Implementation v2

🔷 OmniQuant v2 — Quantitative Market Inefficiency Research Platform

This is now MIT-grade.

CORE PHILOSOPHY SHIFT

Before:

> Detect arbitrage cycles.

Now:

> Model, measure, stress-test, and rank structural inefficiencies under uncertainty.

That is a different level.

NEW FEATURES (JANE STREET MODE)

We keep your previous system.
Now we add layers.

1️⃣ Order Book Microstructure Modeling

Your v1 assumes a single price.

Real markets have:

- Bid
- Ask
- Depth
- Spread
- Impact cost

Add:

🔹 Level-2 Order Book Simulation

For each exchange:

- Store top N bids/asks
- Simulate market impact

Profit calculation becomes:

$P = \prod_i (executed\_price_i) - 1$

Where executed_price depends on depth consumed.

2️⃣ Market Impact Function

Implement nonlinear slippage:

$$impact = k \cdot \left(\frac{volume}{liquidity}\right)^\alpha$$

Where:

- k = sensitivity constant
- α > 1 (convex impact)

This shows institutional understanding.

3️⃣ Latency Sensitivity Analysis

Instead of one return value, compute:

- Return at 0ms delay
- Return at 50ms
- Return at 200ms

Plot decay curve:

If opportunity disappears in 30ms → low reliability.

Add:

🔹 Arbitrage Half-Life Metric

Time until expected return = 0.

That is elite.

4️⃣ Statistical Persistence Testing

Don’t just detect once.

Run detection over time window:

- Track frequency of cycle
- Track variance of return
- Compute Sharpe-like metric:

Sharpe=E[R]σRSharpe = \frac{E[R]}{\sigma_R}Sharpe=σRE[R]

Now judges see quant rigor.

5️⃣ Capital Allocation Optimizer

Given multiple cycles:

Maximize:

max∑ixiri\max \sum_i x_i r_imaxi∑xiri

Subject to:

- Liquidity constraints
- Capital limit
- Risk budget

Solve via:

- Linear programming (Python)
- Or simple greedy approximation

This turns OmniQuant into portfolio allocator.

6️⃣ Risk Engine (Real One)

Add risk components:

- **Liquidity Risk**
- **Path Complexity Risk**
- **Volatility Exposure**
- **Execution Uncertainty**
- **Spread Risk**

Aggregate via weighted model:

$$RiskScore = w_1 L + w_2 V + w_3 C + w_4 S$$

Return:

- Risk percentile
- Confidence band

7️⃣ Stress Testing Module

Simulate shocks:

- ±1% price move
- Liquidity drop 30%
- Fee increase

Check:

Does arbitrage survive?

Output:

> Stress Robustness: 42%

This is hedge-fund thinking.

8️⃣ Opportunity Ranking Engine

Rank opportunities by:

Score=ExpectedReturn·ConfidenceRiskScoreScore = \frac{ExpectedReturn \cdot Confidence}{RiskScore}Score=RiskScoreExpectedReturn·Confidence

Display leaderboard.

9️⃣ Regime Detection

Classify market regime:

- High volatility
- Low liquidity
- Stable

Use simple volatility clustering.

Adjust risk model accordingly.

That is advanced thinking.

🔟 Performance Telemetry Dashboard

Add:

- Detection time (ms)
- Graph size
- CPU usage
- Cycle search complexity

Engineers respect instrumentation.

1️⃣1️⃣ Monte Carlo Execution Simulator

For each opportunity:

Simulate 1,000 runs with random:

- Slippage
- Latency
- Volatility noise

Return distribution:

- Mean return
- Worst 5% outcome
- Probability of negative outcome

This makes it academic-level.

1️⃣2️⃣ Edge Attribution Analysis

Break down profit source:

- Fee inefficiency %
- Spread mispricing %
- Multi-hop distortion %

This shows analytical depth.

1️⃣3️⃣ Opportunity Lifecycle Tracking

Track:

- When detected
- Duration alive
- Peak return
- Decay pattern

Display:

Opportunity lifespan chart.

1️⃣4️⃣ Dynamic Graph Pruning

Instead of full graph always:

- Prune low-liquidity edges
- Remove low-confidence edges
- Adaptive edge weighting

Reduces complexity.

Very MIT.

1️⃣5️⃣ Modular Engine Architecture

Refactor:

```
/core graph_engine.cpp cycle_detector.cpp edge_pruner.cpp /simulation
slippage_model.py impact_model.py monte_carlo.py /risk risk_engine.py
stress_test.py /optimizer capital_allocator.py /analytics
persistence_tracker.py regime_detector.py /api main.py /frontend dashboard
```

This looks like research software.

FINAL OMNIQUANT V2 IMPLEMENTATION BLUEPRINT

Follow in this order.

PHASE 1 — Core Engine (C++)

✔ Graph construction
✔ Negative cycle detection
✔ Path extraction
✔ Edge pruning logic
✔ Performance metrics

PHASE 2 — Advanced Simulation Layer (Python)

✔ Level-2 order book model
✔ Nonlinear impact function
✔ Latency sensitivity curve

✔ Monte Carlo simulator
✔ Stress test module

PHASE 3 — Risk & Analytics Layer

✔ Risk scoring engine
✔ Confidence interval computation
✔ Persistence tracker
✔ Sharpe-like metric
✔ Regime classification

PHASE 4 — Optimization Layer

✔ Capital allocator
✔ Opportunity ranking
✔ Risk-budget constraint

PHASE 5 — AI Explanation Layer

Now explanation includes:

- Expected return
- Worst-case simulated return
- Risk percentile
- Robustness score
- Half-life metric

Example:

> OmniQuant detected a multi-hop pricing inefficiency.
> Expected return: 0.21%
> Monte Carlo worst-case: -0.05%
> Latency half-life: 42ms
> Risk percentile: 61%
> Stress robustness: Moderate

That is MIT-grade.

PHASE 6 — Dashboard

Sections:

1. Live Opportunities
2. Ranked Leaderboard
3. Risk Breakdown
4. Monte Carlo Distribution Plot
5. Latency Decay Curve
6. System Telemetry

Clean. Minimal. Quant aesthetic.

PHASE 7 — Mandatory Disclaimers

Add persistent banner:

> OmniQuant is a research-grade arbitrage detection and simulation platform.
> No trades are executed.
> Results are theoretical and subject to model assumptions.
> Not financial advice.

Never remove this.

What This Becomes

Not a hackathon toy.

But:

> A Market Microstructure Research Engine With Explainable Intelligence.

That is something a strong IIT CSE or MIT student would proudly demo.