COMPREHENSIVE STEP-BY-STEP GUIDE FOR ALL REQUESTED PHASES

I'll provide detailed, non-technical instructions for all requested phases, broken down into simple, actionable steps.

PART 1: REAL TEE INTEGRATION & REDUCING RELIANCE ON MOCKS

TEE Phase 1: Hardware & Library Setup (1-2 months)

Why This Phase Matters

The blockchain currently pretends to use secure hardware. We need real hardware to provide actual security guarantees.

Step 1: Choose Your Path (Week 1)

Option A: Cloud (EASIEST, recommended)

- Intel SGX: Use Azure Confidential Computing VMs
- AMD SEV-SNP: Use AWS EC2 instances
- Cost: $50-200 for testing

Option B: Local Hardware (Advanced)

- Intel SGX: Desktop with 11th-gen Intel CPU (i7-11700+)
- AMD SEV-SNP: Expensive server with AMD EPYC CPU

Recommendation: Start with Azure for Intel SGX

Step 2: Set Up Azure Account & SGX VM (Week 1)

Follow these exact steps:

1. **Create Azure account:**
   - **Go to: https://azure.microsoft.com/free/**
   - **Click "Start free"**
   - **Sign up with email and credit card (won't be charged unless you use beyond $200 credit)**
2. **Request SGX VM access (takes 1-7 days):**
   - **Google: "Azure confidential computing access request"**
   - **Click first Microsoft link**
   - **Fill form with:**
     - **Email: Your email**
     - **Company: "NERV Blockchain"**
     - **Use case: "Blockchain privacy with TEE"**
     - **Region: "East US" or "West Europe"**
   - **Submit and wait for approval email**
3. **Create SGX VM (once approved):**
   - **Login to https://portal.azure.com**
   - **Click "≡" menu → "Virtual machines" → "Create" → "Azure virtual machine"**
   - **Fill in:**
     - **Subscription: Your free trial**
     - **Resource group: Create new → "nerv-tee-test"**
     - **Virtual machine name: "nerv-sgx-vm1"**
     - **Region: Choose one with DCsv3 available (East US)**
     - **Image: Ubuntu 22.04 LTS**
     - **Size: Click "See all sizes" → Search "DC" → Choose "DC2s_v3"**
     - **Username: "nervadmin"**

        - Authentication type: SSH public key
        - Generate new key pair: Click "Create new"
        - Save the .pem file when prompted
    - Click "Review + create" → "Create"
    - Wait 5-10 minutes for deployment
4. Connect to VM:
    - In Azure portal → Go to your VM → Click "Connect" → "SSH"
    - Copy the "Example" command:
    - bash

    -
      ```
      ssh -i ~/Downloads/nerv-sgx-vm1_key.pem nervadmin@YOUR-VM-IP
      ```
    - On your computer:
        - Open Terminal (Mac/Linux) or PowerShell (Windows)
        - Run the command above

Step 3: Install SGX Libraries on VM (Week 1)

Run these commands in your SSH session:

bash
```
# 1. Update system

sudo apt update && sudo apt upgrade -y



# 2. Add Intel's repository

wget https://download.01.org/intel-sgx/sgx_repo/ubuntu/intel-sgx-deb.key

sudo apt-key add intel-sgx-deb.key

echo 'deb [arch=amd64] https://download.01.org/intel-sgx/sgx_repo/ubuntu jammy
main' | sudo tee /etc/apt/sources.list.d/intel-sgx.list
```

```
# 3. Install SGX libraries

sudo apt update

sudo apt install -y libsgx-dcap-ql libsgx-dcap-quote-verify
libsgx-dcap-default-qpl



# 4. Verify installation

dcap-quote-verify --version

# Should show version like "1.16.100.2"
```

Step 4: Set Up AMD SEV-SNP (Alternative, Week 2)

If you want AMD instead:

1. **Create AWS account:**
   - **Go to https://aws.amazon.com/free/**
   - **Click "Create a Free Account"**
2. **Launch SEV-SNP instance:**
   - **Go to EC2 Dashboard → "Launch instance"**
   - **Name: "nerv-sev-test"**
   - **AMI: Ubuntu 24.04**
   - **Instance type: Click "See all instance types" → Search "c7a" → Choose "c7a.xlarge"**
   - **Key pair: Create new → "nerv-sev-key"**
   - **Network settings: Allow SSH from anywhere**
   - **Advanced details: Under CPU options, check "AMD SEV-SNP"**
   - **Click "Launch instance"**
3. **Install SEV tools:**

4.
   ```
   bash
   ```

```
sudo apt update

sudo apt install -y linux-modules-extra-aws

git clone https://github.com/AMDESE/sev-guest

cd sev-guest

make
```

5. `sudo cp snpguest /usr/bin/`

Step 5: Integrate Real SGX into Code (Week 2-3)

On your local computer (NOT the VM):

1. **Open project in VS Code:**
   - **Install VS Code: https://code.visualstudio.com/**
   - **Open your NERV project folder**
2. **Add TEE real mode feature:**
   - **Open file: `Cargo.toml`**
   - **Find `[features]` section or add it:**
   - `toml`

```
[features]
```

   - `tee_real = []`
3. Modify SGX code:
   - Open file: `src/privacy/tee/sgx.rs`
   - **Find function that generates attestation (search for "mock" or "simulation")**
   - **Replace with:**
   - `rust`

```
#[cfg(feature = "tee_real")]

fn generate_quote(&self) -> Result<Vec<u8>> {

    // Real quote generation using DCAP

    let mut quote = vec![0u8; 2048];

    // This is placeholder - you'll add real DCAP calls

    Ok(quote)

}



#[cfg(not(feature = "tee_real"))]

fn generate_quote(&self) -> Result<Vec<u8>> {

    // Mock quote for testing

    Ok(vec![0u8; 128])
```

    - }

4. Build with real mode:
5. `bash`
6. `cargo build --features tee_real`
7. Copy code to VM:
8. `bash`

```
# From your local computer
```

9. `scp -i ~/Downloads/nerv-sgx-vm1_key.pem -r ./nerv nervadmin@YOUR-VM-IP:/home/nervadmin/`
10. Test on VM:
11. `bash`

```
# SSH into VM

cd nerv

cargo build --features tee_real
```

12. `cargo run --features tee_real`

TEE Phase 2: Replace Mocks with Real Behavior (2-3 months)

Why This Phase Matters

We replace pretend security with real hardware verification.

Step 1: Prepare Development Environment (Week 1)

1. Install VS Code extensions:
    - Open VS Code
    - Click Extensions (square icon on left)
    - Search and install:
        - "rust-analyzer"
        - "CodeLLDB"
        - "Remote - SSH"
2. Connect VS Code to VM:
    - Click Remote Explorer icon (left sidebar)
    - Click "+" → "Connect to Host"
    - Enter: `ssh nervadmin@YOUR-VM-IP`
    - **Choose the .pem key file when asked**

Step 2: Implement Real Attestation (Week 2-4)

File to edit: `src/privacy/tee/attestation.rs`

1. Find mock verification function (search for "verify_attestation_report")
2. Replace with real verification:
3. `rust`

```
#[cfg(feature = "tee_real")]

pub fn verify_attestation_report(report: &AttestationReport) -> Result<bool,
AttestationError> {

    use std::time::{SystemTime, UNIX_EPOCH};



    // 1. Check timestamp (within 5 minutes)

    let now = SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();

    if (now as i64 - report.timestamp as i64).abs() > 300 {

        return Err(AttestationError::TimestampError);

    }



    // 2. Verify quote using DCAP

    // This is simplified - you'll need to call actual DCAP functions

    let verify_result = unsafe {

        // Placeholder: Real DCAP verification goes here

        sgx_dcap_verify_quote(report.quote.as_ptr(), report.quote.len())

    };
    if verify_result != 0 {
        return Err(AttestationError::QuoteVerification(format!("DCAP error {}", verify_result)));
    }

    // 3. Check measurement matches expected
    let expected_measurement = [
        0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0, // Replace with real measurement
        // ... 48 bytes total
    ];

    if report.measurement != expected_measurement {
        return Err(AttestationError::MeasurementMismatch);
    }

    Ok(true)
```

4. }
5. Get real measurement:
   - Build enclave on VM:
   - bash
   - `cargo build --features tee_real --release`
   - **Look in logs for "Measurement:" or similar**
   - **Copy the 48-byte hex value**

- **Replace `expected_measurement` with your actual value**

Step 3: Real Model Loading (Week 5-6)

1. Find model loading code:
    - Open `src/consensus/predictor.rs`
    - **Look for `Predictor::new()` function**
2. **Replace fallback with real loading:**
3. `rust`

```
pub fn new(config: &ConsensusConfig) -> Result<Self> {

    // Try to load model

    let model_path = &config.predictor_model_path;



    // Check file exists

    if !std::path::Path::new(model_path).exists() {

        return Err(NervError::Model(format!("Model file not found: {}",
model_path)));

    }



    // Load model

    let model = load_model(model_path)?;



    // Verify model hash
    let expected_hash = hex::decode("a1b2c3d4e5f6...")?; // Replace with
actual hash

    let actual_hash = blake3::hash(&std::fs::read(model_path)?);



    if actual_hash.as_bytes() != &expected_hash[..] {

        return Err(NervError::Model("Model hash mismatch".into()));

    }



    Ok(Self { model })
```

4. }
5. Get model hash:
   - Download model from your source
   - Calculate hash:
   - `bash`

```
# On VM
```

   - `blake3 your-model-file.bin`
   - **Copy the hash and replace in code**

Step 4: Testing Strategy (Week 7-8)

1. Create test directory:
2. `bash`
3. `mkdir -p tests/integration_tee`
4. Create real hardware test:
   `tests/integration_tee/real_attestation.rs`:

5. rust

```rust
#[cfg(feature = "tee_real")]

#[tokio::test]

async fn test_real_attestation() {

    use nerv::privacy::tee::attestation;



    // Generate real attestation

    let report = attestation::generate_report().await.unwrap();



    // Verify it

    let result = attestation::verify_attestation_report(&report).await;



    assert!(result.is_ok(), "Real attestation should pass");
```

6. }
7. Run tests:
8. bash

```bash
# Regular tests (mock mode)

cargo test



# Real hardware tests
```

9. `cargo test --features tee_real --test real_attestation`

TEE Phase 3: Security & Auditing (Ongoing)

Why This Phase Matters

We need to ensure the TEE implementation is actually secure.

Step 1: Self-Audit Checklist (Week 1-2)

Create file `TEE_SECURITY_CHECKLIST.md`:

```markdown
# TEE Security Checklist


## Critical Checks

- [ ] No mock/simulation code in release builds

- [ ] All attestation failures logged

- [ ] Measurements pinned and verified

- [ ] Timestamp freshness checked (±5 min)

- [ ] Failed attestations cause rejection


## Code Review Points

1. File: attestation.rs

   - [ ] Line 45: DCAP verification called
```

```
    - [ ] Line 52: Measurement comparison

    - [ ] Line 60: Error handling


2. File: sgx.rs

    - [ ] Line 123: Real quote generation

    - [ ] Line 145: No secret data in logs


## Test Results

- [ ] Invalid quote rejected: PASS/FAIL

- [ ] Wrong measurement rejected: PASS/FAIL

- [ ] Old timestamp rejected: PASS/FAIL
```

Step 2: Add Logging & Metrics (Week 3)

1. Add logging to verification:
2. `rust`

```rust
if verify_result != 0 {

    tracing::error!("Quote verification failed: DCAP error {}",
verify_result);

    return Err(AttestationError::QuoteVerification(...));

} else {

    tracing::info!("Attestation verified successfully");
```

3. }
4. Add metrics (edit `src/utils/metrics.rs`):
5. rust

```
lazy_static! {
    pub static ref ATTESTATION_SUCCESS: Counter = register_counter!(
        "nerv_tee_attestation_success_total",
        "Number of successful TEE attestations"
    );
    pub static ref ATTESTATION_FAILURE: Counter = register_counter!(
        "nerv_tee_attestation_failure_total",
        "Number of failed TEE attestations"
    );
}


// In verification function:
if verify_result == 0 {
    ATTESTATION_SUCCESS.inc();
} else {
    ATTESTATION_FAILURE.inc();
```

6. }
7. Test logging:
8. bash

```
RUST_LOG=info cargo run --features tee_real
```

9. `# Check logs for "Attestation verified" messages`

Step 3: Fuzzing Tests (Week 4)

1. Install fuzzer:
2. `bash`
3. `cargo install cargo-fuzz`
4. Create fuzz target:
5. `bash`

```
cd nerv
```

6. `cargo fuzz init`
7. Edit `fuzz/fuzz_targets/attestation.rs`:
8. `rust`

```
#![no_main]

use libfuzzer_sys::fuzz_target;

use nerv::privacy::tee::attestation::verify_attestation_report;



fuzz_target!(|data: &[u8]| {

    // Create fake attestation from random data

    let fake_report = AttestationReport {

        quote: data.to_vec(),

        measurement: [0u8; 48],
        timestamp: 0,
    };

    // Should handle without crashing
    let _ = verify_attestation_report(&fake_report);
```

9. });
10. Run fuzzer:
11. bash
12. `cargo fuzz run attestation -- -max_total_time=3600`

Step 4: Prepare for External Audit (Week 5-6)

1. Create audit documentation:
2. bash
3. `mkdir -p docs/audit`
4. Create `docs/audit/TEE_SECURITY_REPORT.md`:
5. markdown

```
# NERV TEE Security Report


## Threat Model

1. Fake quotes: Mitigated by DCAP verification

2. Replay attacks: Mitigated by timestamp check

3. Measurement spoofing: Mitigated by pinning
```

```
## Implementation Details

- SGX DCAP library: v1.16.100.2

- Quote verification: Full chain validation

- Freshness window: 5 minutes


## Test Results

- Invalid quote rejection: ✓

- Measurement mismatch: ✓

- Timestamp rejection: ✓


## Known Limitations

- Requires Intel/AMD hardware
```

6. `- Initial measurement hardcoded`
7. Share for community review:
    - Create GitHub issue: "TEE implementation ready for review"
    - Post on NERV Discord/Telegram
    - Email to security researchers you trust

PART 2: NETWORKING - PHASE 1 ONLY

Networking Phase 1: Choose & Integrate a Mature P2P Stack (1-2 months)

Why This Phase Matters

The current networking only works on a single computer. We need real internet connectivity.

Step 1: Research libp2p (Week 1)

1. Visit libp2p website:
   - Go to: https://libp2p.io/
   - **Click "Documentation"**
   - **Read "Getting Started"**
2. **Check Rust implementation:**
   - **Go to: https://github.com/libp2p/rust-libp2p**
   - **Look at examples directory**
3. **Compare with current code:**
   - **Your current `src/network/dht.rs` → libp2p Kademlia**
   - **Your current `src/network/gossip.rs` → libp2p Gossipsub**

Step 2: Add libp2p Dependencies (Week 1)

1. Open `Cargo.toml`:
   - Find **`[dependencies]` section**
2. **Add libp2p dependencies:**
3. `toml`

```toml
[dependencies]

# Add these lines:

libp2p = { version = "0.53", features = ["tcp", "quic", "noise", "yamux",
"gossipsub", "kad", "identify", "ping", "request-response", "autonat",
"relay"] }
```

```
libp2p-gossipsub = "0.47"
```

4. `libp2p-kad = "0.45"`
5. Build to check dependencies:
6. `bash`
7. `cargo build`
   - **If errors, adjust versions (check https://crates.io for latest)**

Step 3: Create libp2p Swarm (Week 2-3)

1. Create new file: `src/network/libp2p_swarm.rs`
2. Add basic structure:
3. `rust`

```rust
use libp2p::{

    gossipsub, identify, kad::{self, Kademlia, KademliaConfig},

    ping, relay, autonat, Swarm, SwarmBuilder, Noise, Yamux,

    Transport, Multiaddr, PeerId, identity, tcp

};

use libp2p::swarm::{SwarmEvent, NetworkBehaviour};

use tokio::sync::mpsc;



// Define network behavior

#[derive(NetworkBehaviour)]

struct NervBehaviour {

    gossipsub: gossipsub::Behaviour,
    kademlia: Kademlia<kad::store::MemoryStore>,

    identify: identify::Behaviour,

    ping: ping::Behaviour,

    relay: relay::Behaviour,

    autonat: autonat::Behaviour,

}



// Main network manager

pub struct P2PManager {

    swarm: Swarm<NervBehaviour>,

}



impl P2PManager {

    pub async fn new() -> Self {

        // Create cryptographic identity for this node

        let local_key = identity::Keypair::generate_ed25519();

        let peer_id = PeerId::from(local_key.public());



        println!("Our peer ID: {}", peer_id);



        // Create transport (how we send data)
```

```rust
let transport = tcp::tokio::Transport::new(tcp::Config::default())

    .upgrade(libp2p::core::upgrade::Version::V1Lazy)

    .authenticate(Noise::new(&local_key).unwrap())

    .multiplex(Yamux::new())

    .boxed();


// Configure Gossipsub (for broadcasting)

let gossipsub_config = gossipsub::ConfigBuilder::default()

    .heartbeat_interval(std::time::Duration::from_secs(10))

    .build()

    .unwrap();


// Configure Kademlia DHT (for peer discovery)

let mut kad_config = KademliaConfig::default();

kad_config.set_query_timeout(std::time::Duration::from_secs(30));


let store = kad::store::MemoryStore::new(peer_id);

let kademlia = Kademlia::with_config(peer_id, store, kad_config);


// Create the network behavior

let behaviour = NervBehaviour {
```

```
            gossipsub: gossipsub::Behaviour::new(
                gossipsub::MessageAuthenticity::Signed(local_key.clone()),
                gossipsub_config
            ).unwrap(),
            kademlia,
            identify: identify::Behaviour::new(
                identify::Config::new("/nerv/1.0".to_string(),
local_key.public())
            ),
            ping: ping::Behaviour::new(ping::Config::new()),
            relay: relay::Behaviour::new(peer_id, relay::Config::default()),
            autonat: autonat::Behaviour::new(peer_id,
autonat::Config::default()),
        };

        // Create the swarm (main network object)
        let mut swarm = SwarmBuilder::with_tokio_executor(transport,
behaviour, peer_id)
            .build();

        // Listen on all interfaces
        swarm.listen_on("/ip4/0.0.0.0/tcp/0".parse().unwrap()).unwrap();
```

```
    Self { swarm }

}


pub async fn run(mut self) {

    println!("Starting P2P network...");


    loop {

        match self.swarm.select_next_some().await {

            SwarmEvent::NewListenAddr { address, .. } => {

                println!("Listening on: {}", address);

            }

            SwarmEvent::Behaviour(event) => {

                self.handle_event(event).await;

            }

            _ => {}

        }

    }

}


async fn handle_event(&mut self, event: NervEvent) {
    match event {
        // We'll fill this in later
        _ => println!("Network event: {:?}", event),
    }
}
```

4. }

Step 4: Connect to Main Code (Week 3)

1. Modify main network module:
   - Open `src/network/mod.rs`
   - **Replace content with:**
2. `rust`

```
mod libp2p_swarm;
pub use libp2p_swarm::P2PManager;
```

3. `// Keep existing DHT/gossip for now, we'll migrate later`
4. Update node startup:
   - Open main node file (likely `src/node/mod.rs`)
   - **Find where network is initialized**
   - **Replace with:**
5. `rust`

```
use crate::network::P2PManager;
```

```
impl Node {
    pub async fn new(config: Config) -> Result<Self> {
        // ... existing code ...

        // Initialize P2P network
        let network = P2PManager::new().await;

        // ... rest of initialization ...

        Ok(Self {
            network,
            // ... other fields
        })
    }
```

6. }

Step 5: Test Basic Connectivity (Week 4)

1. Create simple test program: `test_p2p.rs`:
2. rust

```
use nerv::network::P2PManager;
```

```rust
#[tokio::main]

async fn main() {

    println!("Starting NERV P2P test...");



    let p2p = P2PManager::new().await;



    println!("Node started. Press Ctrl+C to exit.");

    println!("Try connecting from another terminal...");



    // Run for 60 seconds

    tokio::time::sleep(tokio::time::Duration::from_secs(60)).await;
```

3. `}`
4. Run test:
5. `bash`
6. `cargo run --example test_p2p`
   - **Should show: "Listening on: /ip4/0.0.0.0/tcp/XXXXX"**
7. **Test with two nodes:**
   - **Open two terminals**
   - **Terminal 1:** `cargo run --example test_p2p`
   - **Terminal 2:** `cargo run --example test_p2p`
   - **They should find each other automatically**

Step 6: Add Bootstrap Nodes (Week 4)

1. Create configuration:
   `config/network.toml`:
2. toml

```
# Initial nodes to connect to

bootstrap_nodes = [


"/ip4/104.131.131.82/tcp/4001/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",  # IPFS bootstrap


"/ip4/104.131.131.82/udp/4001/quic/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",

]



# Local test node (for development)

local_test_nodes = [

  "/ip4/127.0.0.1/tcp/30333",
```

3.]
4. Load config in P2PManager:
5. rust

```
impl P2PManager {

    pub async fn new() -> Self {

        // ... existing code ...



        // Load bootstrap nodes from config
        let config: NetworkConfig =
load_config("config/network.toml").unwrap();


        // Dial bootstrap nodes

        for addr in &config.bootstrap_nodes {

            match addr.parse() {

                Ok(addr) => {

                    println!("Dialing bootstrap: {}", addr);

                    swarm.dial(addr).unwrap();

                }

                Err(e) => println!("Invalid bootstrap address {}: {}", addr,
e),

            }

        }


        Self { swarm }

    }
```

6. }

Step 7: Migration Plan (Week 5-8)

Create migration document `NETWORK_MIGRATION_PLAN.md`:

```
markdown
```

# Network Migration from Custom to libp2p

## Phase A: Coexistence (Week 5)

1. Keep old DHT/gossip code

2. New code uses libp2p

3. Bridge between them

## Phase B: Gradual Replacement (Week 6-7)

1. Replace DHT with Kademlia

2. Replace gossip with Gossipsub

3. One component at a time

## Phase C: Cleanup (Week 8)

1. Remove old code

2. Update all tests

3. Final integration

## File-by-File Migration:

1. `src/network/dht.rs` → Use libp2p-kad

2. `src/network/gossip.rs` → Use libp2p-gossipsub

3. `src/network/transport.rs` → Use libp2p transports

Step 8: Testing & Validation (Ongoing)

1. Create network test suite:
2. bash
3. `mkdir -p tests/network`
4. Basic connectivity test: `tests/network/basic.rs`:
5. rust

```rust
#[tokio::test]

async fn test_p2p_connectivity() {

    // Start two nodes

    let node1 = P2PManager::new().await;

    let node2 = P2PManager::new().await;



    // Get node1's address

    let addr = node1.listen_addr().unwrap();



    // Node2 connects to node1

    node2.dial(addr).await.unwrap();



    // Wait for connection

    tokio::time::sleep(Duration::from_secs(2)).await;
```

```
// Verify connection

assert!(node1.has_peers());

assert!(node2.has_peers());
```

6. }
7. Run network tests:
8. `bash`
9. `cargo test --test network`

COMPLETION CHECKLISTS

TEE Integration Checklist:

- **Azure/AWS account created**
- **SGX/SEV VM running**
- **DCAP libraries installed**
- **Real attestation code implemented**
- **Model loading without fallbacks**
- **Security logging added**
- **Fuzzing tests passing**
- **Audit documentation complete**

Networking Phase 1 Checklist:

- **libp2p dependencies added**
- **P2PManager created**
- **Basic connectivity working**
- **Bootstrap nodes configured**

- **Migration plan documented**
- **Initial tests passing**

TIMELINE SUMMARY

Phase	Duration	Key Deliverables
TEE Phase 1	**1-2 months**	**Working SGX VM, Real libraries**
TEE Phase 2	**2-3 months**	**Real attestation, No mocks**
TEE Phase 3	**Ongoing**	**Security audit, Fuzzing**
Network Phase 1	**1-2 months**	**libp2p integration, Basic P2P**

Total Estimated Time: 4-7 months for all phases

TROUBLESHOOTING GUIDE

Common TEE Issues:

1. SGX VM not available in Azure:
    - Try different region

    - Check if DCsv3 series is supported
    - Contact Azure support
2. DCAP libraries not installing:
3. `bash`

```
# Try alternative install

sudo apt install -y libsgx-dcap-ql-dev
```

4. `sudo apt install -y libsgx-dcap-quote-verify-dev`
5. Quote generation fails:
    - Check VM has SGX enabled: `dmesg | grep sgx`
    - **Verify DCAP service running:** `ps aux | grep dcap`

Common Networking Issues:

1. libp2p version conflicts:
    - Check https://crates.io **for compatible versions**
    - **Use** `cargo tree` **to see dependencies**
2. **Nodes can't connect:**
    - **Check firewall:** `sudo ufw status`
    - **Try different port**
    - **Use** `autonat` **feature for NAT traversal**
3. **Build errors:**
4. `bash`

```
# Clean and rebuild

cargo clean
```

5. `cargo build`

NEXT STEPS AFTER COMPLETION

1. TEE:
   - External security audit
   - Multi-VM deployment
   - Performance benchmarking
2. Networking:
   - Proceed to Phase 2 (Peer Lifecycle)
   - Test with 10+ nodes
   - Measure latency/TPS

Remember: Start small, test frequently, and document everything. Each successful step brings NERV closer to being a real, secure, scalable blockchain!

Detailed Guide for Phase 2: Implement Real Peer Lifecycle (2-3 months)

Why This Phase Matters (Simple Explanation)

Right now, NERV nodes can't actually talk to each other over the internet. The networking code works only "in memory" on a single computer. For a real blockchain, nodes need to:

- Discover each other across the internet
- Share transactions and blocks
- Route messages through mixers (for privacy)
- Scale to thousands of nodes

This phase makes your nodes actually connect and communicate like a real blockchain network.

Phase 2 Overview: What You'll Build

You'll turn the libp2p setup from Phase 1 into a working peer-to-peer network where:

1. Nodes find each other using a DHT (like a phone book for nodes)
2. They gossip (share) transactions and blocks
3. They manage connections intelligently
4. Everything works with real internet connections

Estimated Time: 2-3 months part-time
Prerequisites: Completed Phase 1 (libp2p integrated)

Step-by-Step Implementation Guide

Week 1-2: Basic Peer Discovery

Goal: Nodes should find each other automatically

1.1 Create Bootstrap Configuration

Create file `config/network.toml`:

```toml
# Initial known nodes (like DNS for your network)
bootstrap_nodes = [
  "/ip4/54.123.45.67/tcp/30333/p2p/12D3KooWABC...",  # Example node 1
  "/ip4/98.76.54.32/tcp/30333/p2p/12D3KooWXYZ..."   # Example node 2
]

# How many peers to connect to
min_peers = 5
max_peers = 50

# Port to listen on

listen_port = 30333
```

1.2 Implement Bootstrap Logic

Edit `src/network/libp2p_swarm.rs`:

```
rust
```

```
impl P2PManager {
    pub async fn new(local_key: identity::Keypair) -> Self {
        // ... (existing code from Phase 1) ...

        // BOOTSTRAP: Connect to known nodes
        let bootstrap_nodes = load_bootstrap_config(); // Load from
config/network.toml

        for addr in bootstrap_nodes {
            tracing::info!("Dialing bootstrap node: {}", addr);
            match swarm.dial(addr.parse().unwrap()) {
                Ok(_) => tracing::info!("Successfully dialed bootstrap node"),
                Err(e) => tracing::warn!("Failed to dial bootstrap node: {}",
e),
            }
        }

        // KADEMLIA: Start DHT discovery
        for bootstrap_addr in bootstrap_nodes {
            let peer_id = extract_peer_id_from_addr(&bootstrap_addr);
            swarm.behaviour_mut().kademlia.add_address(&peer_id,
bootstrap_addr.parse().unwrap());
        }

        swarm.behaviour_mut().kademlia.bootstrap().unwrap();

        Self { swarm, command_sender: tx }
    }

}
```

1.3 Test Basic Connectivity

Create test script `tests/network_basic.sh`:

```
bash
```

```
#!/bin/bash
# Run 3 local nodes that find each other
```

```
echo "Starting Node 1 on port 30333..."
cargo run -- --port 30333 --name node1 &

echo "Starting Node 2 on port 30334..."
cargo run -- --port 30334 --name node2 --bootstrap /ip4/127.0.0.1/tcp/30333 &

echo "Starting Node 3 on port 30335..."
cargo run -- --port 30335 --name node3 --bootstrap /ip4/127.0.0.1/tcp/30333 &

sleep 10  # Wait for nodes to connect

echo "Checking connections..."
```

```
# Each node should show 2 peers
```

Week 3-4: Message Propagation (Gossip)

Goal: Nodes should share transactions and blocks

2.1 Define Message Types

Create `src/network/messages.rs`:

```
rust
```

```rust
#[derive(Serialize, Deserialize, Clone)]
pub enum NetworkMessage {
    // Transaction messages
    Transaction {
        tx_hash: [u8; 32],
        encrypted_payload: Vec<u8>,  // Already encrypted from mixer
        attestations: Vec<AttestationReport>,
    },

    // Block messages
    Block {
        block_hash: [u8; 32],
        embedding_root: [u8; 32],
        signatures: Vec<BlsSignature>,
    },
    // Peer management
    PeerList(Vec<PeerInfo>),

    // Health check
    Ping { timestamp: u64, nonce: u32 },
    Pong { timestamp: u64, nonce: u32 },
}
```

2.2 Implement Gossip Topics

Edit `src/network/libp2p_swarm.rs`:

```rust
impl P2PManager {
    pub async fn new(local_key: identity::Keypair) -> Self {
        // ... (existing code) ...

        // Create gossip topics
        let tx_topic = gossipsub::IdentTopic::new("nerv-tx-v1");
        let block_topic = gossipsub::IdentTopic::new("nerv-block-v1");
        let peer_topic = gossipsub::IdentTopic::new("nerv-peer-v1");

        // Subscribe to topics
        behaviour.gossipsub.subscribe(&tx_topic).unwrap();
        behaviour.gossipsub.subscribe(&block_topic).unwrap();
        behaviour.gossipsub.subscribe(&peer_topic).unwrap();

        // Store topic IDs for later use
        let topics = Topics {
            tx: tx_topic,
            block: block_topic,
            peer: peer_topic,
        };

        Self { swarm, command_sender: tx, topics }
    }

    pub async fn broadcast_transaction(&mut self, tx: NetworkMessage) {
        let data = bincode::serialize(&tx).unwrap();
        self.swarm.behaviour_mut().gossipsub.publish(self.topics.tx.clone(),
data).unwrap();
    }

}
```

2.3 Handle Incoming Messages

Add to swarm event loop:

```
rust
```

```
// Inside the swarm event loop
match event {
    SwarmEvent::Behaviour(NervEvent::Gossipsub(gossipsub::Event::Message {
        message, ..
    })) => {
        let msg = match bincode::deserialize(&message.data) {
            Ok(m) => m,
            Err(_) => return,  // Invalid message
        };

        match msg {
            NetworkMessage::Transaction { tx_hash, encrypted_payload,
attestations } => {
                tracing::info!("Received transaction: {}",
hex::encode(&tx_hash[..8]));

                // Forward to TEE mixer (if we're a mixer node)
                if self.is_mixer_node {
                    self.forward_to_mixer(encrypted_payload,
attestations).await;
                }
                // Otherwise, add to encrypted mempool
                else {
                    self.add_to_mempool(encrypted_payload,
attestations).await;
                }
            }
            // Handle other message types...
        }
    }
}
```

Week 5-6: Peer Scoring & Reputation

Goal: Prevent spam and bad actors

3.1 Implement Peer Scoring

Create `src/network/scoring.rs`:

```rust
pub struct PeerScore {
    peer_id: PeerId,
    score: i32,  // Can go negative
    last_seen: Instant,
    violations: Vec<Violation>,
}

#[derive(Clone)]
pub enum Violation {
    InvalidMessage,      // -10 points
    SpamMessage,         // -5 points per spam
    NoAttestation,       // -50 points (security violation)
    BadSignature,        // -100 points
}

impl PeerScore {
    pub fn new(peer_id: PeerId) -> Self {
        Self {
            peer_id,
            score: 100,  // Start with positive score
            last_seen: Instant::now(),
            violations: Vec::new(),
        }
    }

    pub fn apply_penalty(&mut self, violation: Violation) {
        let penalty = match violation {
            Violation::InvalidMessage => 10,
            Violation::SpamMessage => 5,
            Violation::NoAttestation => 50,
            Violation::BadSignature => 100,
        };

        self.score -= penalty;
        self.violations.push(violation);

        if self.score < 0 {
            tracing::warn!("Banning peer {}: score {}", self.peer_id,
self.score);
            self.ban_peer();
        }
    }

    pub fn reward_good_behavior(&mut self) {
        // Slowly recover score for good behavior
        if self.score < 100 {
            self.score += 1;
        }
        self.last_seen = Instant::now();
    }

}
```

3.2 Integrate with libp2p

Edit `src/network/libp2p_swarm.rs`:

rust

```
// Add scoring to behaviour
struct NervBehaviour {
    gossipsub: gossipsub::Behaviour,
    kademlia: Kademlia<kad::store::MemoryStore>,
    identify: identify::Behaviour,
    ping: ping::Behaviour,
    relay: relay::Behaviour,
    autonat: autonat::Behaviour,
    scoring: PeerScoring,  // New: Track peer scores
}

// In message handler:
match msg {
    NetworkMessage::Transaction { .. } => {
        // Verify before processing
        if self.verify_message(&msg).is_ok() {
            self.scoring.reward_good_behavior(&source_peer);
            // Process message...
        } else {
            self.scoring.apply_penalty(&source_peer,
Violation::InvalidMessage);
            // Disconnect from bad peer
            self.swarm.disconnect_peer_id(source_peer).ok();
        }
    }

}
```

Week 7-8: Encrypted Mempool Integration

Goal: Handle private transactions securely

4.1 Create Encrypted Mempool

Create `src/network/mempool.rs`:

```rust
pub struct EncryptedMempool {
    // Store encrypted transactions (encrypted again for at-rest storage)
    transactions: HashMap<[u8; 32], EncryptedTx>,
    // Track which shard each transaction belongs to
    shard_map: HashMap<[u8; 32], ShardId>,
    // Maximum size
    max_size: usize,
}

impl EncryptedMempool {
    pub fn new(max_size: usize) -> Self {
        Self {
            transactions: HashMap::new(),
            shard_map: HashMap::new(),
            max_size,
        }
    }

    pub async fn add_transaction(
        &mut self,
        encrypted_tx: Vec<u8>,
        attestations: Vec<AttestationReport>,
        shard_id: ShardId,
    ) -> Result<[u8; 32], MempoolError> {
        // 1. Verify attestations are valid
        for attestation in &attestations {
            if !verify_attestation(attestation).await? {
                return Err(MempoolError::InvalidAttestation);
            }
        }

        // 2. Generate transaction hash
        let tx_hash = blake3::hash(&encrypted_tx).into();

        // 3. Re-encrypt for at-rest storage (extra layer)
        let storage_ciphertext = encrypt_for_storage(&encrypted_tx)?;

        // 4. Store
        let tx = EncryptedTx {
            ciphertext: storage_ciphertext,
            attestations,
            received_at: Instant::now(),
        };

        self.transactions.insert(tx_hash, tx);
        self.shard_map.insert(tx_hash, shard_id);

        // 5. Evict if full (oldest first)
        if self.transactions.len() > self.max_size {
            self.evict_oldest();
        }

        Ok(tx_hash)
    }

    pub fn get_for_shard(&self, shard_id: ShardId, limit: usize) -> Vec<([u8;
32], EncryptedTx)> {
        // Return transactions for specific shard
        self.transactions
            .iter()
            .filter(|(hash, _)| self.shard_map.get(hash) == Some(&shard_id))
            .take(limit)
            .map(|(hash, tx)| (*hash, tx.clone()))
            .collect()
    }

}
```

4.2 Connect Mempool to Network

Edit main node file (`src/node/mod.rs`):

```
rust
```

```
pub struct Node {
    network: P2PManager,
    mempool: EncryptedMempool,
    tee_runtime: TEERuntime,
    // ... other components
}

impl Node {
    pub async fn run(mut self) -> Result<()> {
        loop {
            tokio::select! {
                // Network events
                network_event = self.network.next_event() => {
                    self.handle_network_event(network_event).await?;
                }

                // TEE events
                tee_event = self.tee_runtime.next_event() => {
                    self.handle_tee_event(tee_event).await?;
                }

                // Mempool batching (every 100ms)
                _ = tokio::time::sleep(Duration::from_millis(100)) => {
                    self.batch_transactions_for_shards().await?;
                }
            }
        }
    }

    async fn handle_network_event(&mut self, event: NetworkEvent) ->
Result<()> {
        match event {
            NetworkEvent::NewTransaction(tx_hash, encrypted_payload,
attestations) => {
                // Determine which shard this belongs to
                let shard_id =
self.determine_shard(&encrypted_payload).await?;

                // Add to encrypted mempool
                self.mempool.add_transaction(encrypted_payload, attestations,
shard_id).await?;

                tracing::debug!("Added tx {} to shard {} mempool",
                    hex::encode(&tx_hash[..4]), shard_id);
            }
            // ... other events
        }
        Ok(())
    }

}
```

Week 9-10: Multi-Node Testing

Goal: Test with real multiple nodes

5.1 Create Docker Setup

Create `docker/docker-compose-multi.yml`:

```yaml
version: '3.8'
services:
  nerv-bootstrap:
    build: .
    command: ["--name", "bootstrap", "--port", "30333", "--listen-addr",
"0.0.0.0"]
```

```yaml
    ports:
      - "30333:30333"
    networks:
      nerv-net:
        ipv4_address: 172.20.0.10

  nerv-node-1:
    build: .
    command:
      - "--name"
      - "node-1"
      - "--port"
      - "30334"
      - "--bootstrap"
      - "/ip4/172.20.0.10/tcp/30333/p2p/BOOTSTRAP_PEER_ID"
    depends_on:
      - nerv-bootstrap
    networks:
      nerv-net:
        ipv4_address: 172.20.0.11

  nerv-node-2:
    build: .
    command:
      - "--name"
      - "node-2"
      - "--port"
      - "30335"
      - "--bootstrap"
      - "/ip4/172.20.0.10/tcp/30333/p2p/BOOTSTRAP_PEER_ID"
    depends_on:
      - nerv-bootstrap
    networks:
      nerv-net:
        ipv4_address: 172.20.0.12

  nerv-node-3:
    build: .
    command:
      - "--name"
      - "node-3"
      - "--port"
```

```
        - "30336"
        - "--bootstrap"
        - "/ip4/172.20.0.10/tcp/30333/p2p/BOOTSTRAP_PEER_ID"
        - "--bootstrap"
        - "/ip4/172.20.0.11/tcp/30334/p2p/NODE_1_PEER_ID"
      depends_on:
        - nerv-bootstrap
        - nerv-node-1
      networks:
        nerv-net:
          ipv4_address: 172.20.0.13

networks:
  nerv-net:
    ipam:
      config:
```

```
        - subnet: 172.20.0.0/16
```

5.2 Create Test Script

Create `scripts/test-multi-node.sh`:

```bash
#!/bin/bash
echo "=== NERV Multi-Node Network Test ==="
echo "Starting 4-node cluster..."

# Build the Docker image
docker build -t nerv-node:test .

# Start the cluster
docker-compose -f docker/docker-compose-multi.yml up -d

echo "Waiting for nodes to start..."
sleep 15

echo "=== Checking Node Status ==="
# Check each node's peer count
for container in nerv-bootstrap nerv-node-1 nerv-node-2 nerv-node-3; do
    echo -n "$container: "
    docker exec $container curl -s http://localhost:9090/metrics | grep
"peers_connected" || echo "No metrics"
done

echo "=== Sending Test Transactions ==="
# Generate and send test transactions
for i in {1..10}; do
    echo "Sending transaction $i..."
    # Use the first node's API to submit a test transaction
    TX_HASH=$(docker exec nerv-node-1 curl -s -X POST
http://localhost:8080/submit-test-tx)
    echo "Transaction $i hash: $TX_HASH"
    sleep 0.5
done

echo "=== Verifying Propagation ==="
# Check if transactions reached all nodes
for container in nerv-bootstrap nerv-node-1 nerv-node-2 nerv-node-3; do
    echo -n "$container mempool size: "
    docker exec $container curl -s http://localhost:8080/mempool-size
done

echo "=== Network Stats ==="
docker exec nerv-bootstrap curl -s http://localhost:9090/metrics | grep -E
"(messages_sent|messages_received|peer_count)"

echo "Test complete!"
```

```
echo "To view logs: docker-compose -f docker/docker-compose-multi.yml logs -f"
```

Week 11-12: Performance Optimization

Goal: Make network fast and efficient

6.1 Implement Connection Pooling

Edit `src/network/connection_pool.rs`:

```
rust
```

```rust
pub struct ConnectionPool {
    // Reuse connections instead of creating new ones
    connections: HashMap<PeerId, PooledConnection>,
    // LRU cache for quick eviction
    lru: LruCache<PeerId, ()>,
    max_pool_size: usize,
}

impl ConnectionPool {
    pub async fn get_connection(&mut self, peer_id: &PeerId) ->
Result<PooledConnection> {
        // 1. Check if we already have a connection
        if let Some(conn) = self.connections.get_mut(peer_id) {
            if conn.is_healthy() {
                self.lru.get(peer_id);  // Update LRU
                return Ok(conn.clone());
            }
        }

        // 2. Create new connection if pool not full
        if self.connections.len() < self.max_pool_size {
            let new_conn = self.create_connection(peer_id).await?;
            self.connections.insert(*peer_id, new_conn.clone());
            self.lru.put(*peer_id, ());
            return Ok(new_conn);
        }

        // 3. Evict least recently used and create new
        if let Some((lru_peer, _)) = self.lru.pop_lru() {
            self.connections.remove(&lru_peer);
            let new_conn = self.create_connection(peer_id).await?;
            self.connections.insert(*peer_id, new_conn.clone());
            self.lru.put(*peer_id, ());
            Ok(new_conn)
        } else {
            Err(ConnectionError::PoolFull)
        }
    }

}
```

6.2 Add Message Compression

Edit message sending code:

```rust
impl P2PManager {
    pub async fn send_message(&mut self, peer_id: PeerId, message: NetworkMessage) -> Result<()> {
        let serialized = bincode::serialize(&message)?;

        // Compress if message is large (>1KB)
        let compressed = if serialized.len() > 1024 {
            let mut encoder = zstd::Encoder::new(Vec::new(), 3)?;
            encoder.write_all(&serialized)?;
            encoder.finish()?
        } else {
            serialized
        };

        // Add compression header
        let final_message = if compressed.len() < serialized.len() {
            let mut msg = vec![0x01];  // Compression flag = compressed
            msg.extend_from_slice(&compressed);
            msg
        } else {
            let mut msg = vec![0x00];  // Compression flag = uncompressed
            msg.extend_from_slice(&serialized);
            msg
        };

        // Send via libp2p
        self.swarm.behaviour_mut().request_response.send_request(&peer_id, final_message);

        Ok(())
    }

}
```

Phase 2 Completion Checklist:

- Nodes can discover each other via bootstrap/DHT
- Transactions propagate via gossip to all nodes
- Encrypted mempool stores and retrieves transactions
- Peer scoring prevents spam/bad actors
- 4+ nodes can run together in Docker
- Messages are compressed for efficiency
- Connections are pooled and reused

Detailed Guide for Phase 3: Battle-Testing & Hardening (3-4 months)

Why This Phase Matters

Now that nodes can talk to each other, we need to ensure:

1. The network works reliably on real internet (not just local)
2. It can handle attacks (DoS, spam, etc.)
3. It performs well under load (high TPS)
4. It's ready for real users

Phase 3 Overview

Test on real internet → Find/fix problems → Make it robust

Step-by-Step Battle Testing Guide

Month 1: Real Internet Testnet

1.1 Set Up Cloud Testnet (Week 1)

Create cloud deployment script `deploy/cloud-setup.sh`:

```bash
#!/bin/bash
# Deploy NERV nodes to AWS/Azure/DigitalOcean

REGIONS=("us-east-1" "eu-west-1" "ap-southeast-1" "sa-east-1")
NODES_PER_REGION=3
TOTAL_NODES=$(( ${#REGIONS[@]} * NODES_PER_REGION ))

echo "Deploying $TOTAL_NODES NERV testnet nodes..."

# 1. Create base image with NERV
echo "Creating base VM image..."
# (AWS example) Create AMI with NERV installed

# 2. Launch instances in each region
BOOTSTRAP_IP=""
for region in "${REGIONS[@]}"; do
    echo "Launching nodes in $region..."

    for i in $(seq 1 $NODES_PER_REGION); do
        NODE_NAME="nerv-testnet-${region}-${i}"

        # Cloud-specific launch command
        if [ "$CLOUD_PROVIDER" = "aws" ]; then
            aws ec2 run-instances \
                --region $region \
                --image-id ami-xxxxxx \
                --instance-type c6i.large \
                --tag-specifications
"ResourceType=instance,Tags=[{Key=Name,Value=$NODE_NAME}]" \
                --user-data file://deploy/user-data.sh
        fi

        # For first node in first region, set as bootstrap
        if [ -z "$BOOTSTRAP_IP" ]; then
            BOOTSTRAP_IP=$(get_instance_ip $NODE_NAME $region)
            echo "Bootstrap node: $BOOTSTRAP_IP"
        fi
    done
done
```

```
echo "Testnet deployed! Bootstrap at: $BOOTSTRAP_IP:30333"
```

1.2 Create Monitoring Dashboard (Week 2)

Set up Prometheus + Grafana:

`deploy/monitoring/docker-compose-monitor.yml`:

```yaml
version: '3'
services:
  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prom_data:/prometheus
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=nerv123
    ports:
      - "3000:3000"
    volumes:
      - grafana_data:/var/lib/grafana

  node-exporter:
    image: prom/node-exporter
    ports:
      - "9100:9100"
```

`deploy/monitoring/prometheus.yml`:

```yaml
global:
  scrape_interval: 15s
```

```
scrape_configs:
  - job_name: 'nerv-nodes'
    static_configs:
      - targets:
        - 'node1.yourdomain.com:9090'
        - 'node2.yourdomain.com:9090'
        - 'node3.yourdomain.com:9090'

    metrics_path: '/metrics'
```

1.3 Test Cross-Region Connectivity (Week 3-4)

Create test script `tests/cross-region-test.py`:

```
python
```

```
import requests
import time
from datetime import datetime

NODES = [
    "http://node-us-east-1:8080",
    "http://node-eu-west-1:8080",
    "http://node-ap-southeast-1:8080",
    "http://node-sa-east-1:8080"
]

def test_latency():
    """Measure message propagation time between regions"""
    results = {}

    for i, source in enumerate(NODES):
        for j, target in enumerate(NODES):
            if i == j:
                continue

            # Send ping from source to target
            start = time.time()
            response = requests.post(
                f"{source}/send-ping",
                json={"target": target, "timestamp": time.time()}
            )
```

```python
            # Wait for pong
            pong_received = False
            timeout = time.time() + 10
            while time.time() < timeout:
                pongs = requests.get(f"{source}/received-pongs").json()
                if any(t["from"] == target for t in pongs):
                    pong_received = True
                    break
                time.sleep(0.1)

            latency = time.time() - start if pong_received else None
            results[f"{source}→{target}"] = latency

    return results

def test_transaction_propagation():
    """Send transaction from one region, verify it reaches all"""

    # Send transaction from US node
    tx_data = {"test": True, "amount": 100, "timestamp": time.time()}
    response = requests.post(
        f"{NODES[0]}/submit-test-tx",
        json=tx_data
    )
    tx_hash = response.json()["hash"]

    print(f"Transaction submitted: {tx_hash}")

    # Check all nodes received it within 5 seconds
    received_by = []
    for node in NODES:
        for attempt in range(50):  # Check for 5 seconds
            mempool = requests.get(f"{node}/mempool").json()
            if tx_hash in mempool:
                received_by.append(node)
                break
            time.sleep(0.1)

    print(f"Transaction received by {len(received_by)}/{len(NODES)} nodes")
    return len(received_by) == len(NODES)
```

```python
if __name__ == "__main__":
    print("=== Cross-Region Network Test ===")

    # Test 1: Latency
    print("\n1. Measuring latency between regions...")
    latencies = test_latency()
    for path, latency in latencies.items():
        print(f"  {path}: {latency:.3f}s" if latency else f"  {path}: FAILED")

    # Test 2: Propagation
    print("\n2. Testing transaction propagation...")
    success = test_transaction_propagation()
    print(f"  Result: {'PASS' if success else 'FAIL'}")
```

Month 2: Attack Simulation & Hardening

2.1 Simulate DoS Attacks (Week 1-2)

Create attack simulation script `tests/simulate_attacks.py`:

```python
import asyncio
import aiohttp
from concurrent.futures import ThreadPoolExecutor
import random

class AttackSimulator:
    def __init__(self, target_nodes):
        self.targets = target_nodes
        self.results = []

    async def spam_transactions(self, transactions_per_second,
duration_seconds):
        """Simulate transaction spam attack"""
        print(f"Starting spam attack: {tps} TPS for {duration_seconds}s")

        async def send_transaction(target):
            # Generate random (invalid) transaction
            fake_tx = {
                "data": "x" * random.randint(100, 1000),  # Random size
                "signature": "0" * 128,  # Invalid
                "timestamp": time.time()
            }

            try:
                async with aiohttp.ClientSession() as session:
                    async with session.post(
                        f"{target}/submit-tx",
                        json=fake_tx,
                        timeout=2
                    ) as resp:
                        return resp.status
            except:
                return "timeout"

        # Send transactions as fast as possible
        start = time.time()
        sent = 0
        while time.time() - start < duration_seconds:
            tasks = []
            for _ in range(transactions_per_second):
                target = random.choice(self.targets)
                tasks.append(send_transaction(target))

            results = await asyncio.gather(*tasks, return_exceptions=True)
            self.results.extend(results)
            sent += len(tasks)

            # Wait to maintain TPS rate
            await asyncio.sleep(1)

        print(f"Sent {sent} spam transactions")
        return self.analyze_results()

    async def connection_flood(self, connections_per_second):
        """Simulate connection flood attack"""
        print(f"Starting connection flood: {connections_per_second}
connections/s")

        # Try to open many connections
        import socket
```

```python
        successful = 0
        failed = 0

        for i in range(connections_per_second * 10):  # 10 second burst
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(1)

            target = random.choice(self.targets)
            host, port = target.replace("http://", "").split(":")

            try:
                s.connect((host, int(port)))
                successful += 1
                # Send garbage data
                s.send(b"GET / HTTP/1.1\r\n" + b"X" * 1000 + b"\r\n\r\n")
                s.close()
            except:
                failed += 1

            if i % 100 == 0:
                print(f"  Connections: {successful} success, {failed} failed")

        return successful, failed

    def analyze_results(self):
        """Check if node defended against attacks"""
        # Count how many spam transactions were rejected
        rejected = sum(1 for r in self.results if r in [400, 401, 429])
        accepted = sum(1 for r in self.results if r == 200)

        print(f"\nAttack Results:")
        print(f"  Rejected (good!): {rejected}")
        print(f"  Accepted (bad!): {accepted}")

        # Good defense: >95% rejection
        return rejected / len(self.results) > 0.95 if self.results else False

# Run attacks
if __name__ == "__main__":
    targets = [
        "http://your-node-1:8080",
        "http://your-node-2:8080"
    ]

    simulator = AttackSimulator(targets)

    # Run spam attack
    asyncio.run(simulator.spam_transactions(
        transactions_per_second=100,
        duration_seconds=30
    ))

    # Run connection flood
```

```
    asyncio.run(simulator.connection_flood(50))
```

2.2 Implement Rate Limiting (Week 3)

Add to `src/network/rate_limiter.rs`:

```
rust
```

```
pub struct RateLimiter {
    // Track requests per IP/peer
    buckets: HashMap<String, TokenBucket>,
    // Global limits
    global_limits: RateLimits,
}

#[derive(Clone)]
pub struct RateLimits {
    pub requests_per_second: u32,
    pub connections_per_minute: u32,
    pub data_per_second: u64,  // bytes
}

impl RateLimiter {
    pub fn new(limits: RateLimits) -> Self {
        Self {
            buckets: HashMap::new(),
            global_limits: limits,
        }
    }
```

```rust
    pub fn check_rate_limit(&mut self, key: &str, request_size: usize) ->
Result<(), RateLimitError> {
        let bucket = self.buckets.entry(key.to_string()).or_insert_with(|| {
            TokenBucket::new(
                self.global_limits.requests_per_second,
                self.global_limits.data_per_second,
            )
        });

        // Check request count limit
        if !bucket.try_take_tokens(1) {
            return Err(RateLimitError::TooManyRequests);
        }

        // Check data size limit
        if !bucket.try_take_bytes(request_size as u64) {
            return Err(RateLimitError::TooMuchData);
        }

        Ok(())
    }

    pub fn check_connection_rate(&mut self, ip: &str) -> bool {
        // Use separate counter for connections
        let connection_key = format!("conn:{}", ip);
        let count = self.connection_counts.entry(connection_key).or_insert(0);

        if *count >= self.global_limits.connections_per_minute {
            false
        } else {
            *count += 1;
            true
        }
    }
}

// Integrate with network handler
impl P2PManager {
    async fn handle_incoming_message(&mut self, source: PeerId, message:
Vec<u8>) {
        // Check rate limits
        let source_ip = self.get_peer_ip(&source);
        if let Err(e) = self.rate_limiter.check_rate_limit(&source_ip, 
message.len()) {
            tracing::warn!("Rate limiting peer {}: {}", source, e);
            self.disconnect_peer(source).await;
            return;
        }

        // Process message if not rate limited
        // ...
    }

}
```

2.3 Add Proof-of-Work for Connection (Week 4)

Implement lightweight PoW to prevent connection spam:

`src/network/pow_challenge.rs`:

```rust
pub struct ConnectionChallenge {
    difficulty: u32,  // Adjust based on attack level
}

impl ConnectionChallenge {
    pub fn new(base_difficulty: u32) -> Self {
        Self {
            difficulty: base_difficulty,
        }
    }

    pub fn generate_challenge(&self) -> (Vec<u8>, u32) {
        // Create random challenge
        let challenge: [u8; 32] = rand::random();
        (challenge.to_vec(), self.difficulty)
    }

    pub fn verify_solution(&self, challenge: &[u8], nonce: u64, difficulty: 
u32) -> bool {
        // Check if hash(challenge + nonce) has enough leading zeros
        let mut hasher = blake3::Hasher::new();
        hasher.update(challenge);
        hasher.update(&nonce.to_le_bytes());
        let hash = hasher.finalize();

        // Count leading zero bits
        let leading_zeros = hash.as_bytes()[..4]
            .iter()
            .map(|b| b.leading_zeros())
            .sum::<u32>();

        leading_zeros >= difficulty
    }
}

// Use in connection handler
impl P2PManager {
    async fn handle_new_connection(&mut self, peer_addr: SocketAddr) {
        // Only require PoW if we have many connections
        if self.active_connections > 100 {
            let (challenge, difficulty) =
self.pow_challenge.generate_challenge();

            // Send challenge to peer
            self.send_to_peer(peer_addr, NetworkMessage::PoWChallenge {
                challenge,
                difficulty,
            }).await;

            // Wait for solution (timeout: 5 seconds)
            match tokio::time::timeout(
                Duration::from_secs(5),
                self.wait_for_pow_solution(peer_addr)
            ).await {
                Ok(true) => self.accept_connection(peer_addr).await,
                _ => self.reject_connection(peer_addr).await,
            }
        } else {
            // Accept without PoW under normal load
            self.accept_connection(peer_addr).await;
        }
    }

}
```

Month 3: Performance Tuning & Final Testing

3.1 Load Testing (Week 1-2)

Create high-load test `tests/load_test.py`:

```python
import asyncio
import aiohttp
import time
import statistics
from tqdm import tqdm

class LoadTester:
    def __init__(self, node_urls):
        self.nodes = node_urls
        self.results = []

    async def test_transaction_throughput(self, duration=60):
        """Measure maximum TPS the network can handle"""
        print(f"Testing transaction throughput for {duration} seconds...")

        stats = {
            'submitted': 0,
            'confirmed': 0,
            'latencies': [],
            'errors': 0
        }

        # Generate test transactions
        test_transactions = self.generate_test_transactions(1000)

        # Start sending
        start_time = time.time()
        with tqdm(total=duration) as pbar:
            while time.time() - start_time < duration:
                # Send batch of transactions
                batch_size = 100
                tasks = []

                for i in range(batch_size):
```

```
                tx = test_transactions[i % len(test_transactions)]
                node = self.nodes[i % len(self.nodes)]
                tasks.append(self.submit_transaction(node, tx))

            # Submit batch
            batch_results = await asyncio.gather(*tasks,
return_exceptions=True)

            # Update stats
            for result in batch_results:
                if isinstance(result, dict):
                    stats['submitted'] += 1
                    if result.get('confirmed'):
                        stats['confirmed'] += 1
                        stats['latencies'].append(result['latency'])
                else:
                    stats['errors'] += 1

            # Update progress
            pbar.update(1)
            pbar.set_description(f"TPS:
{stats['submitted']/(time.time()-start_time):.1f}")

        # Calculate final stats
        total_time = time.time() - start_time
        tps = stats['submitted'] / total_time

        print(f"\nResults:")
        print(f"  Duration: {total_time:.1f}s")
        print(f"  Transactions submitted: {stats['submitted']}")
        print(f"  Transactions confirmed: {stats['confirmed']}")
        print(f"  Average TPS: {tps:.1f}")
        print(f"  Error rate:
{(stats['errors']/stats['submitted']*100):.1f}%")

        if stats['latencies']:
            print(f"  Average latency:
{statistics.mean(stats['latencies']):.3f}s")
            print(f"  P95 latency: {statistics.quantiles(stats['latencies'],
n=20)[18]:.3f}s")

        return tps
```

```
    async def test_scalability(self, max_nodes=10):
        """Test how performance scales with more nodes"""
        print("Testing scalability with increasing node count...")

        scalability_results = []

        for node_count in range(1, max_nodes + 1):
            # Use subset of nodes
            test_nodes = self.nodes[:node_count]
            print(f"\nTesting with {node_count} nodes...")

            # Run throughput test
            tps = await self.test_transaction_throughput(duration=30)

            scalability_results.append({
                'nodes': node_count,
                'tps': tps
            })

        # Plot results (optional)
        print("\nScalability Results:")
        for result in scalability_results:
            print(f"  {result['nodes']} nodes: {result['tps']:.1f} TPS")

        return scalability_results

# Run load tests
if __name__ == "__main__":
    nodes = [f"http://node-{i}:8080" for i in range(5)]
    tester = LoadTester(nodes)

    # Test 1: Maximum throughput
    asyncio.run(tester.test_transaction_throughput(duration=120))

    # Test 2: Scalability
    asyncio.run(tester.test_scalability(max_nodes=5))
```

3.2 Memory & Resource Optimization (Week 3)

Add memory monitoring and cleanup:

`src/network/resource_manager.rs`:

```rust
pub struct ResourceManager {
    // Track memory usage
    memory_usage: usize,
    max_memory_mb: usize,

    // Track open connections
    active_connections: usize,
    max_connections: usize,

    // Garbage collection timer
    last_gc: Instant,
}

impl ResourceManager {
    pub fn new(max_memory_mb: usize, max_connections: usize) -> Self {
        Self {
            memory_usage: 0,
            max_memory_mb,
            active_connections: 0,
            max_connections,
            last_gc: Instant::now(),
        }
    }

    pub fn check_memory_usage(&mut self, additional_bytes: usize) ->
Result<(), ResourceError> {
        let new_total = self.memory_usage + additional_bytes;
        let max_bytes = self.max_memory_mb * 1024 * 1024;

        if new_total > max_bytes {
            // Try to free memory
            self.garbage_collect();

            // Check again
            if self.memory_usage + additional_bytes > max_bytes {
                return Err(ResourceError::OutOfMemory);
            }
        }
        self.memory_usage = new_total;
        Ok(())
    }

    pub fn garbage_collect(&mut self) {
        // Clear old messages from mempool
        self.mempool.evict_old(Instant::now() - Duration::from_secs(30));

        // Clear old peer info
        self.peer_store.remove_stale();

        // Clear expired rate limit buckets
        self.rate_limiter.cleanup();

        // Update memory usage
        self.update_memory_stats();

        self.last_gc = Instant::now();
        tracing::info!("Garbage collected. Memory: {}MB", self.memory_usage /
(1024*1024));
    }

    pub fn monitor_resources(&self) -> ResourceMetrics {
        ResourceMetrics {
            memory_mb: self.memory_usage / (1024 * 1024),
            connections: self.active_connections,
            uptime: self.start_time.elapsed().as_secs(),
            last_gc_secs_ago: self.last_gc.elapsed().as_secs(),
        }
    }
}

// Integrate with node
impl Node {
    async fn check_resources(&mut self) {
        // Run every 30 seconds
        loop {
            tokio::time::sleep(Duration::from_secs(30)).await;

            let metrics = self.resource_manager.monitor_resources();

            // Log warning if near limits
            if metrics.memory_mb > self.resource_manager.max_memory_mb * 0.8 {
                tracing::warn!("High memory usage: {}MB", metrics.memory_mb);
                self.resource_manager.garbage_collect();
            }

            if metrics.connections > self.resource_manager.max_connections *
0.8 {
                tracing::warn!("High connection count: {}",
metrics.connections);
                self.disconnect_least_useful_peers().await;
            }

            // Export metrics for monitoring
            self.export_metrics(metrics);
        }
    }
}
```

3.3 Final Integration Testing (Week 4)

Create comprehensive test suite:

`tests/integration_test.rs`:

```rust
#[tokio::test]
async fn test_full_network_scenario() {
    // Scenario: Network partition recovery

    println!("=== Test: Network Partition Recovery ===");

    // Start 5 nodes
    let mut nodes = Vec::new();
    for i in 0..5 {
        let node = TestNode::new(i).await;
        nodes.push(node);
    }

    // Let them connect
    tokio::time::sleep(Duration::from_secs(10)).await;
    // Create partition (nodes 0-2 and 3-4 can't see each other)
    println!("Creating network partition...");
    nodes[0].block_peers(&[nodes[3].peer_id(), nodes[4].peer_id()]);
    nodes[1].block_peers(&[nodes[3].peer_id(), nodes[4].peer_id()]);
    nodes[2].block_peers(&[nodes[3].peer_id(), nodes[4].peer_id()]);

    // Send transactions in partition A
    println!("Sending transactions in partition A...");
    for i in 0..10 {
        nodes[0].send_transaction(TestTransaction::new(i)).await;
    }

    // Heal partition
    println!("Healing partition...");
    nodes[0].unblock_all_peers();
    nodes[1].unblock_all_peers();
    nodes[2].unblock_all_peers();

    // Wait for sync
    println!("Waiting for sync...");
    tokio::time::sleep(Duration::from_secs(30)).await;

    // Verify all nodes have all transactions
    println!("Verifying consistency...");
    for (i, node) in nodes.iter().enumerate() {
        let tx_count = node.get_transaction_count().await;
        println!("  Node {}: {} transactions", i, tx_count);
        assert_eq!(tx_count, 10, "Node {} missing transactions", i);
    }

    println!("✓ Partition recovery test passed!");
}

#[tokio::test]
async fn test_high_load_with_mixers() {
    println!("=== Test: High Load with Mixers ===");

    // Start network with 1 bootstrap, 3 regular nodes, 2 mixer nodes
    let bootstrap = BootstrapNode::new().await;
    let mut nodes = Vec::new();
    let mut mixers = Vec::new();
    for i in 0..3 {
        nodes.push(RegularNode::new(i, &bootstrap).await);
    }

    for i in 0..2 {
        mixers.push(MixerNode::new(i, &bootstrap).await);
    }

    // Send 1000 transactions through mixers
    println!("Sending 1000 transactions through mixers...");
    let start = Instant::now();

    let mut tasks = Vec::new();
    for i in 0..1000 {
        let mixer = &mixers[i % mixers.len()];
        let node = &nodes[i % nodes.len()];

        tasks.push(tokio::spawn(async move {
            let tx = TestTransaction::new(i);
            mixer.process_transaction(tx).await;
            node.await_transaction(i).await;
        }));
    }

    // Wait for all
    futures::future::join_all(tasks).await;

    let elapsed = start.elapsed();
    let tps = 1000.0 / elapsed.as_secs_f64();

    println!("  Time: {:.2}s", elapsed.as_secs_f64());
    println!("  TPS: {:.1}", tps);

    // Verify all transactions received
    for node in &nodes {
        let received = node.get_received_transactions().await;
        println!("  Node received: {} transactions", received.len());
        assert!(received.len() >= 900, "Node missing too many transactions");
    }

    println!("✓ High load test passed!");
}
```

Phase 3 Completion Checklist:

- Network runs on real internet across multiple regions
- Monitoring dashboard shows real-time metrics
- Network survives DoS/spam attacks (95%+ rejection)
- Rate limiting prevents abuse
- Resource usage stays within limits under load
- Achieves target TPS (start with 1k, aim for 10k+)
- All nodes stay in sync during network partitions
- Memory doesn't leak over 24+ hour runs

Final Recommendations for Going Live:

1. Start Small: Begin with 5-10 nodes on testnet
2. Monitor Closely: Watch metrics dashboard 24/7 initially
3. Gradual Scaling: Add nodes slowly (5 per week)
4. Keep Backups: Regular snapshots of node state
5. Community Engagement: Invite others to run nodes early
6. Bug Bounty: Offer rewards for finding security issues

Common Issues & Solutions:

Problem	Solution
Nodes can't connect through NAT	Enable libp2p autonat + relay
High memory usage	Implement garbage collection, increase `max_memory_mb`

Slow transaction propagation	Adjust gossip parameters, enable compression
Some nodes fall behind	Increase `max_peers`, check network latency
Mixer nodes overloaded	Add more mixer nodes, implement load balancing

Next Steps After Phase 3:

1. External security audit
2. Public testnet launch
3. Community node operator program
4. Mainnet preparation

Total Estimated Time: 6-9 months for networking (Phases 1-3 combined with TEE work)

Remember: Building robust P2P networking is iterative. You'll find and fix issues as you test. The key is starting with a solid foundation (libp2p) and building up gradually.