🧠 MNEMO (Memory Arbitrage System) - Complete Build Specification

Version: 1.0
Date: February 26, 2026
Platform: Mist Inc. - Memory Arbitrage Division
Sister Platform: GP4U (GPU Marketplace)

Executive Summary

Mnemo is a VRAM/RAM-as-a-Service marketplace that captures idle memory from data centers, edge clusters, and consumer machines (Mist Nodes), then rents it to AI teams needing burst capacity. Unlike GPU marketplaces that rent whole machines, Mnemo arbitrages the **memory layer itself** - an unfilled market niche.

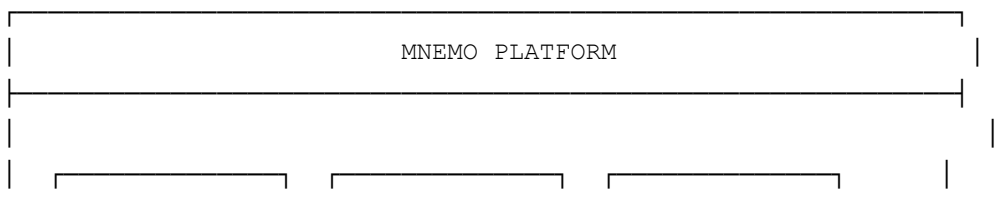
Core Innovation: Geographic clustering creates local memory meshes that outperform AWS/GCP through proximity (<1ms latency vs 10-30ms to cloud), enabling community-owned infrastructure with network effects.

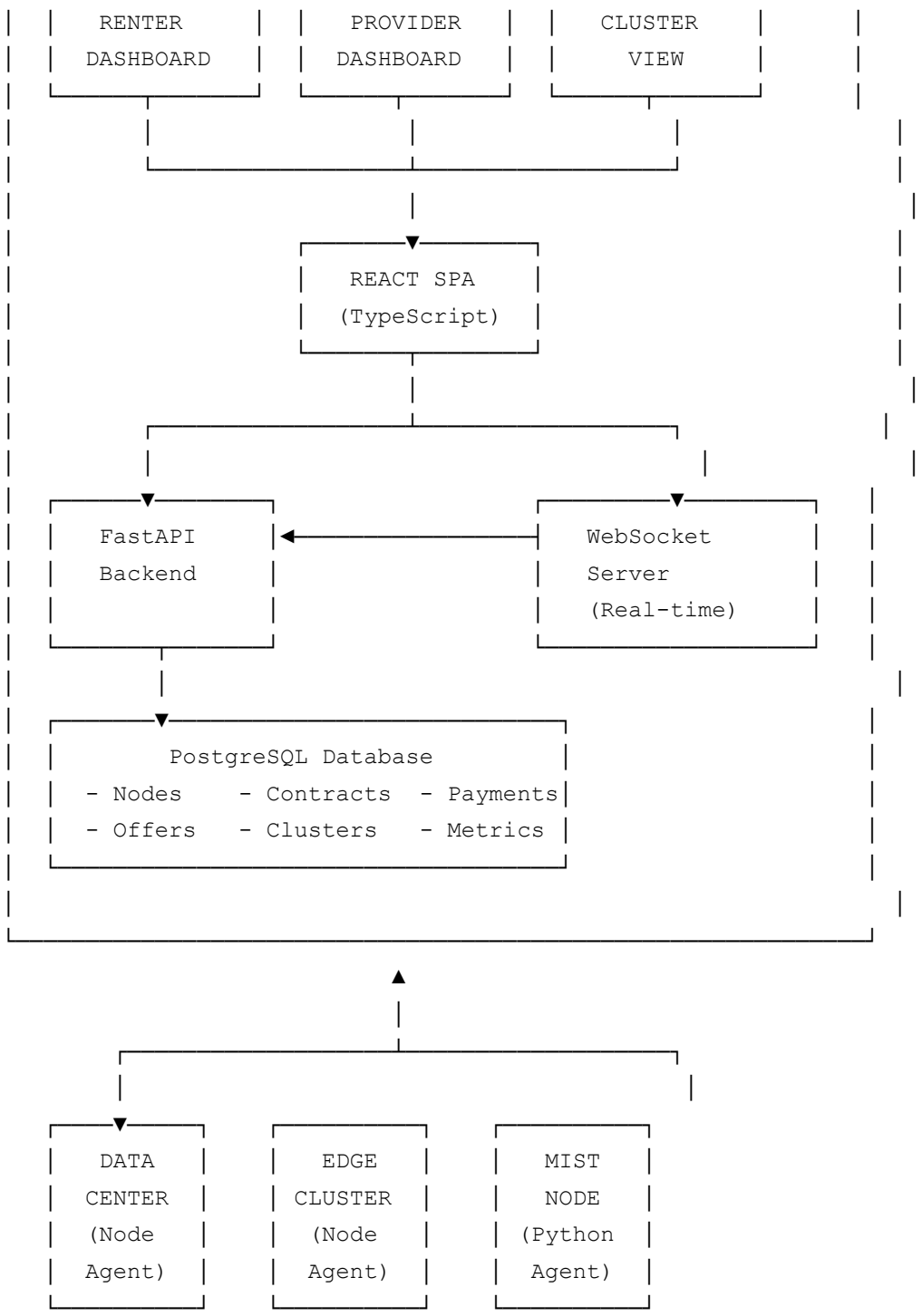
Market Opportunity:

- Every GPU server has idle VRAM between tasks (estimated 60-80% idle time)
- Consumer gaming PCs have 24GB VRAM, used ~12GB for 4 hours/day
- 240 GB-hours/day idle per machine = $2-5/day potential earnings
- 1,000 machines in Long Island = instant 12TB memory pool
- Zero serious competitors doing VRAM/RAM-as-a-Service at scale

---

🏗️ System Architecture Overview





📊 Database Schema (PostgreSQL)

Core Tables

```
-- Node providers (data centers, edge clusters, mist nodes)
CREATE TABLE nodes (
```

```
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    node_type VARCHAR(50) NOT NULL, -- 'datacenter', 'edge_cluster', 'mist_node'
    name VARCHAR(255) NOT NULL,
    owner_id UUID REFERENCES users(id),
    region VARCHAR(50) NOT NULL,
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),

    -- Capacity metrics
    total_ram_gb INTEGER NOT NULL,
    available_ram_gb INTEGER NOT NULL,
    total_vram_gb INTEGER NOT NULL,
    available_vram_gb INTEGER NOT NULL,

    -- Performance
    bandwidth_mbps INTEGER NOT NULL,
    base_latency_ms DECIMAL(8,3) NOT NULL,

    -- Reliability
    uptime_score DECIMAL(5,2) DEFAULT 99.0,
    reputation_score INTEGER DEFAULT 100,

    -- Pricing
    price_per_gb_sec DECIMAL(12,9) NOT NULL,

    -- Metadata
    metadata JSONB, -- {idle_schedule, gpu_model, cooling_type, etc}
    status VARCHAR(20) DEFAULT 'active', -- active, maintenance, offline
    last_heartbeat TIMESTAMP,

    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_nodes_region ON nodes(region);
CREATE INDEX idx_nodes_type ON nodes(node_type);
CREATE INDEX idx_nodes_status ON nodes(status);
CREATE INDEX idx_nodes_location ON nodes(latitude, longitude);

-- Memory offers/listings
CREATE TABLE offers (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    node_id UUID REFERENCES nodes(id) ON DELETE CASCADE,

    -- Capacity
    ram_gb INTEGER NOT NULL,
    vram_gb INTEGER NOT NULL,
```

```
    -- Performance guarantees
    min_bandwidth_mbps INTEGER,
    max_latency_ms DECIMAL(8,3),

    -- Pricing
    price_per_gb_sec DECIMAL(12,9) NOT NULL,
    egress_price_per_gb DECIMAL(8,5),

    -- Terms
    min_duration_sec INTEGER DEFAULT 60,
    max_duration_sec INTEGER DEFAULT 86400,
    replication_eligible BOOLEAN DEFAULT true,

    -- Status
    status VARCHAR(20) DEFAULT 'available', -- available, reserved, fulfilled
    expires_at TIMESTAMP,

    created_at TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_offers_node ON offers(node_id);
CREATE INDEX idx_offers_status ON offers(status);

-- Clients/renters
CREATE TABLE clients (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES users(id),
    org_name VARCHAR(255) NOT NULL,

    -- Location for proximity matching
    default_region VARCHAR(50),
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),

    -- Billing
    budget_monthly_usd DECIMAL(10,2),
    current_spend_usd DECIMAL(10,2) DEFAULT 0,

    -- Preferences
    prefer_local BOOLEAN DEFAULT true,
    max_latency_ms DECIMAL(8,3),
    min_reliability DECIMAL(5,2),

    created_at TIMESTAMP DEFAULT NOW()
);
```

```
-- Memory contracts (rental agreements)
CREATE TABLE contracts (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    client_id UUID REFERENCES clients(id),
    node_id UUID REFERENCES nodes(id),

    -- Allocation
    ram_gb INTEGER NOT NULL,
    vram_gb INTEGER NOT NULL,

    -- Timing
    duration_sec INTEGER NOT NULL,
    start_time TIMESTAMP NOT NULL,
    end_time TIMESTAMP NOT NULL,

    -- Pricing
    price_per_gb_sec DECIMAL(12,9) NOT NULL,
    total_cost_usd DECIMAL(10,4) NOT NULL,

    -- Performance
    actual_latency_ms DECIMAL(8,3),
    egress_gb DECIMAL(10,3) DEFAULT 0,

    -- Status
    status VARCHAR(20) DEFAULT 'pending', -- pending, active, completed, failed
    settlement_hash VARCHAR(128), -- for payment verification

    created_at TIMESTAMP DEFAULT NOW(),
    completed_at TIMESTAMP
);

CREATE INDEX idx_contracts_client ON contracts(client_id);
CREATE INDEX idx_contracts_node ON contracts(node_id);
CREATE INDEX idx_contracts_status ON contracts(status);
CREATE INDEX idx_contracts_time ON contracts(start_time, end_time);

-- Payment transactions
CREATE TABLE transactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    contract_id UUID REFERENCES contracts(id),

    amount_usd DECIMAL(10,4) NOT NULL,
    payment_method VARCHAR(50), -- stripe, crypto_eth, crypto_sol

    -- Stripe
    stripe_payment_id VARCHAR(255),
```

```
    -- Crypto
    blockchain VARCHAR(20),
    tx_hash VARCHAR(128),
    wallet_address VARCHAR(128),

    status VARCHAR(20) DEFAULT 'pending', -- pending, completed, failed, refunded

    created_at TIMESTAMP DEFAULT NOW(),
    settled_at TIMESTAMP
);

CREATE INDEX idx_transactions_contract ON transactions(contract_id);
CREATE INDEX idx_transactions_status ON transactions(status);

-- Node heartbeat/metrics (time-series data)
CREATE TABLE node_metrics (
    id BIGSERIAL PRIMARY KEY,
    node_id UUID REFERENCES nodes(id) ON DELETE CASCADE,

    -- Metrics at time of report
    available_ram_gb INTEGER NOT NULL,
    available_vram_gb INTEGER NOT NULL,
    cpu_usage_pct DECIMAL(5,2),
    gpu_usage_pct DECIMAL(5,2),
    temperature_c INTEGER,
    bandwidth_mbps INTEGER,

    -- Geolocation (can change for mobile nodes)
    latitude DECIMAL(9,6),
    longitude DECIMAL(9,6),

    timestamp TIMESTAMP DEFAULT NOW()
);

CREATE INDEX idx_metrics_node_time ON node_metrics(node_id, timestamp DESC);

-- Geographic clusters (computed view)
CREATE TABLE clusters (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    region VARCHAR(50) NOT NULL UNIQUE,

    -- Aggregated capacity
    total_nodes INTEGER DEFAULT 0,
    datacenter_nodes INTEGER DEFAULT 0,
    edge_nodes INTEGER DEFAULT 0,
    mist_nodes INTEGER DEFAULT 0,
```

```
    total_ram_gb INTEGER DEFAULT 0,
    available_ram_gb INTEGER DEFAULT 0,
    total_vram_gb INTEGER DEFAULT 0,
    available_vram_gb INTEGER DEFAULT 0,

    -- Pricing
    avg_price_per_gb_sec DECIMAL(12,9),

    -- Location
    center_latitude DECIMAL(9,6),
    center_longitude DECIMAL(9,6),

    last_updated TIMESTAMP DEFAULT NOW()
);

-- Users (shared with GP4U)
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(20) DEFAULT 'user', -- user, provider, admin

    -- Profile
    full_name VARCHAR(255),
    organization VARCHAR(255),

    -- Auth
    api_key VARCHAR(128) UNIQUE,
    jwt_secret VARCHAR(255),

    created_at TIMESTAMP DEFAULT NOW(),
    last_login TIMESTAMP
);
```

🔌 API Specification (FastAPI)

Authentication

All endpoints require JWT token or API key in header:

```
Authorization: Bearer <jwt_token>
X-API-Key: <api_key>
```

Core Endpoints

Node Management

```
# POST /api/nodes/register
# Register a new node (provider-side)
{
  "node_type": "mist_node",
  "name": "Alice_RTX4090",
  "region": "us-east-2",
  "latitude": 40.7282,
  "longitude": -73.7949,
  "total_ram_gb": 64,
  "total_vram_gb": 24,
  "bandwidth_mbps": 980,
  "base_latency_ms": 0.8,
  "price_per_gb_sec": 0.0000008,
  "metadata": {
    "gpu_model": "RTX 4090",
    "idle_schedule": "9am-5pm, 11pm-7am",
    "cooling": "air"
  }
}

# POST /api/nodes/{node_id}/heartbeat
# Node reports metrics every 60 seconds
{
  "available_ram_gb": 48,
  "available_vram_gb": 18,
  "cpu_usage_pct": 12.5,
  "gpu_usage_pct": 8.3,
  "temperature_c": 42,
  "latitude": 40.7282,
  "longitude": -73.7949
}

# GET /api/nodes
# List all nodes with filters
Query params: ?node_type=mist_node&region=us-east-2&min_ram=32&status=active

# GET /api/nodes/{node_id}
# Get detailed node info + earnings

# PUT /api/nodes/{node_id}
# Update node configuration
```

```
# DELETE /api/nodes/{node_id}
# Deactivate node
```

Marketplace

```
# GET /api/marketplace
# Browse available memory capacity
Query params:
  ?node_type=mist_node
  &region=us-east-2
  &min_ram_gb=32
  &min_vram_gb=12
  &max_price_per_gb_sec=0.000002
  &min_uptime_score=95.0
  &client_lat=40.7282
  &client_lng=-73.7949
  &max_distance_km=50

Response:
{
  "offers": [
    {
      "node_id": "...",
      "node_name": "Alice_RTX4090",
      "node_type": "mist_node",
      "region": "us-east-2",
      "available_ram_gb": 48,
      "available_vram_gb": 18,
      "price_per_gb_sec": 0.0000008,
      "uptime_score": 94.3,
      "distance_km": 3.2,
      "estimated_latency_ms": 0.8
    }
  ],
  "total_count": 47
}

# POST /api/marketplace/request
# Client requests memory with matching
{
  "ram_gb": 32,
  "vram_gb": 12,
  "duration_sec": 3600,
  "max_price_per_gb_sec": 0.000002,
  "prefer_local": true,
```

```
  "max_distance_km": 100,
  "min_uptime_score": 90.0
}

Response:
{
  "request_id": "...",
  "matches": [
    {
      "node_id": "...",
      "match_score": 287.5,
      "estimated_cost": 0.0305,
      "proximity_score": 98.7,
      "price_score": 45.2,
      "reliability_score": 47.1
    }
  ]
}
```

Contracts

```
# POST /api/contracts/create
# Create contract (allocate memory)
{
  "client_id": "...",
  "node_id": "...",
  "ram_gb": 32,
  "vram_gb": 12,
  "duration_sec": 3600
}

# GET /api/contracts
# List contracts (filtered by client/node/status)

# GET /api/contracts/{contract_id}
# Get contract details + real-time metrics

# POST /api/contracts/{contract_id}/settle
# Complete contract and trigger payment
{
  "actual_egress_gb": 2.4
}

# POST /api/contracts/{contract_id}/extend
# Extend active contract duration
```

Payments

```
# POST /api/payments/create
# Process payment for contract
{
  "contract_id": "...",
  "payment_method": "stripe",
  "stripe_token": "..."
}

# POST /api/payments/crypto
# Process crypto payment
{
  "contract_id": "...",
  "blockchain": "ethereum",
  "tx_hash": "0x...",
  "wallet_address": "0x..."
}

# GET /api/payments/invoice/{client_id}
# Generate invoice for client
```

Clusters

```
# GET /api/clusters
# Get geographic cluster statistics

Response:
{
  "clusters": [
    {
      "region": "us-east-2",
      "center": {"lat": 40.7282, "lng": -73.7949},
      "total_nodes": 847,
      "node_composition": {
        "datacenter": 2,
        "edge_cluster": 12,
        "mist_node": 833
      },
      "total_ram_gb": 54528,
      "available_ram_gb": 41232,
      "avg_price_per_gb_sec": 0.0000009
    }
  ]
}
```

```
# GET /api/clusters/{region}/nodes
# Get all nodes in a cluster

# GET /api/clusters/{region}/stats
# Detailed cluster analytics
```

Analytics

```
# GET /api/analytics/node/{node_id}/earnings
# Node earnings over time

# GET /api/analytics/client/{client_id}/spending
# Client spending breakdown

# GET /api/analytics/market/supply
# Global supply/demand metrics

# GET /api/analytics/market/pricing
# Price trends over time
```

🤖 Node Agent (Python)

File: `node_agent.py`

```
#!/usr/bin/env python3
"""
Mnemo Node Agent
Runs on provider machines to report memory availability
"""

import os
import time
import json
import requests
import psutil
import platform
from datetime import datetime

# Try to import GPU monitoring (optional)
try:
    import pynvml
```

```
    pynvml.nvmlInit()
    GPU_AVAILABLE = True
except:
    GPU_AVAILABLE = False

class MnemoNodeAgent:
    def __init__(self, config_path='node_config.json'):
        """Initialize node agent with configuration"""
        with open(config_path, 'r') as f:
            self.config = json.load(f)

        self.api_url = self.config['api_url']
        self.api_key = self.config['api_key']
        self.node_id = self.config.get('node_id')
        self.heartbeat_interval = self.config.get('heartbeat_interval', 60)

        # Register node if not already registered
        if not self.node_id:
            self.register_node()

    def register_node(self):
        """Register this machine as a node"""
        print("Registering node with Mnemo platform...")

        # Detect system info
        ram_info = psutil.virtual_memory()

        payload = {
            "node_type": self.config['node_type'],
            "name": self.config['name'],
            "region": self.config['region'],
            "latitude": self.config.get('latitude'),
            "longitude": self.config.get('longitude'),
            "total_ram_gb": int(ram_info.total / (1024**3)),
            "total_vram_gb": self.get_total_vram(),
            "bandwidth_mbps": self.config.get('bandwidth_mbps', 1000),
            "base_latency_ms": self.config.get('base_latency_ms', 1.0),
            "price_per_gb_sec": self.config['price_per_gb_sec'],
            "metadata": {
                "os": platform.system(),
                "cpu": platform.processor(),
                "gpu_model": self.get_gpu_model(),
                "idle_schedule": self.config.get('idle_schedule', 'always')
            }
        }

        response = requests.post(
```

```
            f"{self.api_url}/api/nodes/register",
            headers={"X-API-Key": self.api_key},
            json=payload
        )

        if response.status_code == 200:
            data = response.json()
            self.node_id = data['node_id']

            # Save node_id to config
            self.config['node_id'] = self.node_id
            with open('node_config.json', 'w') as f:
                json.dump(self.config, f, indent=2)

            print(f"✓ Node registered: {self.node_id}")
        else:
            print(f"✗ Registration failed: {response.text}")
            raise Exception("Node registration failed")

    def get_total_vram(self):
        """Get total VRAM in GB"""
        if not GPU_AVAILABLE:
            return 0

        try:
            device_count = pynvml.nvmlDeviceGetCount()
            total_vram = 0
            for i in range(device_count):
                handle = pynvml.nvmlDeviceGetHandleByIndex(i)
                info = pynvml.nvmlDeviceGetMemoryInfo(handle)
                total_vram += info.total
            return int(total_vram / (1024**3))
        except:
            return 0

    def get_gpu_model(self):
        """Get GPU model name"""
        if not GPU_AVAILABLE:
            return "No GPU"

        try:
            handle = pynvml.nvmlDeviceGetHandleByIndex(0)
            return pynvml.nvmlDeviceGetName(handle).decode('utf-8')
        except:
            return "Unknown GPU"

    def collect_metrics(self):
```

```
    """Collect current system metrics"""
    # RAM
    ram_info = psutil.virtual_memory()
    available_ram_gb = int(ram_info.available / (1024**3))

    # VRAM
    available_vram_gb = self.get_available_vram()

    # CPU
    cpu_usage = psutil.cpu_percent(interval=1)

    # GPU usage
    gpu_usage = self.get_gpu_usage()

    # Temperature
    temp = self.get_gpu_temperature()

    metrics = {
        "available_ram_gb": available_ram_gb,
        "available_vram_gb": available_vram_gb,
        "cpu_usage_pct": cpu_usage,
        "gpu_usage_pct": gpu_usage,
        "temperature_c": temp,
        "latitude": self.config.get('latitude'),
        "longitude": self.config.get('longitude')
    }

    return metrics

def get_available_vram(self):
    """Get available VRAM in GB"""
    if not GPU_AVAILABLE:
        return 0

    try:
        device_count = pynvml.nvmlDeviceGetCount()
        available_vram = 0
        for i in range(device_count):
            handle = pynvml.nvmlDeviceGetHandleByIndex(i)
            info = pynvml.nvmlDeviceGetMemoryInfo(handle)
            available_vram += info.free
        return int(available_vram / (1024**3))
    except:
        return 0

def get_gpu_usage(self):
    """Get GPU utilization percentage"""
```

```
    if not GPU_AVAILABLE:
        return 0

    try:
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        util = pynvml.nvmlDeviceGetUtilizationRates(handle)
        return util.gpu
    except:
        return 0

def get_gpu_temperature(self):
    """Get GPU temperature in Celsius"""
    if not GPU_AVAILABLE:
        return 0

    try:
        handle = pynvml.nvmlDeviceGetHandleByIndex(0)
        temp = pynvml.nvmlDeviceGetTemperature(handle, pynvml.NVML_TEMPERATUR
        return temp
    except:
        return 0

def send_heartbeat(self, metrics):
    """Send heartbeat to API"""
    try:
        response = requests.post(
            f"{self.api_url}/api/nodes/{self.node_id}/heartbeat",
            headers={"X-API-Key": self.api_key},
            json=metrics,
            timeout=10
        )

        if response.status_code == 200:
            print(f"✓ Heartbeat sent: RAM {metrics['available_ram_gb']}GB, VR
            return True
        else:
            print(f"✗ Heartbeat failed: {response.status_code}")
            return False
    except Exception as e:
        print(f"✗ Heartbeat error: {str(e)}")
        return False

def run(self):
    """Main loop - send heartbeats continuously"""
    print(f"Starting Mnemo Node Agent for {self.config['name']}")
    print(f"Node ID: {self.node_id}")
    print(f"Heartbeat interval: {self.heartbeat_interval}s")
```

```
        print("Press Ctrl+C to stop\n")

        try:
            while True:
                metrics = self.collect_metrics()
                self.send_heartbeat(metrics)
                time.sleep(self.heartbeat_interval)

        except KeyboardInterrupt:
            print("\n\nStopping node agent...")
            if GPU_AVAILABLE:
                pynvml.nvmlShutdown()
            print("Agent stopped.")

if __name__ == "__main__":
    agent = MnemoNodeAgent()
    agent.run()
```

Configuration file: `node_config.json`

```
{
  "api_url": "https://api.mnemo.io",
  "api_key": "your_api_key_here",
  "node_type": "mist_node",
  "name": "Alice_RTX4090",
  "region": "us-east-2",
  "latitude": 40.7282,
  "longitude": -73.7949,
  "price_per_gb_sec": 0.0000008,
  "bandwidth_mbps": 980,
  "base_latency_ms": 0.8,
  "idle_schedule": "9am-5pm, 11pm-7am",
  "heartbeat_interval": 60
}
```

🧱 Matching Algorithm (Core Logic)

```
# backend/services/matching.py

from typing import List, Dict
import math
from models import Node, Client
```

```
def calculate_distance(lat1: float, lng1: float, lat2: float, lng2: float) -> flo
    """Calculate distance between two points in km (Haversine formula)"""
    R = 6371  # Earth radius in km

    dlat = math.radians(lat2 - lat1)
    dlng = math.radians(lng2 - lng1)

    a = (math.sin(dlat/2) ** 2 +
         math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) *
         math.sin(dlng/2) ** 2)

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))

    return R * c

def match_nodes(
    client: Client,
    requirements: Dict,
    available_nodes: List[Node]
) -> List[Dict]:
    """
    Match client requirements to best nodes

    Scoring system (max ~300 points):
    - Proximity: 0-100 points (3x weight if prefer_local)
    - Price: 0-50 points
    - Reliability: 0-50 points (uptime_score * 0.5)
    - Capacity: 0-30 points (overcapacity bonus)
    - Node type bonus: +20 for mist_nodes (community preference)
    """

    ram_needed = requirements['ram_gb']
    vram_needed = requirements['vram_gb']
    max_price = requirements['max_price_per_gb_sec']
    prefer_local = requirements.get('prefer_local', True)
    max_distance = requirements.get('max_distance_km', 10000)

    # Filter nodes that meet hard requirements
    candidates = [
        node for node in available_nodes
        if (node.available_ram_gb >= ram_needed and
            node.available_vram_gb >= vram_needed and
            node.price_per_gb_sec <= max_price and
            node.status == 'active')
    ]

    matches = []
```

```
for node in candidates:
    score = 0

    # 1. PROXIMITY SCORE (most important if prefer_local)
    if client.latitude and client.longitude and node.latitude and node.longit
        distance = calculate_distance(
            client.latitude, client.longitude,
            node.latitude, node.longitude
        )

        if distance > max_distance:
            continue  # Skip nodes outside max distance

        # Score: 100 at 0km, decreasing to 0 at 1000km
        proximity_score = max(0, 100 - (distance / 10))

        # Apply 3x weight if prefer_local
        if prefer_local:
            score += proximity_score * 3
        else:
            score += proximity_score

    # 2. PRICE SCORE (lower price = higher score)
    price_ratio = 1 - (node.price_per_gb_sec / max_price)
    price_score = price_ratio * 50
    score += price_score

    # 3. RELIABILITY SCORE
    reliability_score = node.uptime_score * 0.5  # Max 50 points
    score += reliability_score

    # 4. CAPACITY SCORE (overcapacity = better failover)
    total_needed = ram_needed + vram_needed
    total_available = node.available_ram_gb + node.available_vram_gb
    capacity_ratio = total_available / total_needed
    capacity_score = min(30, capacity_ratio * 10)  # Max 30 points
    score += capacity_score

    # 5. NODE TYPE BONUS (encourage mist node usage)
    if node.node_type == 'mist_node':
        score += 20  # Community preference bonus

    # Calculate estimated cost
    duration_sec = requirements.get('duration_sec', 3600)
    total_gb = ram_needed + vram_needed
    estimated_cost = total_gb * duration_sec * node.price_per_gb_sec
```

```
            matches.append({
                'node_id': str(node.id),
                'node_name': node.name,
                'node_type': node.node_type,
                'region': node.region,
                'match_score': round(score, 2),
                'distance_km': round(distance, 2) if 'distance' in locals() else None
                'estimated_cost': round(estimated_cost, 4),
                'estimated_latency_ms': node.base_latency_ms,

                # Score breakdown (for transparency)
                'score_breakdown': {
                    'proximity': round(proximity_score if 'proximity_score' in locals
                    'price': round(price_score, 2),
                    'reliability': round(reliability_score, 2),
                    'capacity': round(capacity_score, 2)
                }
            })

        # Sort by match score (highest first)
        matches.sort(key=lambda x: x['match_score'], reverse=True)

        return matches
```

🎨 Frontend Structure (React + TypeScript)

```
mnemo-frontend/
├── src/
│   ├── components/
│   │   ├── layout/
│   │   │   ├── Header.tsx
│   │   │   ├── Sidebar.tsx
│   │   │   └── Footer.tsx
│   │   ├── marketplace/
│   │   │   ├── NodeCard.tsx
│   │   │   ├── FilterPanel.tsx
│   │   │   ├── RequestMemoryForm.tsx
│   │   │   └── MatchResultsModal.tsx
│   │   ├── provider/
│   │   │   ├── NodeDashboard.tsx
│   │   │   ├── EarningsChart.tsx
│   │   │   └── NodeRegistrationForm.tsx
│   │   ├── clusters/
```

```
│   │   │   ├── ClusterMap.tsx
│   │   │   ├── ClusterCard.tsx
│   │   │   └── NetworkEffectsViz.tsx
│   │   └── common/
│   │       ├── Button.tsx
│   │       ├── Card.tsx
│   │       └── Badge.tsx
│   ├── pages/
│   │   ├── RenterDashboard.tsx
│   │   ├── ProviderDashboard.tsx
│   │   ├── ClusterView.tsx
│   │   ├── ContractsList.tsx
│   │   └── Analytics.tsx
│   ├── services/
│   │   ├── api.ts
│   │   ├── websocket.ts
│   │   └── matching.ts
│   ├── hooks/
│   │   ├── useNodes.ts
│   │   ├── useContracts.ts
│   │   └── useRealtime.ts
│   ├── utils/
│   │   ├── proximity.ts
│   │   ├── pricing.ts
│   │   └── formatting.ts
│   ├── types/
│   │   └── index.ts
│   └── App.tsx
├── package.json
└── tsconfig.json
```

🔒 Security & Auth

JWT Authentication

```
# backend/auth/jwt_handler.py

from datetime import datetime, timedelta
from jose import jwt, JWTError
from passlib.context import CryptContext

SECRET_KEY = os.getenv("JWT_SECRET_KEY")
ALGORITHM = "HS256"
```

```
ACCESS_TOKEN_EXPIRE_MINUTES = 60 * 24  # 24 hours

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def create_access_token(data: dict):
    to_encode = data.copy()
    expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    to_encode.update({"exp": expire})
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        return None
```

Node Security

- **Sandboxing:** Docker containers for memory isolation
- **Hash Verification:** Memory state hashed before/after lease
- **Capacity Locks:** Prevent double-booking via database transactions
- **Rate Limiting:** Max 1 heartbeat per node per 30 seconds

💳 Payment Integration

Stripe (Traditional)

```
# backend/payments/stripe_handler.py

import stripe
stripe.api_key = os.getenv("STRIPE_SECRET_KEY")

async def process_payment(contract_id: str, amount_usd: float, token: str):
    """Process payment via Stripe"""
    try:
        charge = stripe.Charge.create(
            amount=int(amount_usd * 100),  # Convert to cents
            currency="usd",
            source=token,
            description=f"Mnemo Memory Contract {contract_id}"
```

```
        )

        return {
            "success": True,
            "charge_id": charge.id,
            "status": charge.status
        }
    except stripe.error.CardError as e:
        return {
            "success": False,
            "error": str(e)
        }
```

Crypto (Web3)

```
# backend/payments/crypto_handler.py

from web3 import Web3

async def verify_crypto_payment(tx_hash: str, expected_amount: float, blockchain:
    """Verify crypto payment on blockchain"""
    if blockchain == "ethereum":
        w3 = Web3(Web3.HTTPProvider(os.getenv("ETH_RPC_URL")))
        tx = w3.eth.get_transaction(tx_hash)

        # Verify amount and recipient
        if tx and tx['value'] >= Web3.toWei(expected_amount, 'ether'):
            return True

    return False
```

📈 Real-Time Updates (WebSocket)

```
# backend/websocket/server.py

from fastapi import WebSocket
import json

class ConnectionManager:
    def __init__(self):
        self.active_connections: Dict[str, WebSocket] = {}
```

```
    async def connect(self, websocket: WebSocket, client_id: str):
        await websocket.accept()
        self.active_connections[client_id] = websocket

    async def disconnect(self, client_id: str):
        if client_id in self.active_connections:
            del self.active_connections[client_id]

    async def send_update(self, client_id: str, message: dict):
        if client_id in self.active_connections:
            await self.active_connections[client_id].send_text(
                json.dumps(message)
            )

    async def broadcast_market_update(self, data: dict):
        """Broadcast to all connected clients"""
        for connection in self.active_connections.values():
            await connection.send_text(json.dumps({
                "type": "market_update",
                "data": data
            }))

manager = ConnectionManager()

@app.websocket("/ws/{client_id}")
async def websocket_endpoint(websocket: WebSocket, client_id: str):
    await manager.connect(websocket, client_id)
    try:
        while True:
            # Keep connection alive
            data = await websocket.receive_text()
    except WebSocketDisconnect:
        await manager.disconnect(client_id)
```

🎯 Phase-by-Phase Build Plan

Phase 1: Foundation (Weeks 1-2)

Goal: Core backend + node agent working

- Setup FastAPI backend structure
- Create PostgreSQL database with schema
- Implement JWT authentication

- Build node registration endpoint
- Create Python node agent (psutil + pynvml)
- Test node heartbeat system
- Basic marketplace GET endpoint

Deliverable: Node agent can register and report metrics to backend

---

Phase 2: Matching Engine (Weeks 3-4)

Goal: Core arbitrage logic operational

- Implement matching algorithm with proximity scoring
- Create contract creation endpoint
- Build contract settlement logic
- Add capacity allocation/deallocation
- Implement cluster aggregation (materialized view)
- Basic analytics endpoints

Deliverable: Complete request→match→contract flow working

Phase 3: Frontend MVP (Weeks 5-6)

Goal: Usable UI for renters and providers

- React app setup with routing
- Renter dashboard (marketplace + request form)
- Provider dashboard (node stats + earnings)
- Cluster view (geographic visualization)
- Real-time updates via WebSocket
- Responsive design

Deliverable: End-to-end demo: request memory → see matches → create contract

---

Phase 4: Payments (Week 7)

Goal: Money flows

- Stripe integration (cards)
- Crypto payment verification (ETH/SOL)
- Invoice generation
- Provider payout system
- Transaction history

Deliverable: Users can pay, providers get paid

Phase 5: Polish & Scale (Week 8+)

Goal: Production-ready

- Trust/reputation scoring
- Stake mechanism for providers
- Automatic failover on node failure
- Advanced analytics dashboard
- Mobile app (React Native)
- Docker containerization
- Load testing + optimization
- Documentation

Deliverable: Launch-ready platform

🔗 Integration with GP4U

Since Mnemo and GP4U are sister platforms under Mist Inc., they should share:

1. **User Authentication:** Single sign-on, shared JWT
2. **Database:** Same Postgres instance, different schemas
3. **Payment Processing:** Unified Stripe account

4. **UI Components:** Shared React component library
5. **Analytics:** Combined dashboard showing GPU + memory arbitrage

Cross-platform features:

- Rent GPU from GP4U + memory from Mnemo in one transaction
- Providers can offer both GPU compute and memory capacity
- Unified earnings dashboard

📦 Tech Stack Summary

Layer	Technology	Purpose
Backend	FastAPI + Python 3.11	REST API
Database	PostgreSQL 15	Data persistence
Real-time	WebSocket	Live updates
Frontend	React 18 + TypeScript	User interface
Node Agent	Python + psutil + pynvml	System monitoring
Auth	JWT + bcrypt	Security
Payments	Stripe + Web3	Money flow
Deployment	Docker + Kubernetes	Scalability
Monitoring	Prometheus + Grafana	Observability

🚀 Quick Start Commands

```
# Backend
cd backend
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install -r requirements.txt
uvicorn main:app --reload
```

```
# Frontend
cd frontend
npm install
npm start

# Node Agent
cd node-agent
pip install -r requirements.txt
python node_agent.py

# Database
docker-compose up -d postgres
python scripts/init_db.py
```

📋 Requirements Files

backend/requirements.txt

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
sqlalchemy==2.0.23
psycopg2-binary==2.9.9
pydantic==2.5.0
python-jose[cryptography]==3.3.0
passlib[bcrypt]==1.7.4
python-multipart==0.0.6
stripe==7.4.0
web3==6.11.3
websockets==12.0
requests==2.31.0
psutil==5.9.6
```

node-agent/requirements.txt

```
requests==2.31.0
psutil==5.9.6
nvidia-ml-py==12.535.133
```

frontend/package.json

```
{
  "dependencies": {
    "react": "^18.2.0",
```

```
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "typescript": "^5.3.2",
    "recharts": "^2.10.3",
    "axios": "^1.6.2",
    "socket.io-client": "^4.5.4",
    "@headlessui/react": "^1.7.17",
    "tailwindcss": "^3.3.5"
  }
}
```

🎓 Key Success Metrics

For Renters:

- Average latency improvement vs AWS (target: <1ms for local clusters)
- Cost savings (target: 40-60% cheaper than cloud providers)
- Availability (target: 99.5%+ uptime)

For Providers (Mist Nodes):

- Average monthly earnings per node (target: $50-150/month for consumer GPU)
- Utilization rate (target: 60%+ of idle capacity rented)
- Payout reliability (target: <24 hour settlement)

Platform Growth:

- Network effect: Each new node increases cluster value by 1.2-1.5x
- Geographic coverage: Clusters in 50+ cities within 12 months
- Market share: Capture 10% of memory rental market (currently dominated by AWS/GCP)

🌟 Competitive Advantages

1. **First-mover in VRAM/RAM arbitrage** - No serious competitors at scale
2. **Local network effects** - Can't be beat on latency for regional workloads
3. **Community ownership model** - People prefer supporting neighbors over AWS

4. **Idle capacity arbitrage** - Near-zero marginal cost for providers
5. **Cross-platform synergy** - Integration with GP4U creates unified compute marketplace

---

📞 Support & Resources

Documentation: https://docs.mnemo.io
API Reference: https://api.mnemo.io/docs
Community: https://discord.gg/mnemo
Status: https://status.mnemo.io

Built with ⚡ by Mist Inc.
Making the internet's physical layer programmable and efficient