1. Real TEE Integration & Reducing Reliance on Mocks/Simulation

Current state: The TEE runtimes (SGX/SEV) are excellent prototypes with simulation modes that always "pass" attestation and use mock reports. This is great for development/CI, but it bypasses the core security guarantee: verifiable remote attestation with real hardware quotes and certificate chains.

Why this is critical: The whitepaper's privacy claims (k-anonymity >1M, blind validation, secure aggregation) depend on **verifiable** TEE execution. Simulation mode is fine for testing logic but not for security audits or mainnet trust.

Step-by-Step Plan to Address It:

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

- **Acquire test hardware**:
    - Intel SGX: Need a recent Intel CPU with SGX enabled (e.g., 11th-gen+ desktop with Flex enabled in BIOS, or Azure DCsv3 VMs).
    - AMD SEV-SNP: Need EPYC 7003+ series (Milan) or newer; easiest via AWS EC2 or dedicated server.
    - Start with SGX (more mature ecosystem) unless AMD is a priority.
- **Integrate production libraries** (replace simulation paths):
    - For **SGX**: Use Intel's **DCAP** (Data Center Attestation Primitives) fully.
        - Current code already references sgx_dcap_ql and sgx_dcap_quoteverify—enable the real paths.
        - Install Intel SGX DCAP SDK on dev machines (Linux preferred for real quotes; Windows has limitations).
        - Implement real quote generation in SGXRuntime::local_attest using sgx_ql_get_quote.
        - Verify remote quotes with full chain (PCK cert → intermediate → root) using sgx_dcap_quoteverify.
    - For **SEV-SNP**: Use AMD's sev crate more deeply.
        - Replace mock reports with real Firmware::get_attestation_report.
        - Verify signature chain (VCEK → ASK/ARK) using AMD's online services or cached roots.
- **Configuration flag**: Add a Cargo feature or env var (e.g., RUSTFLAGS="--cfg tee_real") to toggle simulation vs real mode. Default to simulation for CI, real for release/testing.

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

- **Attestation verification**:
    - In attestation::verify_attestation_report, implement full chain validation + freshness checks (timestamp ±5min).

    - Pin expected measurements (MRSIGNER/MRENCLAVE) in code or config—hardcode initial values, make updatable via governance.
    - Fail hard in real mode if quote invalid (current sim always passes).
- **Predictor real model loading**:
    - Current fallback is deliberate for no-model testing.
    - Download/distribute the real 1.8MB distilled .pt or ONNX model (via IPFS/Arweave or bundled).
    - In LstmLoadPredictor and consensus predictor, load via tch-rs or tract-onnx without fallback.
    - Add model hash pinning + consensus vote for updates.
- **Testing strategy**:
    - Unit tests: Keep simulation mode.
    - Integration tests: Add new suite requiring real hardware (or cloud VMs) – use #[cfg(tee_real)].
    - Remote attestation end-to-end: Run two nodes (one in TEE VM), submit tx, verify quote chain.

Phase 3: Security & Auditing (Ongoing)

- Audit the real paths (especially quote verification logic).
- Add logging/metrics for attestation failures.

Estimated effort: 4-6 months part-time for one developer familiar with SGX/SEV.

2. Stubbed Networking (No Real Peer Connections, DHT/Gossip Not Battle-Tested)

Current state: DHT, gossip, and mempool are solid prototypes with in-memory structures and mocks. No actual socket listening, peer discovery, or message relay—everything is local/single-node.

Why this is critical: Infinite scalability and privacy (mixing, gossip propagation) require robust P2P. Stubbed networking works for unit tests but not for multi-node synchronization or real TPS.

Step-by-Step Plan to Address It:

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

- **Best option**: Switch to **libp2p-rs** (the de-facto Rust P2P library, used by Substrate, Filecoin, etc.).
    - It provides Kademlia DHT, Gossipsub, QUIC/TCP/Noisesec transports out-of-the-box.
    - Your current DHT/gossip designs map almost 1:1.
    - Benefits: Battle-tested, supports NAT traversal, relay circuits, autonat.
- **Migration steps**:

    - Replace custom DhtManager with libp2p's Kademlia.
    - Replace custom GossipManager with libp2p's Gossipsub (topic-based, mesh, scoring).
    - Use libp2p's Swarm for connection management.
    - Keep encrypted mempool logic—plug it into gossip topics (e.g., /nerv/tx/1).
- **Transports**: Start with QUIC + Noise (PQ-secure), fallback TCP.

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

- **Bootstrapping**: Hardcode initial bootstrap nodes (your testnet nodes), then discover via DHT.
- **Peer scoring**: Integrate libp2p's built-in scoring with your reputation system.
- **Message propagation**:
    - Broadcast blocks/transactions via gossip.
    - Query DHT for shard peers or mixer hops.
- **Encrypted mempool integration**: On receiving gossip tx message, decrypt/verify attestation before adding to mempool.
- **Multi-node testing**:
    - Spin up 5-10 local nodes (different ports) in integration tests.
    - Use Docker Compose for reproducible multi-node setups.
    - Measure propagation latency, message loss.

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

- Run on testnet with real internet (e.g., 20+ nodes on cloud VMs).
- Fuzz networking (property-based testing with proptest).
- Add DoS protection (rate limiting, proof-of-work for new connections).
- Metrics: Peer count, message latency, DHT lookup success rate.

Estimated effort: 6-9 months, potentially with a second developer focused on networking.

Final Recommendations

- **Prioritization**: Fix TEE first (core security/privacy promise), then networking (scalability).
- **Milestones**:
    - Q1-Q2 2026: Real TEE + basic multi-node networking.
    - Q3-Q4 2026: Full P2P + testnet launch.
    - 2027: Optimization, audits, benchmarks toward 1M+ TPS.
- **Team/Resources**: Consider contributors familiar with libp2p (common in Substrate ecosystem) and SGX/DCAp (Intel has good docs).
- **Testing**: Add CI jobs on cloud VMs with real TEE for nightly checks.

Detailed Guide to Phase 1: Hardware & Library Setup for Real TEE Integration

Why This Phase Matters (Simple Explanation)

The current NERV code uses "simulation mode" for Trusted Execution Environments (TEEs) like Intel SGX and AMD SEV-SNP. This means it pretends to do secure attestation (proving the code runs in protected hardware) but always says "yes, it's secure" without checking real hardware. This is fine for testing code logic but **not secure** for real use—the whitepaper's privacy guarantees (no visible addresses/amounts, >1M anonymity set, secure FL aggregation) rely on **real, verifiable hardware quotes**.

Phase 1 replaces mocks with real hardware checks. We'll focus on **one TEE at a time**—start with Intel SGX (more mature tools, easier for beginners). AMD SEV-SNP is similar but hardware is rarer.

Estimated Time: 1-2 months part-time (mostly waiting for cloud setup/accounts, installing, testing).

Who This is For: Non-experts—follow commands exactly. If stuck, copy errors into Google/search.

Recommendation for Beginners: Use **cloud VMs** (rented servers) instead of local hardware. Local SGX/SEV requires specific expensive CPUs + BIOS tweaks (high failure risk). Cloud is easier, cheaper for testing (~$0.50-2/hour).

Step 1: Choose Your Path (10 minutes)

- **Option A: Cloud (Easiest, Recommended)**
    - Intel SGX: Azure Confidential VMs (DCsv3 series).
    - AMD SEV-SNP: AWS EC2 (specific instances).
    - Cost: $50-200 for testing (pay-as-you-go).
- **Option B: Local Hardware (Advanced, Risky)**
    - SGX: Desktop with 11th-gen+ Intel CPU (e.g., i7-11700) + SGX enabled in BIOS.
    - SEV-SNP: Server with AMD EPYC Milan+ (expensive, rare).
    - Skip if unsure—use cloud.

We'll detail **Cloud first** (SGX on Azure), then notes for AWS SEV-SNP and local.

Step 2: Set Up Cloud Account & VM for Intel SGX (Azure – 1-2 hours + approval wait)

Azure has ready SGX VMs—no BIOS hassle.

1. Create Azure account:
    - Go to https://azure.microsoft.com/free/ (free trial $200 credit).
    - Sign up with email/credit card (no charge until used).
2. Enable Confidential Computing:
    - Confidential VMs need approval (quick, free).

    - Search Google: "Azure confidential computing access" → Follow Microsoft form (request DCsv3 series access).
    - Wait 1-7 days for approval email.
3. Launch SGX VM (once approved):
    - Log in to https://portal.azure.com.
    - Search "Create a virtual machine".
    - Basics:
        - Subscription: Free trial/default.
        - Resource group: Create new (e.g., "nerv-tee").
        - VM name: "nerv-sgx-test".
        - Region: One with SGX (e.g., East US, West Europe—check availability).
    - Size: Search "DC" → Choose DC2s_v3 or higher (2 vCPU, 8GB RAM minimum).
    - Image: Ubuntu Server 22.04 LTS (common for SGX).
    - Authentication: SSH key (generate via ssh-keygen on your computer).
    - Leave defaults, review + create (10-20 minutes).
4. Connect to VM:
    - In Azure portal → Your VM → Connect → SSH.
    - Copy command (e.g., ssh username@ip).
    - Run in your local terminal.

Step 3: Install Real SGX DCAP Libraries on the VM (30-60 minutes)

Inside the SSH session (Ubuntu):

1. Update system:
   text
   ```
   sudo apt update && sudo apt upgrade -y
   ```

Add Intel repo (latest as of 2026):
text
```
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
```

2. sudo apt update
3. Install DCAP packages:
   text
   ```
   sudo apt install -y libsgx-dcap-ql libsgx-dcap-quote-verify libsgx-dcap-default-qpl
   ```
4. Install driver (if needed):
   text
   ```
   sudo apt install -y sgx-dcap-pce sgx-dcap-ql-dev
   ```
5. Verify:
   text
   ```
   dcap-quote-verify --version
   ```
    - Should show version.

(From Intel's official guide: https://cc-enabling.trustedservices.intel.com/intel-sgx-sw-installation-guide-linux/)

Step 4: AMD SEV-SNP Alternative (AWS Cloud – Similar Time)

If preferring SEV:

1. AWS account: https://aws.amazon.com/free/.
2. Launch instance:
    - Console → EC2 → Launch instance.
    - Name: "nerv-sev-test".
    - AMI: Ubuntu 24.04.
    - Instance type: Search "SEV-SNP" → e.g., M6a or C7a with SEV-SNP.
    - Enable SEV-SNP: Advanced → CPU options → Check "AMD SEV-SNP".
    - Key pair: Create/download SSH key.

Install tools:
text
```
sudo apt update
sudo apt install -y linux-modules-extra-aws
# Build snpguest for attestation
git clone https://github.com/AMDESE/sev-guest
cd sev-guest
make
```

3. sudo cp snpguest /usr/bin/

Step 5: Local Hardware Notes (If Insisting)

- SGX: Buy compatible CPU/motherboard → Enter BIOS (Del/F2) → Enable "SGX" and "Flex Mode".
- Then same install commands as Azure VM.

Step 6: Integrate into NERV Code (1-2 weeks coding)

Back on your local machine (clone repo):

1. Open code: Files like src/privacy/tee/sgx.rs already reference DCAP crates.
2. Replace simulation:
    - In sgx.rs: Comment out mock sections, enable real quote gen with sgx_ql_get_quote().

Example (pseudo):
Rust
```
#[cfg(feature = "tee_real")]
use sgx_dcap_ql::quote;
// Real quote code
```

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

- *// Mock code*

Add Cargo feature: In Cargo.toml:
text
```
[features]
```

3. tee_real = []
4. Build/test:
   text
   ```
   cargo build --features tee_real
   ```
   - On real VM: Copy code via SCP, build there.
5. Test: Run node on VM → Check logs for real quote generation.

—----------

Detailed Guide to Phase 2: Replace Mocks with Real Behavior in TEE Integration

Quick Recap & Why This Phase

In Phase 1, you set up real hardware (cloud VM or local) and installed genuine SGX/SEV libraries. Now, we replace the "pretend" (mock/simulation) code with **real** attestation logic. This means:

- The node will generate and verify actual hardware quotes (proofs that code runs securely in TEE).
- It will **reject** invalid/fake attestations (no more auto-pass).
- We'll add real model loading (no fallbacks) and pinned security checks.

This makes the privacy features (mixing, blind validation, FL aggregation) trustworthy.

Estimated Time: 2-3 months part-time (mostly coding + testing on your Phase 1 VM).

Assumptions:

- You have the NERV repo cloned locally.
- You have a working SGX VM from Phase 1 (Azure recommended).
- Focus on SGX first (SEV similar).
- You're comfortable with basic Rust editing (use VS Code).

Step 1: Prepare Your Development Setup (1-2 days)

1. Install VS Code (free editor): https://code.visualstudio.com/
   - Extensions: "rust-analyzer", "CodeLLDB" for Rust support.

On your local machine:
text
```
cd nerv  # Your repo
```

2. git pull # Get latest
3. Enable real mode feature (from Phase 1):

In Cargo.toml, ensure:
text
```
[features]
```

    - tee_real = []
    - Build with real mode: cargo build --features tee_real
4. Copy code to VM for real testing:
    - Use SCP: scp -r . username@vm-ip:/home/username/nerv
    - Or Git clone directly on VM.

Step 2: Implement Real Attestation Verification (1-2 weeks)

Focus file: src/privacy/tee/attestation.rs (and sgx.rs/sev.rs).

1. Open attestation.rs in VS Code.
2. Find mock sections (search "mock" or "simulation"):
    - Replace with real verification.
3. Add full chain + freshness check:

Example code to add/replace in verify_attestation_report:
Rust
```rust
#[cfg(feature = "tee_real")]
pub fn verify_attestation_report(report: &AttestationReport) -> Result<bool, AttestationError> {
    // 1. Check timestamp freshness (±5 minutes)
    let now = SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();
    if (now as i64 - report.timestamp as i64).abs() > 300 {  // 5 min
        return Err(AttestationError::TimestampError);
    }

    // 2. Verify quote signature chain (Intel DCAP)
    let verify_result = sgx_dcap_quoteverify::verify_quote(&report.quote);
    if verify_result != 0 {
        return Err(AttestationError::QuoteVerification(format!("DCAP error {}", verify_result)));
    }

    // 3. Pin expected measurement (hardcode initial, update via governance later)
    let expected_mr = [0u8; 48];  // REPLACE with your enclave's real MRSIGNER/MRENCLAVE
    // Get from build: Run enclave in debug, log measurement
    if report.measurement != expected_mr {
        return Err(AttestationError::MeasurementMismatch);
```

```
    }

    Ok(true)
}

#[cfg(not(feature = "tee_real"))]
pub fn verify_attestation_report(_report: &AttestationReport) -> Result<bool, AttestationError> {
    Ok(true)  // Old mock
```

        - }
    4. Pin measurements:
        - Build enclave on VM: cargo build --features tee_real.
        - Run and log the measurement (add println!("Measurement: {:x?}", report.measurement);).
        - Copy the 48-byte value into code as expected_mr.
    5. Make it fail hard:
        - In node startup/privacy manager: If verification fails → log error and shutdown/halt TEE tasks.

Step 3: Real Model Loading (No Fallbacks) (3-5 days)

Files: src/consensus/predictor.rs, src/sharding/lstm_predictor.rs.

1. Remove fallback dummy prediction:

In Predictor::new: If model load fails → return Error instead of fallback.
Rust
```
pub fn new(config: &ConsensusConfig) -> Result<Self> {
    let model = load_model(&config.predictor_model_path)?;
    // No fallback
    Ok(Self { model, fallback: false })
```

        - }
    2. Pin model hashes:

Add BLAKE3 hash check:
Rust
```
let expected_hash = hex::decode("actual_hash_from_ipfs").unwrap();  // Get from trained model file
let actual_hash = blake3::hash(&fs::read(&path)?);
if actual_hash.as_bytes() != expected_hash {
    return Err(NervError::Model("Hash mismatch".into()));
```

        - }
    3. Download/distribute models:
        - In README/docs: Instructions to fetch from IPFS/Arweave (pinned hashes).
        - Consensus vote for updates: Stub a governance proposal for new hash.

Step 4: Testing Strategy (2-4 weeks)

Separate tests for safety.

1. Keep unit tests in simulation:
    - Run normally: cargo test (uses mock, fast).
2. New integration tests for real hardware:
    - Add folder tests/integration_tee_real/.

Mark with attribute:
Rust
```
#[cfg(feature = "tee_real")]
#[tokio::test]
async fn test_real_attestation() {
   // Generate quote, verify
   assert!(PrivacyManager::new().local_attest(&data).is_ok());
```

    - }
    - Run on VM: cargo test --features tee_real.
3. End-to-end remote attestation:
    - Spin 2 VMs (Node A in TEE, Node B normal).
    - Node A generates quote during tx/mixer.
    - Node B verifies chain → accepts/rejects.
    - Manual test: Run nodes, submit test tx, check logs ("Attestation verified" or "failed").
4. Common issues & fixes:
    - Quote error: Check DCAP services running (ps aux | grep dcap).
    - Measurement mismatch: Rebuild enclave after code changes.
    - Timestamp error: Sync VM clock (sudo ntpdate pool.ntp.org).

Success Signs:

- Logs show real quote verification.
- Invalid/mocked quote rejected.
- Models load only if hash matches.

—---------

Detailed Guide to Phase 3: Security & Auditing for Real TEE Integration

Quick Recap & Why This Phase

Phases 1 and 2 gave you real hardware (VM) and replaced mocks with genuine attestation (quotes, verification, pinned measurements, no fallbacks). Now, we focus on **making it secure and trustworthy**:

- Audit the new real code paths (find bugs/mistakes).

- Add monitoring (logs/metrics) for attestation issues.
- Prepare for external audits/community review.
- Ongoing: Keep improving as threats evolve.

This phase is "ongoing" because security never ends—blockchain projects like NERV (privacy-focused) need constant vigilance.

Estimated Time: Ongoing (3-6 months initial push, then maintenance). Start with 2-4 weeks for basics.

Goal: Your node rejects fake/invalid attestations, logs problems clearly, and is ready for code review/audit. This aligns with whitepaper's "full NIST post-quantum security from genesis" and TEE-bound privacy.

Step 1: Initial Self-Audit of Real Paths (1-2 weeks)

Review your changes manually—non-experts can do this with checklists.

1. List changed files:
    - src/privacy/tee/attestation.rs
    - src/privacy/tee/sgx.rs (or sev.rs)
    - Any new features/flags in Cargo.toml or startup code.
2. Checklist (go line-by-line in VS Code):
    - Search "#[cfg(feature = "tee_real")]": Ensure real code runs only in real mode.
    - Check error handling: Every real verification returns Err on failure (no silent pass).
    - Timestamp check: ±5 minutes max (prevent replay attacks).
    - Measurement pinning: Hardcoded value matches your enclave build (no placeholder zeros).
    - Quote verification: Calls DCAP library and checks return code != 0 → error.
    - No leaks: No printing secret keys/quotes in logs.
3. Manual tests on VM:
    - Tamper quote: Edit a byte in report → Run node → Should error/shutdown.
    - Wrong measurement: Change pinned value → Rebuild → Should fail.
    - Old timestamp: Mock old time (advanced: date command on VM) → Fail.
4. Document: Create TEE_AUDIT.md in repo root:
    - List checks.
    - Screenshots/logs of pass/fail tests.

Step 2: Add Logging & Metrics for Attestation (3-5 days)

Make problems visible—nodes should alert on failures.

1. Use existing tracing (repo has tracing crate).

In attestation.rs verify function:
Rust
```
if verify_result != 0 {
    tracing::error!("Quote verification failed: DCAP code {}", verify_result);
    // Optional: metrics increment
    metrics::ATTETSTATION_FAILURES.inc();
    return Err(...);
} else {
    tracing::info!("Attestation verified successfully");
    metrics::ATTETSTATION_SUCCESS.inc();
```

        - }
    2. Add metrics (repo has Prometheus in utils/metrics.rs):

New counters/gauges:
Rust
```
lazy_static! {
    static ref ATTESTATION_SUCCESS: Counter = register_counter!(... "nerv_tee_attestation_success");
    static ref ATTESTATION_FAILURE: Counter = register_counter!(... "nerv_tee_attestation_failure");
    static ref ATTESTATION_TYPE: GaugeVec = register_gauge_vec!(... "nerv_tee_type", &["type"]);  // sgx/sev
```

        - }
        - Increment on success/failure.
    3. Log details safely:
        - Log quote hash (not full quote) for debugging.
        - Error types: "Measurement mismatch", "Expired timestamp", etc.
    4. Test: Run node → Submit fake tx → Check logs/metrics (Prometheus endpoint).

Step 3: Fuzzing & Automated Testing (1-2 weeks)

Find hidden bugs automatically.

1. Install cargo-fuzz (Rust fuzzer):
   text
   ```
   cargo install cargo-fuzz
   ```
2. Create fuzz target:

New file fuzz/fuzz_targets/attestation.rs:
Rust
```
#![no_main]
use libfuzzer_sys::fuzz_target;
use nerv::privacy::tee::attestation::verify_attestation_report;

fuzz_target!(|data: &[u8]| {
    // Create fake report from data
    let fake_report = AttestationReport { quote: data.to_vec(), ... };  // Fill minimally
```

```
let _ = verify_attestation_report(&fake_report);  // Should handle bad input gracefully
```

        - });
3. Run fuzzer on VM:
   text
   ```
   cargo fuzz run attestation -- -max_total_time=3600  # 1 hour
   ```
    - Fix crashes found.
4. Property tests (proptest crate if added):
    - Test: Invalid quotes always rejected, valid always accepted.

Step 4: Prepare for External Audit & Community Review (2-4 weeks)

Make code auditable.

1. Clean code:
    - Comments: Explain why each check (e.g., "// Prevent replay attacks").
    - Separate real/mock clearly with features.
2. Write audit report:
    - TEE_SECURITY_REPORT.md:
        - Threat model (replay, fake quotes, side-channels).
        - Mitigations (DCAP chain, pinning, freshness).
        - Test results.
3. Community:
    - Open GitHub issue/PR with changes.
    - Post on repo discussions/X: "TEE real mode implemented—review welcome!"
    - Invite security researchers (bug bounty stub).
4. Formal audit prep:
    - Scope: Only TEE paths.
    - Firms (future): Trail of Bits, Quantstamp (cost $50k+).
    - Start with free community audits.

Step 5: Ongoing Maintenance

- Update libraries: Monthly apt update on VM, rebuild.
- Monitor Intel/AMD advisories (subscribe to security lists).
- Governance: Add on-chain vote for new measurement hashes (when enclave updates).
- Rotate pins: Plan for enclave version bumps.

Success Signs:

- Node runs on real VM, verifies own quotes.
- Fake attestations logged/rejected.
- Metrics show successes, no unexplained failures.
- Code reviewed (at least self + 1 other).

You're done with TEE gaps! The node now has verifiable hardware security. Next could be networking (libp2p) or full testnet simulation.

—----------

Detailed Guide to Phase 1: Choose & Integrate a Mature P2P Stack for Networking

Quick Recap & Why This Phase

The current NERV code has excellent prototype networking (DHT for peer discovery, gossip for message spreading, mempool for txs), but it's all "in-memory" and local—no real sockets, connections, or internet peers. This works for single-node testing but not for real scalability/privacy (mixer hops, block sync across nodes, >1M TPS).

Phase 1 replaces custom stubs with **libp2p-rs**, the gold-standard Rust P2P library (used by Polkadot/Substrate, IPFS, Filecoin). It provides:

- Kademlia DHT (exact match for your custom one).
- Gossipsub (topic-based pubsub, meshes, scoring—better than custom gossip).
- Secure transports (QUIC + Noise for post-quantum readiness).
- NAT traversal, relays, autonat.

Benefits: Battle-tested, fewer bugs, faster progress to multi-node testnet.

Estimated Time: 1-2 months part-time (mostly adding dependencies, mapping code, basic tests).

Assumptions:

- You have the NERV repo cloned and can build (cargo build).
- Basic Rust knowledge (copy-paste code).
- Use VS Code for editing.

Step 1: Research & Confirm libp2p is Best Fit (1-2 days)

1. Read docs: https://docs.rs/libp2p/latest/libp2p/
    - Tutorials: https://github.com/libp2p/rust-libp2p/tree/master/examples
2. Compare to current code:
    - Your src/network/dht.rs → libp2p Kademlia.
    - src/network/gossip.rs → libp2p Gossipsub.
    - src/network/mod.rs → libp2p Swarm (central event loop).

Decision: Yes—perfect 1:1 mapping.

Step 2: Add libp2p Dependencies (1 day)

1. In repo root: Edit Cargo.toml.

Add under [dependencies]:
toml
```
libp2p = { version = "0.53", features = ["tcp", "quic", "noise", "yamux", "gossipsub", "kad", "identify", "ping",
"request-response", "autonat", "relay"] }
tokio = { version = "1", features = ["full"] }  # Already there, ensure full
futures = "0.3"
```

2. async-std = "1.12" # Optional for compat

Specific behaviors:
toml
```
libp2p-gossipsub = "0.47"
libp2p-kad = "0.45"
```

3. libp2p-identify = "0.43"
4. Build test:
   text
   cargo build
   - May take time (downloads). Fix errors (update versions if conflicts).

Step 3: Create libp2p Swarm Bootstrap (3-5 days)

Central piece: Replace custom managers with Swarm.

New file src/network/libp2p_swarm.rs:
Rust
```
use libp2p::{
    gossipsub, identify, kad::{self, Kademlia, KademliaConfig}, ping, relay, autonat,
    Swarm, SwarmBuilder, Noise, Yamux, Transport, Multiaddr, PeerId, identity,
    quic, tcp::TokioTcpConfig,
};
use libp2p::swarm::{SwarmEvent, NetworkBehaviour};
use tokio::sync::mpsc;

#[derive(NetworkBehaviour)]
struct NervBehaviour {
    gossipsub: gossipsub::Behaviour,
    kademlia: Kademlia<kad::store::MemoryStore>,
    identify: identify::Behaviour,
    ping: ping::Behaviour,
    relay: relay::Behaviour,
    autonat: autonat::Behaviour,
}

pub struct P2PManager {
    swarm: Swarm<NervBehaviour>,
```

```
    command_sender: mpsc::UnboundedSender<Command>,  // For sending messages
}

pub enum Command {
    PublishBlock(Vec<u8>),  // Example
    QueryDHT(PeerId),
}

impl P2PManager {
    pub async fn new(local_key: identity::Keypair) -> Self {
        let peer_id = PeerId::from(local_key.public());

        // Transports: QUIC + TCP fallback
        let transport = tcp::tokio::Transport::new(TokioTcpConfig::default().nodelay(true))
            .upgrade(libp2p::core::upgrade::Version::V1Lazy)
            .authenticate(Noise::new(&local_key).unwrap())
            .multiplex(Yamux::new())
            .or_transport(quic::tokio::Transport::new(quic::Config::new(&local_key)))
            .boxed();

        // Gossipsub config
        let gossipsub_config = gossipsub::ConfigBuilder::default()
            .heartbeat_interval(std::time::Duration::from_secs(10))
            .build().unwrap();

        // Kademlia config
        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);

        let mut 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()),
        };

        let mut swarm = SwarmBuilder::with_tokio_executor(transport, behaviour, peer_id).build();

        // Listen on all interfaces
        swarm.listen_on("/ip4/0.0.0.0/udp/0/quic-v1".parse().unwrap()).unwrap();
        swarm.listen_on("/ip4/0.0.0.0/tcp/0".parse().unwrap()).unwrap();
```

```
        let (tx, rx) = mpsc::unbounded_channel();
        tokio::spawn(async move {
            loop {
                tokio::select! {
                    event = swarm.select_next_some() => {
                        match event {
                            SwarmEvent::NewListenAddr { address, .. } => tracing::info!("Listening on {}", address),
                            // Handle other events (gossip messages, DHT queries)
                            _ => {}
                        }
                    }
                    command = rx.recv() => {
                        if let Some(cmd) = command {
                            // Handle publish, query
                        }
                    }
                }
            }
        });

        Self { swarm, command_sender: tx }
    }
```

1. }

Step 4: Map Custom Code to libp2p (1-2 weeks)

1. Replace DHT:
    - Old DhtManager → Use swarm.behaviour_mut().kademlia.
    - Queries: kademlia.get_record(), put_record().
2. Replace Gossip:
    - Old GossipManager → swarm.behaviour_mut().gossipsub.
    - Topics: gossipsub.subscribe(&topic).
    - Publish: gossipsub.publish(topic, data).
3. Mempool integration:
    - On gossip message: In SwarmEvent → If topic "/nerv/tx" → Add to mempool after TEE check.
4. Bootstrap:
    - Hardcode known peers: swarm.dial(multiaddr).
    - Or DHT bootstrap nodes.

Step 5: Basic Testing (1 week)

1. Run single node: cargo run --features real_network (add feature).
    - Logs: "Listening on /ip4/...".
2. Multi-node local:
    - Run two terminals: Different ports (edit listen addr).

    - Manually dial: Add code to dial known addr.
3. Verify: DHT lookup succeeds, gossip messages received.
