ratio
    intermediate_dim = hidden_dim * 8 / 3  # SwiGLU

    return ModelArchitecture(
        hidden_dim=hidden_dim,
        num_layers=num_layers,
        num_heads=num_heads,
        num_kv_heads=num_kv_heads,
        intermediate_dim=intermediate_dim,
        vocab_size=32000,
        max_seq_len=2048
    )
```

Listing 7: Dynamic Architecture Calculation

B.2 Quorum and Training Configuration

```python
# Speed tier thresholds (ms per layer)
SPEED_TIER_THRESHOLDS = {
    "T1": 10,     # < 10ms (H100, A100)
    "T2": 50,     # 10-50ms (RTX 4090)
    "T3": 200,    # 50-200ms (RTX 3060)
    "T4": 1000,   # 200-1000ms (older GPU)
    "T5": float('inf')  # > 1000ms (Raspberry Pi)
}

# Quorum settings
BASE_SESSION_DURATION = 3600    # 1 hour
RENEWAL_CHECK_RATIO = 0.8       # Check at 80% of session
HEARTBEAT_INTERVAL = 30         # seconds
STALE_THRESHOLD = 4             # missed heartbeats

# Cross-quorum sync (DiLoCo)
SYNC_INTERVAL = 500             # batches between syncs
OUTER_LR = 0.7                  # outer learning rate

```

```
# Rewards
BASE_TRAINING_REWARD = 0.0005   # NEURO per batch per layer
INITIATOR_BONUS = 1.2           # +20% for embedding
FINISHER_BONUS = 1.3            # +30% for LM head

# Verification
CHALLENGE_WINDOW = 600          # 10 minutes
MIN_CHALLENGE_STAKE = 10        # NEURO
SLASH_MULTIPLIER = 2.0          # Lose 2x stake on fraud
```

Listing 8: Quorum and Training Parameters