72 hr

We are building:

> **OmniQuant v2 — Arbitrage Detection & Market Microstructure Simulation Platform**

🔷 GLOBAL RULES (Before You Start)

1. **Stability > Features**
2. **Core engine must work before UI**
3. **Every module must produce measurable output**
4. **All results must be labeled "Simulated / Theoretical"**
5. **No real trade execution**

🧭 PHASE 1 — FOUNDATION & ARCHITECTURE SETUP

Objective:

Create the structural backbone of the system.

Deliverables:

- Repository structure
- Build system working
- C++ core compiling
- Python bridge connected
- Basic API returning mock data

Step 1 — Create Repository Structure

```
OmniQuant/ | ├── core/ (C++ Engine) ├── bindings/ (pybind11) ├── simulation/ ├── risk/ ├── optimizer/ ├── analytics/ ├── api/ ├── frontend/ ├── tests/ └── README.md
```

Step 2 — Setup C++ Build

Use:

- CMake
- C++17

Ensure:

- Builds clean
- Outputs shared library

Step 3 — Setup Python Bridge

Use:

- pybind11

Expose minimal function:

`detect_cycles(market_data)`

Return dummy cycle first.

Test:

`import omniquant_core print(omniquant_core.detect_cycles(...))`

If this works → foundation is stable.

🧠 PHASE 2 — CORE ARBITRAGE ENGINE (C++)

Objective:

Make arbitrage detection mathematically correct.

Step 1 — Graph Model

- Directed weighted graph
- Weight = −log(rate × (1 − fee))

Store:

- Nodes
- Edges
- Metadata

Step 2 — Negative Cycle Detection

Implement:

- Bellman-Ford
- Cycle reconstruction

Output:

- Path
- Raw theoretical return

Test with synthetic data:

- Force known arbitrage cycle

Step 3 — Edge Pruning

Before running Bellman-Ford:

- Remove low-liquidity edges
- Remove extreme-fee edges

Improves performance and realism.

Step 4 — Performance Metrics

Track:

- Detection runtime (ms)
- Graph size
- Edge count

Expose to Python.

📊 PHASE 3 — MARKET MICROSTRUCTURE SIMULATION (PYTHON)

Objective:

Turn raw arbitrage into realistic execution model.

Step 1 — Order Book Model

For each pair:

- Simulated top 5 bids/asks
- Associated depth

Step 2 — Market Impact Model

Implement nonlinear impact:

impact=k·(volume/liquidity)αimpact = k \cdot (volume/liquidity)^\alphaimpact=k·(volume/liquidity)α

Apply to execution price.

Step 3 — Slippage Model

Compute:

- Effective rate after depth consumption
- Multi-hop compounding impact

Step 4 — Latency Sensitivity

Simulate:

- 0ms
- 50ms
- 100ms delay

Adjust price drift randomly.

Compute:

- Arbitrage half-life

Step 5 — Monte Carlo Engine

For each opportunity:

- Run 500–1000 simulations
- Randomize:
  - Latency
  - Slippage
  - Minor volatility noise

Return:

- Mean return
- Std deviation
- 5% worst case
- Probability negative

This makes system institutional-grade.

🛡 PHASE 4 — RISK ENGINE

Objective:

Quantify uncertainty.

Step 1 — Liquidity Risk Score

Based on:

- Volume consumed / depth

Step 2 — Complexity Risk

Longer cycle → higher risk.

Step 3 — Volatility Exposure

Estimate short-window volatility.

Step 4 — Composite Risk Score

Weighted aggregation:

Risk=w1L+w2C+w3VRisk = w_1L + w_2C + w_3VRisk=w1L+w2C+w3V

Normalize 0–100.

Step 5 — Confidence Metric

Confidence=1−Probability(negative_return)Confidence = 1 - Probability(negative_return)Confidence=1−Probability(negative_return)

📈 PHASE 5 — ANALYTICS & PERSISTENCE

Objective:

Look like a research tool.

Step 1 — Opportunity Tracking

Track:

- First detected timestamp
- Duration alive
- Peak return
- Decay pattern

Step 2 — Persistence Metric

Measure:

- Frequency across time window
- Stability of return

Compute:

- Mean
- Variance
- Sharpe-like ratio

Step 3 — Stress Testing

Simulate:

- +1% uniform price shock
- −1% liquidity shock
- Fee increase

Check survival rate.

💼 PHASE 6 — CAPITAL OPTIMIZER

Objective:

Institutional-level decision logic.

Step 1 — Input:

- Capital budget
- List of opportunities
- Risk scores
- Liquidity constraints

Step 2 — Optimization

Maximize:

∑xiri\sum x_i r_i∑xiri

Subject to:

- Σx_i ≤ capital
- Risk budget limit
- Liquidity cap

Start with greedy allocation.
Advanced: linear programming.

🌐 PHASE 7 — API LAYER (FASTAPI)

Endpoints:

`/scan`

Triggers detection pipeline.

`/opportunities`

Returns ranked list.

`/metrics`

Returns system telemetry.

`/stress`

Returns robustness results.

🖥 PHASE 8 — FRONTEND DASHBOARD

Minimal but elite.

Sections:

1. Ranked Opportunities Table
2. Risk Breakdown Panel
3. Monte Carlo Distribution Chart
4. Latency Decay Curve
5. System Telemetry

Dark theme. Clean typography.

No clutter.

🤖 PHASE 9 — AI EXPLANATION LAYER

Convert structured output into:

- Summary
- Expected return
- Worst-case scenario
- Risk level
- Robustness indicator
- Half-life

Always add disclaimer footer.

📚 PHASE 10 — DOCUMENTATION

README must include:

1. Mathematical modeling explanation
2. Graph-theoretic arbitrage proof
3. Simulation assumptions
4. Risk methodology
5. Known limitations

6. Future real-trade integration plan

This is where judges decide seriousness.

🚨 FINAL SYSTEM CHECKLIST

Before demo:

- Arbitrage detection works on synthetic + real API data
- Monte Carlo produces distribution plot
- Risk score varies meaningfully
- Stress test changes results
- Latency half-life visible
- Capital allocator outputs suggestion
- Dashboard responsive
- Disclaimers visible everywhere

WHAT YOU END UP WITH

Not:

“Crypto money machine”

But:

> A deterministic arbitrage detection and execution-simulation research engine with statistical validation and microstructure modeling.

That is how a strong MIT/IIT systems + quant student would build it.