Cooperative Multi-Device Neural Network Intelligence System Via Inference Partitioning in a C#/.NET Framework

Comprehensive Shared Inferencing Architecture Accomplished Through Multi-Device, Capability Aware Mesh Orchestration and Ledgered Accountability Mechanisms.

Malachi David Hooper*
malachihooper@dynsell.com
Dynsell Quantum Research Institute
Mountain View, California, USA

Abstract

Contemporary artificial intelligence systems remain predominantly dependent on centralized cloud infrastructures. These dependencies introduce substantial latency, cost overhead, and an inflexible reliance on persistent network connectivity. Similarly, individual edge devices—smartphones or embedded systems lack the memory and computational capacity to execute large-scale neural networks independently. This paper presents NightFrame, a cooperative multi-device neural network intelligence system that addresses these dual constraints through capability-aware mesh orchestration and tensor-level inference partitioning, implemented entirely within a C#/.NET ecosystem. The proposed architecture decomposes a centrally orchestrated neural network workload into granular shards, which are then distributed across a heterogeneous mesh of edge devices, where each node is dynamically assigned computational responsibilities proportional to its real-time hardware profile. Core C# system services, including and not limited to the OrchestratorService, ShardCoordinator, DroneRegistry, and LedgerService, are constructed as independently scalable micro-services communicating via gRPC and SignalR. This demonstrates a robust distributed AI orchestration that is achievable using standard enterprise tooling outside of traditional Python-centric machine learning environments. To establish trust and incentivize honest participation within the decentralized mesh, we introduce a theoretical cryptographic consensus protocol coupled with a tokenized credit ledger that records, verifies, and rewards node contributions within the form of a digital economy checking/balancing system. We also present companion user interfaces spanning web (Next.js), mobile (React Native/Expo), and desktop (WinForms/WPF) platforms for real-time mesh telemetry, prompt submission, and administrative oversight of the overall product architecture. Although the current implementation remains a research prototype, the networking foundation, cellular telemetry aggregation, and economic ledger infrastructures have been successfully demonstrated in this paper. This working paper establishes a practical blueprint for transitioning artificial intelligence processing from siloed, cloud-dependent paradigms to resilient, privacy-preserving, and horizontally scalable cooperative edge intelligence networks.

*Primary Author.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; *Client-server architectures*; *Client-server architectures*; • **Computing methodologies** → **Neural networks**; • **Security and privacy** → *Distributed systems security*; • **Networks** → Network management; Wireless access networks; *Network monitoring*; • **Human-centered computing** → **Web-based interaction**; **Interactive systems and tools**.

1 Conceptual Framework Introduction and Background

Current artificial intelligence systems rely on centralized cloud computing, where massive datasets and compute-intensive models reside in remote data centers [Avelar and Donovan 2023]. While powerful, these systems introduce significant latency, costs, and the critical dependency on persistent internet connections [Oppenheimer and Patterson 2002]. As the demand for real-time AI services, mobile edge applications, and autonomous systems increases, there is a growing need to shift computation closer to the data source. This shift, known as Edge Computing, aims to reduce latency and cost, as well as to increase privacy. However, individual edge devices (such as smartphones and "Internet of Things" sensors) often lack the memory and processing power required to run modern, large-scale neural networks independently [Hirsch et al. 2025].

1.1 Cooperative Edge Intelligence

To overcome these hardware limitations, the concept of Cooperative Edge Intelligence emerges. Rather than treating each device as an isolated island of compute resource, cooperative intelligence—such as that found in the demonstrated product "NightFrame"—is designed as a mesh of heterogeneous devices that pool their resources in a shared economy. In this model, a high-performance desktop, a mid-range smartphone, and a low-power IoT (Internet of Things) node collaborate to solve singular complex problems. The primary challenge in such a system is the awareness of each node's unique capabilities. This mesh network must understand the specific hardware profiles of every participating node to ensure that tasks are orchestrated amongst the most efficient available nodes without overloading ones that cannot carry out their delegated task portions without failure.

The NightFrame system utilizes a modular neural network framework designed for cooperative edge intelligence. While not pretrained, it is implemented as a customizable, distributed neural core. It is designed to be capable of continuous, autonomous learning and self-modification and knowledge ingestion. The network's key properties are designed for efficient and dynamic corpus management, distributed training across mesh nodes, and integration with web-based data sources. The implementation is partially complete, with secure updating mechanisms and production-grade inferencing pipelines. The limitations of the current implementation are defined further in this paper. The current quality of implementation is pragmatic and experimental. Implementations of the utilized neural networks system requires further refinement before production deployment.

1.2 Tensor-Level Sharding Methodology

A critical technical hurdle in distributed AI within a mesh network is the size of modern neural networks, which often exceed the memory capacity of any single edge device [Barwey et al. 2024]. The portioning methodology utilizes sharding at the tensor level. This is addressed in the NightFrame product model through the sharding of the utilized neural network. In this methodology, model sharding is the process of portioning a large neural network into smaller, manageable "shards". By distributing these shards across multiple nodes in a mesh, a complex inference task can be executed as a pipeline.

1.3 Contributions to the Technical Community

This paper offers several distinct contributions to the wider developer and research communities focused on decentralized artificial intelligence:

- **A Novel Mesh Orchestration Architecture:** We propose and detail the NightFrame architecture, demonstrating how a heterogeneous mix of edge devices (from high-performance desktop PCs to low-power IoT sensors) can successfully pool hardware resources to execute complex inference tasks without relying on centralized cloud infrastructures.
- **Capability-Aware Tensor-Level Sharding:** We introduce a pragmatic methodology for portioning modern, large-scale neural networks at the tensor level. By dynamically mapping shards to nodes based on real-time hardware capabilities and network locality, this approach effectively bypasses the memory and compute limits of individual edge devices.
- **Practical .NET Integration Strategies:** We outline a concrete implementation blueprint using the C#/.NET ecosystem. The leveraging of modern enterprise tooling such as ASP.NET Core, gRPC, and SignalR illustrates how developers can build highly decoupled, scalable, and modular AI micro-services outside of traditional Python-heavy ML environments.
- **Decentralized Trust and Incentive Economics:** We present a theoretical cryptographic consensus protocol and tokenized credit ledger designed to validate distributed compute results. This establishes a self-regulating economy that incentivizes honest participation and dynamically penalizes faulty or malicious nodes.

By addressing both the conceptual challenges and the engineering practicalities of distributed inference, this working paper aims to accelerate the transition from siloed AI processing to truly cooperative, edge-native intelligence networks.

2 User Interfaces

2.1 Orchestrator Web Admin Console (Next.js) (OWAC)

The web-based user interface element of the NightFrame product communicates with the *OrchestratorService* C# class that is later defined in the outline of the NightFrame system architecture. It utilizes SignalR and gRPC to provide mesh health information. The information provided in the Orchestrator Web Admin Console (OWAC) is limited to the following:

(1) Node enumeration
(2) Node roles
(3) Node statuses
(4) Real-time telemetry (cellular/RF maps)
(5) Prompt submissions
(6) Prompt results
(7) Ledger credit balances
(8) Audit logs

The OWAC is compiled as a lightweight application, which displays live data alongside the running *OrchestratorService* for concurrent, streamlined presentation of data.

Some admin pages of the web admin console are placeholders and are not prepared for production, however the overall architecture, purpose, and layout of the Next.js/React console is prepared for prototype evaluation.

2.2 Mobile Expo React-Native Console (mOWAC)

The mobile-first web-based console, built as a React-native/Expo application, is designed for on-the-go monitoring of scouts and to monitor the statuses of node ledgers. It displays the same information as the OWAC and is differentiated from the OWAC primarily by prototype readiness and fundamental user interface format. Most of the mOWAC prototyped screens are simulated, and the mOWAC real-time SignalR listening strategies are omitted to demonstrate fundamental mobile UI architecture.

2.3 Agent 3 Desktop UI

Through the utilization of WinForms and WPF, the Agent 3 Desktop UI is a full-featured desktop client that registers the host as a node, shows local hardware capabilities, lets the user submit prompts, displays viewable neural-mind chats, and watches credit accrual. It exposes manual controls for starting/stopping wider neural network training, toggles SSID hotspot capabilities, and provides a fully functional host experience for human users. All UI elements compile on Windows computers only, and the node can join the mesh as an administrative user. Agent 3 is currently engineered exclusively for the Windows ecosystem, leveraging native WinForms and WPF libraries for direct hardware polling and local node registration.

2.4 Node CLI (Headless)

3 System Architecture

3.1 C# Class Artifact Definitions and Capabilities

The following subsections describe the five core C# classes that constitute the NightFrame orchestration layer. Complete source code for each class is reproduced in Appendix A. The full repository, including all supporting project files, is archived as a zip file and accessible via the Digital Object Identifier (DOI) linked in Appendix B.

3.1.1 Central Coordination Hub. The C# class "OrchestratorService" starts the ASP.NET Core host and configures the initialization of gRPC, SignalR, and applicable REST endpoint wirings. This class also authenticates incoming node connections and routes control messages. It then publishes health-checks and system-wide configuration data to its own self-referential databases and to user-accessible control interfaces. Please see Listing 1 in Appendix A to view a copy of the code found in this class.

3.1.2 Model-Sharding Engine. The C# class "ShardCoordinator" receives a high-level inference or training request. It breaks the neural model into independent shards. These take the form of layer groups, token windows, and ONNX sub-graphs. The engine calculates optimal placement of sharded portion placements based on node capability metadata, as published by the central coordination C# class *OrchestratorService*. Please see Listing 2 in Appendix A to view a copy of the code found in this class.

3.1.3 Dynamic Node Catalog. The C# class "DroneRegistry" maintains a real-time registry of all participating edge devices. It stores hardware profiles, software versions, probationary states, and connectivity health. These profiles include heartbeats, mDNS/UDP discovery. The C# class provides lookup APIs for the *ShardCoordinator* and later-defined *LedgerService* C# classes to discover eligible nodes. Please see Listing 3 in Appendix A to view a copy of the code found in this class.

3.1.4 Contribution-Credit Ledger. The C# class "LedgerService" records cryptographic hashes of each shard's compute result together with node identities. It verifies result signatures with future-proof placeholders for production-grade verifications. It updates per-node credit balances, enabling a token-based incentive model for cooperative compute. The *LedgerService* also exposes audit-trail endpoints for administrators and applicable evaluation practices by either the product's own internal self-learning mechanisms or by external auditing efforts. Please see Listing 4 in Appendix A to view a copy of the code found in this class.

3.1.5 Cellular Telemetry Aggregator. The auxiliary C# class "CellularCoordinator" ingests RF-fingerprinting and cellular signal data from Scout Nodes. This information results in the generation of a global RF map that can be queried by the *ShardCoordinator* C# class for proximity-aware placement and inferencing. Please see Listing 5 in Appendix A to view a copy of the code found in this class.

3.2 Structure Cast in the .NET Framework

All services live under the *OrchestratorService* class and are registered with the ASP.NET Core dependency-injection container at startup. They communicate via gRPC (high-throughput bidirectional streams for shard payloads) and SignalR (low-latency telemetry to the WebConsole). The mesh's computational topology can be visualized as a directed acyclic graph:

```
User Prompt -> OrchestratorService
     |
     |-> ShardCoordinator ---> (multiple nodes)
     |          |
     |          |-> DroneRegistry (node lookup)
     |
     |-> LedgerService (credits & verification)
```

Each micro-service is deliberately isolated, allowing independent scaling (e.g. deploying the *ShardCoordinator* on a high throughput VM while the *LedgerService* runs on a low latency instance). This decomposition is the backbone of NightFrame's capability-aware, cooperative edge intelligence platform.

3.3 Node Role Enumerations

The NightFrame mesh differentiates participating devices through specific role enumerations, ensuring tasks are delegated according to each node's architectural capabilities.

3.3.1 Unknown Enumerated Node. Acts as a probationary placeholder for nodes that have joined the mesh but have not yet reported a specific hardware profile or functional role.

3.3.2 General Node. Executes generic, lightweight inference shards. These nodes handle standard computational workloads when no specialized hardware or high-throughput role is required.

3.3.3 Computational Node. Provides high-throughput CPU and GPU cycles specifically tailored for heavy neural-network sharding and intensive algorithmic processing.

3.3.4 Storage Node. Holds and serves model weights, operational caches, and persistent ledger databases for the wider NightFrame mesh network, acting as a distributed state repository.

3.3.5 Relay Node. Handles low-latency inter-node communication, mDNS/UDP peer discovery, and dynamic message routing across the decentralized topology.

3.3.6 Infiltration Node. Designated for privileged operations, such as dynamic SSID hotspot creation, local network bridging, or gateway management within restricted edge environments.

3.3.7 Scout Node. Deployed for environmental data gathering, this role collects RF fingerprints, cellular telemetry data, and other physical-layer metrics to feed the previously defined Cellular Telemetry Aggregator.

4 Partitioning Strategy and Sharding Optimization Methodology

The NightFrame system employs a capability-aware portioning strategy to divide computational graphs into shards while minimizing inter-node latency. This process balances two critical factors:

node heterogeneity (hardware specifications and power or network availability), and data locality—evaluating if the available node orchestration arrangements can carry tensors or activation maps efficiently between nodes and provide immediate results.

The *OrchestratorService* identifies dependencies between subgraphs (e.g., output of Layer A feeds into Layer B). Shards are portioned to group dependent layers together, reducing cross-node communication. Nodes are assigned shards based on their hardware profiles. For example, GPU-accelerated nodes handle matrix-heavy operations, while CPU-dominant nodes manage tokenization or lightweight layers. If a node's performance experiences degradation (e.g., due to resource contention), the *OrchestratorService* reassigns its shards to alternative nodes.

The *ShardCoordinator* analyzes the ONNX model to identify computational hotspots, such as large matrix-heavy operations and communication bottlenecks that require frequent tensor transfers or a multi-node orchestration requirement. The model is split into logical units, with each unit shard collaborating with the others to establish a computational pipeline, from tokenization shards to attention layer shards, or for example, output decoding shards. These conceptualizations demonstrate a typical orchestration of a multi-shard computational pipeline. Each shard is annotated with its compute requirements and input/output data dependencies. Nodes then bid for shards based on their capabilities. The *OrchestratorService* prioritizes assignments that minimize the distance (network hops) between shards that exchange data and ensures the overloading of nodes with shards exceeding their memory or compute capacity is avoided. Shards are executed sequentially or in parallel, with intermediate results streamed via gRPC. The *LedgerService* verifies results to ensure integrity.

Critical sub-graphs are assigned to nodes with low-latency interconnects. Token windows are processed in batches to reduce overhead, while overlapping computation and communication phases. If a node fails, its shards are reassigned to other nodes without disrupting the entire pipeline. This methodology ensures the mesh operates as a collaborative computational system with minimal opportunity for failure while optimizing for both speed and resource efficiency.

5 Theoretical Consensus Mechanism for Result Validation

To ensure the viability of NightFrame's trust economy, the *LedgerService* employs a decentralized, cryptographic consensus protocol designed to resolve conflicts between heterogeneous nodes. This mechanism operates without relying on centralized authorities and leverages the mesh's distributed nature to validate results collectively.

When a node completes a shard of computation, it generates a cryptographic signature of its result using a unique, node-specific identifier. This signature acts as evidence that computation was performed and is submitted to the *LedgerService* alongside the result. The signature is tied to the node's persistent identity and the specific shard being validated, ensuring accountability among node computational activity.

The *LedgerService* first verifies the signature's authenticity. If the signature is tampered, forged, or otherwise invalid, the result is immediately rejected, and the node's credit is penalized. Valid results are then checked against a reference model's output for the same shard.

5.1 Distributed Voting Process in the Event of Conflicting or Invalidated Shard Outputs

If multiple nodes submit conflicting results for the same shard, the *LedgerService* triggers a distributed voting process. Nodes are assigned weights based on their historical reliability. Nodes may submit zero-knowledge proofs or verifiable computation proofs to demonstrate their result's correctness without revealing internal details. The shared verification framework required for node proof submission and validation has not been developed in the research prototype of NightFrame, however the distributed voting process and invalidation of shard outputs has been implemented. A minimum number of nodes must agree on the correct result before it is accepted.

The system dynamically adjusts all of the node weights based on real-time performance and reliability. Nodes that consistently produce accurate results gain higher weights. Nodes with recent errors or low credit balances are deprioritized in future assignments. This self-regulating economy makes possible the incentivization of nodes to maintain high-quality contributions to preserve their credit balances.

With this model, zero-knowledge proofs or verifiable computation may require significant computational resources, which could burden edge nodes. A quorum-based system must balance security with efficiency, especially as the mesh grows. New nodes must be vetted before participating in consensus to prevent Sybil attacks (where a single entity controls multiple nodes). The integration and development of cryptographic primitives, dynamic weight management methodologies, and conflict-resolution logic is not currently developed within the *OrchestratorService* or *LedgerService* to maintain prototypical practicality.

6 Current Limitations and Future Work

While the theoretical framework and architecture of NightFrame introduce a robust vision for cooperative edge intelligence, the current codebase remains a research prototype with several pragmatic and brutally concrete limitations that define the boundary of current development:

- **Ecosystem Exclusivity:** The full-featured desktop client, Agent 3, is currently engineered exclusively for the Windows ecosystem. Its heavy reliance on native WinForms and WPF libraries for direct hardware polling and local node registration limits cross-platform desktop participation in the mesh.
- **Mocked Tensor-Level Sharding:** Currently, the *ShardCoordinator* does not dynamically partition live ONNX subgraphs or handle real-time attention layer decoding. Instead, the implementation mimics sharding by distributing simulated integer ranges and byte arrays across nodes.
- **Absence of Capability Bidding:** While the paper describes nodes dynamically bidding for shards based on their real-time memory and GPU capabilities, the implemented *ShardCoordinator* currently relies on a static, round-robin

assignment methodology across registered compute and general nodes.

- **Lack of True Cryptographic Consensus:** The *LedgerService* successfully records compute contributions and tracks credits/debits via a LiteDB instance. However, the theoretical distributed voting mechanism, zero-knowledge proofs, dynamic weight management, and cryptographic signature validations discussed in the consensus section are not implemented. The system currently credits nodes blindly upon task completion without cryptographically resolving conflicting shard outputs.
- **UI Placeholder Components:** While the Orchestrator Web Admin Console (OWAC) correctly tracks and displays telemetry via SignalR, several administrative pages remain incomplete placeholders, and the Mobile Expo Console (mOWAC) relies on simulated screens with omitted SignalR listening strategies.
- **Network Bottlenecks:** Despite capability-aware sharding designs, the system fundamentally relies on the availability of high-throughput *Computational Nodes*. In environments saturated entirely with low-power edge devices, complex neural networks may still face insurmountable latency and processing hurdles.

These limitations define the hard boundary between the project's current developments and its theoretical potential. Future work will prioritize abstracting the desktop client for cross-platform compatibility, integrating true ONNX runtime capabilities, optimizing proof-generation/verification algorithms for low-power nodes, and establishing the cryptographic verification and dynamic weight management frameworks required for a trustless environment.

7 Conclusion

The NightFrame project establishes a foundational blueprint for mesh-based decentralized artificial intelligence. By decomposing the heavy monolithic requirements of traditional neural networks into granular distributed operations handled by a heterogeneous mix of edge devices, NightFrame demonstrates the potential to bypass centralized cloud dependencies.

Through the combination of C#/.NET micro-services, including the *OrchestratorService*, *LedgerService*, and *ShardCoordinator*, this paper has proven the viability of managing a distributed edge network using standard enterprise tooling. Despite the concrete limitations of the current prototype in terms of live cryptographic consensus and true tensor sharding, the core networking, cellular aggregation, and economic ledger infrastructures have been successfully established. As edge devices become increasingly powerful, cooperative frameworks like NightFrame will be critical in enabling resilient, private, and horizontally scalable artificial intelligence at the edge.

References

Victor Avelar and Patrick Donovan. 2023. "The AI Disruption: Challenges and Guidance for Data Center Design." *Energy management Research Center*, 1, 1.

Shivam Barwey, Riccardo Balin, Bethany Lusch, Saumil Patel, Ramesh Balakrishnan, Pinaki Pal, Romit Maulik, and Venkatram Vishwanath. Nov. 2024. "Scalable and Consistent Graph Neural Networks for Distributed Mesh-based Data-driven Modeling." In: *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. (Nov. 2024), 1058–1070. doi:10.1109/SCW63240.2024.00146.

Matías Hirsch, Cristian Mateos, and Tim A. Majchrzak. Jan. 2025. "Exploring Smartphone-Based Edge AI Inferences Using Real Testbeds." *Sensors*, 25, 9, (Jan. 2025), 2875. doi:10.3390/s25092875.

D. Oppenheimer and D.A. Patterson. Sept. 2002. "Architecture and Dependability of Large-Scale Internet Services." *IEEE Internet Computing*, 6, 5, (Sept. 2002), 41–49. doi:10.1109/MIC.2002.1036037.

A C# Class Source Listings

The following listings present the complete source code for each of the five core C# classes discussed in Section 3 (System Architecture). All source files reside under `Orchestrator/Services/` in the NightFrame repository (DOI: 10.5281/zenodo.20636614).

A.1 OrchestratorService

```
using Grpc.Core;
using NIGHTFRAME.Orchestrator.Grpc;
using NIGHTFRAME.Orchestrator.Hubs;
using Microsoft.AspNetCore.SignalR;

namespace NIGHTFRAME.Orchestrator.Services;

public class OrchestratorService : NightframeOrchestrator.
      NightframeOrchestratorBase
{
    private readonly DroneRegistry _registry;
    private readonly LedgerService _ledger;
    private readonly ShardCoordinator _shardCoordinator;
    private readonly IHubContext<ConsoleHub> _consoleHub;
    private readonly ILogger<OrchestratorService> _logger;

    public OrchestratorService(
        DroneRegistry registry,
        LedgerService ledger,
        ShardCoordinator shardCoordinator,
        IHubContext<ConsoleHub> consoleHub,
        ILogger<OrchestratorService> logger)
    {
        _registry = registry;
        _ledger = ledger;
        _shardCoordinator = shardCoordinator;
        _consoleHub = consoleHub;
        _logger = logger;
    }

    /// <summary>
    /// Handles initial drone registration.
    /// </summary>
    public override Task<RegistrationResponse> Register(
        DroneManifest manifest, ServerCallContext context)
    {
        var address = context.Peer ?? "unknown";
        _logger.LogInformation(
            "Registration request from {NodeId} @ {Address}",
            manifest.NodeId, address);

        var response = _registry.EvaluateManifest(manifest,
              address);

        _ = _consoleHub.Clients.All.SendAsync("NodeRegistered",
              new
        {
            nodeId = manifest.NodeId,
            role = response.AssignedRole.ToString(),
            accepted = response.Accepted,
            reason = response.Reason
        });

        return Task.FromResult(response);
    }

    /// <summary>
    /// Bidirectional streaming for continuous drone communication
      .
    /// </summary>
    public override async Task Connect(
        IAsyncStreamReader<DroneMessage> requestStream,
        IServerStreamWriter<MothershipCommand> responseStream,
        ServerCallContext context)
    {
        string? nodeId = null;
        try
        {
```

```
            await foreach (var message in
                requestStream.ReadAllAsync(context.
                    CancellationToken))
            {
                switch (message.PayloadCase)
                {
                    case DroneMessage.PayloadOneofCase.Heartbeat:
                        nodeId = message.Heartbeat.NodeId;
                        _registry.RecordHeartbeat(
                            message.Heartbeat.NodeId,
                            message.Heartbeat.CpuLoad,
                            message.Heartbeat.RamUsedMb);
                        break;
                    case DroneMessage.PayloadOneofCase.TaskStatus:
                        await HandleTaskStatus(
                            message.TaskStatus, responseStream);
                        break;
                    case DroneMessage.PayloadOneofCase.
                        PeerDiscovery:
                        await HandlePeerDiscovery(
                            message.PeerDiscovery);
                        break;
                    case DroneMessage.PayloadOneofCase.LogEntry:
                        await HandleLogEntry(message.LogEntry);
                        break;
                }
                if (nodeId != null)
                    await TrySendCommandsAsync(
                        nodeId, responseStream,
                        context.CancellationToken);
            }
        }
        catch (Exception ex)
            when (ex is OperationCanceledException
                || ex.Message.Contains("cancelled"))
        {
            _logger.LogInformation(
                "Connection closed for {NodeId}", nodeId ?? "
                    unknown");
        }
    }

    /// <summary>
    /// Handles a drone requesting a compute shard.
    /// </summary>
    public override Task<ComputeShard> RequestShard(
        ShardRequest request, ServerCallContext context)
    {
        var shard = _shardCoordinator
            .GetNextShardForNode(request.NodeId);
        if (shard == null)
            return Task.FromResult(new ComputeShard
            {
                ShardId = "", ModelHash = "",
                StartLayer = -1, EndLayer = -1
            });
        return Task.FromResult(shard);
    }

    /// <summary>
    /// Handles compute result submission.
    /// </summary>
    public override Task<ResultAck> SubmitResult(
        ShardResult result, ServerCallContext context)
    {
        _shardCoordinator.RecordShardCompletion(
            result.ShardId, result.NodeId,
            result.OutputData.ToByteArray(),
            result.ComputeTimeMs);
        var balance = _ledger.GetBalance(result.NodeId);
        return Task.FromResult(new ResultAck
        {
            Accepted = true,
            CreditsEarned = LedgerService.CreditsPerComputeShard
        });
    }
}
```

Listing 1: OrchestratorService.cs — Central Coordination Hub (gRPC service implementing the Mothership-side mesh protocol).

A.2 ShardCoordinator

```
using System.Collections.Concurrent;
using NIGHTFRAME.Orchestrator.Grpc;

namespace NIGHTFRAME.Orchestrator.Services;

public class InferenceJob
{
    public required string JobId { get; init; }
    public required string PromptId { get; init; }
    public required byte[] InputData { get; init; }
    public required int TotalLayers { get; init; }
    public required int LayersPerShard { get; init; }
    public required DateTime CreatedAt { get; init; }
    public List<ShardAssignment> Shards { get; } = new();
    public ConcurrentDictionary<int, byte[]> CompletedShards
        { get; } = new();
    public byte[]? FinalResult { get; set; }
    public DateTime? CompletedAt { get; set; }
}

public class ShardAssignment
{
    public required string ShardId { get; init; }
    public required string NodeId { get; set; }
    public required int StartLayer { get; init; }
    public required int EndLayer { get; init; }
    public required string NextHopAddress { get; set; }
    public ShardStatus Status { get; set; } = ShardStatus.Pending;
    public DateTime? StartedAt { get; set; }
    public DateTime? CompletedAt { get; set; }
}

public enum ShardStatus
    { Pending, Assigned, Running, Completed, Failed, Retrying }

public class ShardCoordinator
{
    private readonly DroneRegistry _registry;
    private readonly LedgerService _ledger;
    private readonly ILogger<ShardCoordinator> _logger;
    private readonly ConcurrentDictionary<string, InferenceJob>
        _activeJobs = new();
    private readonly ConcurrentQueue<InferenceJob>
        _pendingJobs = new();

    private const int DefaultTotalLayers = 48;
    private const int DefaultLayersPerShard = 12;

    public ShardCoordinator(
        DroneRegistry registry,
        LedgerService ledger,
        ILogger<ShardCoordinator> logger)
    {
        _registry = registry;
        _ledger = ledger;
        _logger = logger;
    }

    /// <summary>
    /// Creates an inference job and shards it across
    /// available compute nodes (Pipeline Parallelism).
    /// </summary>
    public async Task<string> CreateInferenceJobAsync(
        string promptId, byte[] inputData)
    {
        var job = new InferenceJob
        {
            JobId = $"JOB_{DateTime.UtcNow:yyyyMMddHHmmss}"
                    + $"_{Guid.NewGuid():N}"[..20],
            PromptId = promptId,
            InputData = inputData,
            TotalLayers = DefaultTotalLayers,
            LayersPerShard = DefaultLayersPerShard,
            CreatedAt = DateTime.UtcNow
        };

        var computeNodes = _registry
            .GetNodesForRole(NodeRole.RoleCompute)
            .Concat(_registry
                .GetNodesForRole(NodeRole.RoleGeneral))
            .Where(n => !n.IsProbationary)
            .ToList();

        if (computeNodes.Count == 0)
        {
            _pendingJobs.Enqueue(job);
            return job.JobId;
        }
```

```

        int shardCount = (int)Math.Ceiling(
            (double)job.TotalLayers / job.LayersPerShard);

        for (int i = 0; i < shardCount; i++)
        {
            int startLayer = i * job.LayersPerShard;
            int endLayer = Math.Min(
                startLayer + job.LayersPerShard - 1,
                job.TotalLayers - 1);
            var node = computeNodes[i % computeNodes.Count];
            var nextNode = i < shardCount - 1
                ? computeNodes[(i + 1) % computeNodes.Count]
                : null;

            job.Shards.Add(new ShardAssignment
            {
                ShardId = $"{job.JobId}_SHARD_{i:D2}",
                NodeId = node.NodeId,
                StartLayer = startLayer,
                EndLayer = endLayer,
                NextHopAddress = nextNode?.Address
                    ?? "RETURN_TO_ORCHESTRATOR",
                Status = ShardStatus.Assigned
            });
        }

        _activeJobs[job.JobId] = job;
        return job.JobId;
    }

    /// <summary>
    /// Gets the next shard for a node to process.
    /// </summary>
    public ComputeShard? GetNextShardForNode(string nodeId)
    {
        foreach (var job in _activeJobs.Values)
        {
            var shard = job.Shards.FirstOrDefault(s =>
                s.NodeId == nodeId
                && s.Status == ShardStatus.Assigned);
            if (shard == null) continue;

            shard.Status = ShardStatus.Running;
            shard.StartedAt = DateTime.UtcNow;

            byte[] inputData;
            if (shard.StartLayer == 0)
                inputData = job.InputData;
            else
            {
                int prev = job.Shards.IndexOf(shard) - 1;
                if (!job.CompletedShards
                    .TryGetValue(prev, out var prevOut))
                {
                    shard.Status = ShardStatus.Assigned;
                    shard.StartedAt = null;
                    return null;
                }
                inputData = prevOut;
            }

            return new ComputeShard
            {
                ShardId = shard.ShardId,
                ModelHash = "model-v1.0-sha256",
                StartLayer = shard.StartLayer,
                EndLayer = shard.EndLayer,
                InputData = Google.Protobuf.ByteString
                    .CopyFrom(inputData),
                NextHopAddress = shard.NextHopAddress
            };
        }
        return null;
    }

    /// <summary>
    /// Records shard completion and handles pipeline flow.
    /// </summary>
    public void RecordShardCompletion(
        string shardId, string nodeId,
        byte[] outputData, long computeTimeMs)
    {
        foreach (var job in _activeJobs.Values)
        {
            var shard = job.Shards
                .FirstOrDefault(s => s.ShardId == shardId);
            if (shard == null) continue;

            shard.Status = ShardStatus.Completed;
            shard.CompletedAt = DateTime.UtcNow;
            int idx = job.Shards.IndexOf(shard);
            job.CompletedShards[idx] = outputData;

            _ledger.CreditComputeWork(
                nodeId, shardId, computeTimeMs);

            if (job.Shards.All(
                s => s.Status == ShardStatus.Completed))
            {
                int last = job.Shards.Count - 1;
                if (job.CompletedShards
                    .TryGetValue(last, out var final))
                {
                    job.FinalResult = final;
                    job.CompletedAt = DateTime.UtcNow;
                }
            }
            return;
        }
    }
}
```

Listing 2: ShardCoordinator.cs — Model-Sharding Engine (pipeline-parallel inference coordination across the drone mesh).

A.3 DroneRegistry

```
using System.Collections.Concurrent;
using NIGHTFRAME.Orchestrator.Grpc;

namespace NIGHTFRAME.Orchestrator.Services;

public class DroneNode
{
    public required string NodeId { get; init; }
    public required string Address { get; set; }
    public required string BinaryVersion { get; set; }
    public required NodeRole Role { get; set; }
    public required HardwareSpecs Specs { get; set; }
    public required byte[] PublicKey { get; init; }
    public required List<string> CachedModels { get; set; }

    public DateTime ConnectedAt { get; init; }
        = DateTime.UtcNow;
    public DateTime LastHeartbeat { get; set; }
        = DateTime.UtcNow;
    public bool IsOnline =>
        (DateTime.UtcNow - LastHeartbeat).TotalSeconds < 60;

    public double CurrentCpuLoad { get; set; }
    public long RamUsedMb { get; set; }
    public long TotalTasksCompleted { get; set; }
    public long TotalCreditsEarned { get; set; }

    public bool IsProbationary { get; set; } = true;
    public int ShadowTestsPassed { get; set; }
    public const int RequiredShadowTests = 10;

    public bool HasCellular { get; set; }
    public string? CellularTechnology { get; set; }
    public int? SignalStrength { get; set; }
    public bool HasGpu { get; set; }
    public string? GpuName { get; set; }
    public double? EstimatedFlops { get; set; }
    public double? Latitude { get; set; }
    public double? Longitude { get; set; }
    public double? LocationConfidence { get; set; }
}

public class DroneRegistry
{
    private readonly ConcurrentDictionary<string, DroneNode>
        _nodes = new();
    private readonly ILogger<DroneRegistry> _logger;

    private const long MinRamForCompute = 8192;   // 8 GB
    private const long MinDiskForStorage = 102400; // 100 GB
    private const long MinRamForRelay = 1024;      // 1 GB

    public int ActiveNodeCount =>
        _nodes.Values.Count(n => n.IsOnline);
```

```
    public int TotalNodeCount => _nodes.Count;

    public DroneRegistry(ILogger<DroneRegistry> logger)
    {
        _logger = logger;
    }

    /// <summary>
    /// Evaluates a drone manifest and decides admission.
    /// </summary>
    public RegistrationResponse EvaluateManifest(
        DroneManifest manifest, string address)
    {
        if (manifest.BinaryVersion
            != VersionManager.CurrentVersion)
        {
            return new RegistrationResponse
            {
                Accepted = false,
                Reason = "OUTDATED",
                AssignedRole = NodeRole.RoleUnknown,
                GlobalState = GetGlobalState()
            };
        }

        var requiredModel = GetCurrentModelHash();
        bool hasModel = manifest.CachedModels
            .Contains(requiredModel);
        var assignedRole = DetermineRole(
            manifest.Specs, hasModel);

        var node = new DroneNode
        {
            NodeId = manifest.NodeId,
            Address = address,
            BinaryVersion = manifest.BinaryVersion,
            Role = assignedRole,
            Specs = manifest.Specs,
            PublicKey = manifest.PublicKey.ToByteArray(),
            CachedModels = manifest.CachedModels.ToList(),
            IsProbationary = true,
            HasGpu = manifest.Specs.HasGpu,
            GpuName = manifest.Specs.GpuName
        };

        _nodes[manifest.NodeId] = node;

        return new RegistrationResponse
        {
            Accepted = true,
            AssignedRole = assignedRole,
            Reason = hasModel ? "ACTIVE" : "NEEDS_HYDRATION",
            GlobalState = GetGlobalState()
        };
    }

    private NodeRole DetermineRole(
        HardwareSpecs specs, bool hasModel)
    {
        if (specs.HasGpu && specs.RamMb >= MinRamForCompute)
            return NodeRole.RoleCompute;
        if (specs.DiskFreeMb >= MinDiskForStorage)
            return NodeRole.RoleStorage;
        if (specs.RamMb >= MinRamForCompute / 2)
            return NodeRole.RoleGeneral;
        return NodeRole.RoleRelay;
    }

    public void RecordHeartbeat(
        string nodeId, double cpuLoad, long ramUsedMb)
    {
        if (_nodes.TryGetValue(nodeId, out var node))
        {
            node.LastHeartbeat = DateTime.UtcNow;
            node.CurrentCpuLoad = cpuLoad;
            node.RamUsedMb = ramUsedMb;
        }
    }

    public IEnumerable<DroneNode> GetNodesForRole(
        NodeRole role)
    {
        return _nodes.Values
            .Where(n => n.IsOnline && n.Role == role)
            .OrderBy(n => n.CurrentCpuLoad);
    }

    public DroneNode? GetNode(string nodeId) =>
        _nodes.TryGetValue(nodeId, out var node)
            ? node : null;

    public double CalculateMeshHealth()
    {
        if (_nodes.IsEmpty) return 0;
        var online = _nodes.Values
            .Where(n => n.IsOnline).ToList();
        if (!online.Any()) return 0;
        var ratio = (double)online.Count / TotalNodeCount;
        var avg = online.Average(n => n.CurrentCpuLoad);
        return ratio * (1 - Math.Min(avg, 1.0));
    }
}
```

Listing 3: DroneRegistry.cs — Dynamic Node Catalog (tracking all connected drones capabilities and health status).

A.4 LedgerService

```
using System.Collections.Concurrent;
using LiteDB;

namespace NIGHTFRAME.Orchestrator.Services;

public enum TransactionType
{
    ComputeContribution,
    PromptSubmission,
    StorageContribution,
    NetworkRelay,
    PenaltyFraud,
    BonusUptime
}

public class LedgerEntry
{
    public ObjectId Id { get; set; }
        = ObjectId.NewObjectId();
    public required string NodeId { get; init; }
    public required TransactionType Type { get; init; }
    public required long Amount { get; init; }
    public required DateTime Timestamp { get; init; }
    public required string Reference { get; init; }
    public string? Details { get; init; }
}

public class NodeBalance
{
    public required string NodeId { get; init; }
    public long Credits { get; set; }
    public long Debits { get; set; }
    public long Balance => Credits - Debits;
    public DateTime LastActivity { get; set; }
}

public class LedgerService
{
    private readonly ILiteDatabase _db;
    private readonly ILiteCollection<LedgerEntry> _entries;
    private readonly ConcurrentDictionary<string, NodeBalance>
        _balanceCache = new();
    private readonly ILogger<LedgerService> _logger;

    public const long CreditsPerComputeShard = 100;
    public const long CreditsPerGBStored = 10;
    public const long CreditsPerRelayKB = 1;
    public const long CreditsPerUptimeHour = 5;
    public const long DebitPerPrompt = 50;
    public const long DebitPerTrainingRun = 500;

    public LedgerService(ILogger<LedgerService> logger)
    {
        _logger = logger;
        var dbPath = Path.Combine(
            Environment.GetFolderPath(
                Environment.SpecialFolder.ApplicationData),
            "NIGHTFRAME", "ledger.db");
        Directory.CreateDirectory(
            Path.GetDirectoryName(dbPath)!);
        _db = new LiteDatabase(dbPath);
        _entries = _db
            .GetCollection<LedgerEntry>("ledger");

```

```
        _entries.EnsureIndex(x => x.NodeId);
        _entries.EnsureIndex(x => x.Timestamp);
        _entries.EnsureIndex(x => x.Type);
        LoadBalanceCache();
    }

    /// <summary>
    /// Records a compute contribution credit.
    /// </summary>
    public void CreditComputeWork(
        string nodeId, string shardId,
        long computeTimeMs)
    {
        long baseCredits = CreditsPerComputeShard;
        if (computeTimeMs < 1000)
            baseCredits = (long)(baseCredits * 1.5);

        RecordEntry(new LedgerEntry
        {
            NodeId = nodeId,
            Type = TransactionType.ComputeContribution,
            Amount = baseCredits,
            Timestamp = DateTime.UtcNow,
            Reference = shardId,
            Details = $"Compute time: {computeTimeMs}ms"
        });
    }

    /// <summary>
    /// Debits account for a prompt submission.
    /// </summary>
    public bool DebitPromptSubmission(
        string nodeId, string promptId)
    {
        var balance = GetBalance(nodeId);
        if (balance.Balance < -10000) return false;

        RecordEntry(new LedgerEntry
        {
            NodeId = nodeId,
            Type = TransactionType.PromptSubmission,
            Amount = -DebitPerPrompt,
            Timestamp = DateTime.UtcNow,
            Reference = promptId
        });
        return true;
    }

    /// <summary>
    /// Applies a penalty for fraudulent compute results.
    /// </summary>
    public void ApplyFraudPenalty(
        string nodeId, string shardId, string reason)
    {
        RecordEntry(new LedgerEntry
        {
            NodeId = nodeId,
            Type = TransactionType.PenaltyFraud,
            Amount = -CreditsPerComputeShard * 10,
            Timestamp = DateTime.UtcNow,
            Reference = shardId,
            Details = reason
        });
    }

    public NodeBalance GetBalance(string nodeId)
    {
        return _balanceCache.GetOrAdd(nodeId,
            id => new NodeBalance
            {
                NodeId = id, Credits = 0, Debits = 0,
                LastActivity = DateTime.UtcNow
            });
    }

    private void RecordEntry(LedgerEntry entry)
    {
        _entries.Insert(entry);
        var balance = GetBalance(entry.NodeId);
        if (entry.Amount > 0)
            balance.Credits += entry.Amount;
        else
            balance.Debits += Math.Abs(entry.Amount);
        balance.LastActivity = entry.Timestamp;
    }

    private void LoadBalanceCache()
    {
        var entries = _entries.FindAll().ToList();
        foreach (var group in entries
            .GroupBy(e => e.NodeId))
        {
            var credits = group
                .Where(e => e.Amount > 0)
                .Sum(e => e.Amount);
            var debits = group
                .Where(e => e.Amount < 0)
                .Sum(e => Math.Abs(e.Amount));
            _balanceCache[group.Key] = new NodeBalance
            {
                NodeId = group.Key,
                Credits = credits,
                Debits = debits,
                LastActivity = group.Max(e => e.Timestamp)
            };
        }
    }
}
```

Listing 4: LedgerService.cs — Contribution-Credit Ledger (double-entry accounting for the internal economy).

A.5 CellularCoordinator

```
using Microsoft.AspNetCore.SignalR;
using NIGHTFRAME.Orchestrator.Hubs;
using System.Collections.Concurrent;

namespace NIGHTFRAME.Orchestrator.Services;

public class CellularCoordinator
{
    private readonly IHubContext<ConsoleHub, IConsoleClient>
        _hubContext;
    private readonly ILogger<CellularCoordinator> _logger;
    private readonly DroneRegistry _registry;

    private readonly ConcurrentDictionary<long, CellTowerRecord>
        _cellDatabase = new();
    private readonly ConcurrentDictionary<string,
         NodeCellularState>
        _nodeStates = new();
    private readonly ConcurrentQueue<CellMeasurementSample>
        _trainingData = new();
    private const int MaxTrainingDataSize = 100000;

    public CellularCoordinator(
        IHubContext<ConsoleHub, IConsoleClient> hubContext,
        DroneRegistry registry,
        ILogger<CellularCoordinator> logger)
    {
        _hubContext = hubContext;
        _registry = registry;
        _logger = logger;
    }

    /// <summary>
    /// Process a cellular status update from a drone.
    /// </summary>
    public async Task ProcessCellularUpdate(
        string nodeId, CellularStatusData data)
    {
        var state = _nodeStates.GetOrAdd(
            nodeId, _ => new NodeCellularState(nodeId));
        state.Update(data);

        if (data.CellId > 0)
            UpdateCellDatabase(data);

        if (data.Latitude.HasValue
            && data.Longitude.HasValue)
            AddTrainingSample(nodeId, data);

        await _hubContext.Clients.All.CellularUpdate(
            new CellularUpdateDto(
                NodeId: nodeId,
                Technology: data.Technology,
                Rsrp: data.Rsrp,
                Rsrq: data.Rsrq,
                Sinr: data.Sinr,
                CellId: data.CellId,
                NeighborCount: data.NeighborCount,
                Timestamp: DateTime.UtcNow.ToString("o")
```

```
            ));

        _registry.UpdateCellularInfo(
            nodeId, data.Technology, data.Rsrp);
    }

    private void UpdateCellDatabase(CellularStatusData data)
    {
        if (!data.Latitude.HasValue
            || !data.Longitude.HasValue) return;

        var record = _cellDatabase.GetOrAdd(
            data.CellId, _ => new CellTowerRecord
            {
                CellId = data.CellId,
                Mcc = data.Mcc, Mnc = data.Mnc,
                Lac = data.Lac,
                Technology = data.Technology,
                FirstSeen = DateTime.UtcNow
            });
        record.AddObservation(
            data.Latitude.Value,
            data.Longitude.Value, data.Rsrp);
    }

    public IEnumerable<CellTowerRecord> GetCellDatabase()
        => _cellDatabase.Values;

    public IEnumerable<CellTowerRecord> GetNearbyTowers(
        double lat, double lon, double radiusKm = 10)
    {
        return _cellDatabase.Values
            .Where(t => t.EstimatedLatitude.HasValue
                && t.EstimatedLongitude.HasValue)
            .Where(t => CalculateDistance(
                lat, lon,
                t.EstimatedLatitude!.Value,
                t.EstimatedLongitude!.Value)
                <= radiusKm)
            .OrderBy(t => CalculateDistance(
                lat, lon,
                t.EstimatedLatitude!.Value,
                t.EstimatedLongitude!.Value));
    }

    public CoverageStatistics GetCoverageStats()
    {
        var towers = _cellDatabase.Values.ToList();
        return new CoverageStatistics
        {
            TotalTowers = towers.Count,
            LteTowers = towers
                .Count(t => t.Technology == "LTE"),
            Nr5gTowers = towers
                .Count(t => t.Technology == "5G NR"),
            TotalObservations = towers
                .Sum(t => t.ObservationCount),
            TotalTrainingSamples = _trainingData.Count,
            ActiveNodes = _nodeStates.Values
                .Count(s => s.IsActive)
        };
    }

    private static double CalculateDistance(
        double lat1, double lon1,
        double lat2, double lon2)
    {
        const double R = 6371;
        var dLat = ToRadians(lat2 - lat1);
        var dLon = ToRadians(lon2 - lon1);
        var a = Math.Sin(dLat / 2)
                * Math.Sin(dLat / 2)
                + Math.Cos(ToRadians(lat1))
                * Math.Cos(ToRadians(lat2))
                * Math.Sin(dLon / 2)
                * Math.Sin(dLon / 2);
        var c = 2 * Math.Atan2(
            Math.Sqrt(a), Math.Sqrt(1 - a));
        return R * c;
    }

    private static double ToRadians(double degrees)
        => degrees * Math.PI / 180;
}
```

Listing 5: CellularCoordinator.cs — Cellular Telemetry Aggregator (crowd-sourced cell tower database and RF fingerprinting).

B Online Resources

The complete NightFrame source repository, including all C# class files listed in Appendix A, project solution files, protocol buffer definitions, the Next.js web console, the React Native/Expo mobile console, and the WinForms/WPF desktop client, is archived as a zip file and permanently accessible via the following Digital Object Identifier (DOI):

https://doi.org/10.5281/zenodo.20636614

Readers may download and extract the archived repository to replicate the system architecture, inspect the C# micro-service implementations, and evaluate the prototype interfaces described in this paper.

Received 11 June 2026; revised 11 June 2026; accepted 11 June 2026