Implementation v1

Below is a **precise implementation blueprint**.
If you follow this step-by-step, you will end with a complete hackathon-ready product called **OmniQuant**.

This is structured like an engineering spec.

🔷 OMNIQUANT — IMPLEMENTATION BLUEPRINT

⚠️ Mandatory Legal & Ethical Disclaimer (Must Be In App + PPT)

You MUST include this in:

- App landing page
- README
- Final slide

> OmniQuant is a research and educational arbitrage detection simulator.
> All opportunities shown are theoretical and generated under simulated market conditions.
> No trades are executed. No financial returns are guaranteed.
> Users are responsible for independent verification before making financial decisions.

Never skip this.

PHASE 0 — Define Scope (Non-Negotiable Constraints)

You are building:

✔ Arbitrage detection engine
✔ Multi-hop cycle detection
✔ Profit simulation
✔ Risk scoring
✔ AI explanation layer
✔ Clean UI

You are NOT building:

✘ Real trade execution
✘ Blockchain interaction
✘ API key trading
✘ Live capital deployment

PHASE 1 — SYSTEM ARCHITECTURE

Final Architecture:

`Market Data (Simulated or API Pull) ↓ C++ Arbitrage Core ↓ Python Bridge (pybind11) ↓ Profit & Risk Simulator ↓ AI Explanation Layer ↓ Web Dashboard (FastAPI + Frontend)`

PHASE 2 — C++ CORE ENGINE

Step 1: Create Graph Model

Each token = Node
Each exchange rate = Directed edge

Weight formula:

$w = -\log(R \cdot (1 - fee))$

Where:

- R = exchange rate
- fee = exchange fee %

Why?
Multiplicative arbitrage becomes additive in log space.

Step 2: Implement Bellman-Ford

You must:

1. Initialize distances
2. Relax edges |V|-1 times
3. Check for negative cycle

4. Extract cycle path

Output format:

```
{ path: ["BTC", "ETH", "USDT", "BTC"], raw_profit: 0.0042 }
```

Step 3: Optimize for Performance

- Use adjacency list
- Use double precision
- Pre-allocate memory
- Avoid unnecessary object copying

Judges like clean memory handling.

PHASE 3 — Python Bridge

Use:

pybind11

Expose C++ function:

```
std::vector<std::string> detect_arbitrage(Graph g)
```

In Python:

```
import omniquant_core cycles = omniquant_core.detect_arbitrage(data)
```

PHASE 4 — Market Simulator (Python)

This is where realism happens.

Step 1: Slippage Model

Basic model:

effective_rate=rate×(1−volume/liquidity)effective_rate = rate \times (1 - volume / liquidity)effective_rate=rate×(1−volume/liquidity)

Step 2: Fee Model

Include:

- Trading fee
- Simulated withdrawal fee
- Latency penalty %

Step 3: Capital Constraint

Simulate:

- If user capital = $100
- Liquidity only supports $60
- Adjust final return

Step 4: Risk Score

Create simple scoring function:

Risk=f(volatility,liquidity,path_length)Risk = f(volatility, liquidity, path_length)Risk=f(volatility,liquidity,path_length)

Example:

- Path length > 3 → increase risk
- Low liquidity → increase risk
- High volatility → increase risk

Output:

```
{ expected_return: 0.18%, risk_score: 62/100, confidence: 74% }
```

PHASE 5 — AI EXPLANATION LAYER

You are NOT building financial advice.

You are translating math into plain language.

Input:

`Cycle: BTC → ETH → USDT → BTC Return: 0.18% Risk: Medium Confidence: 74%`

Output example:

> OmniQuant detected a temporary pricing imbalance across simulated exchanges.
> Under current conditions, a $100 theoretical allocation may return approximately $100.18 after fees.
> Risk level: Moderate due to liquidity and volatility factors.

Do NOT output:

- “Guaranteed”
- “Safe”
- “Risk-free”

PHASE 6 — Backend API (FastAPI)

Create endpoints:

1. `/scan`

Triggers detection.

2. `/results`

Returns structured JSON:

`{ cycles: [...], best_opportunity: {...}, timestamp: ... }`

PHASE 7 — Frontend (Keep It Elegant)

Dashboard sections:

1️⃣ Live Arbitrage Opportunities

Table:

- Path
- Expected Return

- Risk
- Confidence

2️⃣ AI Explanation Box

Human-readable summary.

3️⃣ Market Graph Visualization (Optional Flex)

Use:

- Network graph
- Highlight negative cycle

That looks impressive.

PHASE 8 — Data Source

For hackathon:

Option A (Safe):

- Simulated price generator

Option B (Better):

- Pull public API price data from exchanges
- Only read prices
- No trading API keys

Safer for demo.

PHASE 9 — Final Polish Features (High Impact)

Add:

- 🔹 **Confidence Meter (Animated gauge)**
- 🔹 **Opportunity Heat Index**
- 🔹 **Capital Allocation Suggestion (Simulated)**

🔹 Dark Quant-Themed UI

Keep UI minimal and sharp.

PHASE 10 — README Structure

Your README must include:

1. Problem Statement
2. Mathematical Model
3. Architecture Diagram
4. Risk Disclaimer
5. Future Work

That makes it serious.

WHAT MAKES THIS MIT-LEVEL

Not UI.
Not hype.

It's:

- Log-space arbitrage modeling
- Negative cycle detection
- Realistic slippage simulation
- Structured architecture
- Clear financial disclaimers
- Deterministic system design

Realistic Hackathon Timeline

Day 1:

- C++ core
- Graph detection
- Python bridge

Day 2:

- Simulator
- API
- Frontend
- AI explanation layer
- Final polishing

Biggest Mistakes To Avoid

❌ Calling it guaranteed
❌ Ignoring slippage
❌ No risk model
❌ Only 2-exchange comparison
❌ No math slide

Final Result

If implemented correctly, OmniQuant becomes:

> A Graph-Theoretic Arbitrage Detection & Simulation Engine with Explainable Intelligence Layer.

That is technically impressive and product impressive.