# **PR: [M4] Oracle TWAP, Multi-Venue Quoting, Gas-Aware Routing, Resiliency & SLOs**

**Branch:** feature/m4-oracles-multivenue-gas  
 **Targets:** UC-4 MVP hardening (post-M3)  
 **Scope (high level):**

* On-chain **Uniswap V3 TWAP** enforcement in policy guards
* **Multi-venue quoting** (Uniswap V3/V2, Curve, 1inch adapter stub)
* **Gas-aware scoring** and **route candidate** selection
* **Resiliency:** retries, timeouts, fallbacks
* **Observability:** metrics + events, SLOs, dashboards
* CLI: plan-best (+ options) and enriched simulate/execute output

## **0) How to pull this PR locally**

git checkout -b feature/m4-oracles-multivenue-gas

# apply changes below

git add -A

git commit -m "M4: Oracle TWAP, multi-venue quoting, gas-aware routing, resiliency & SLOs"

git push origin feature/m4-oracles-multivenue-gas

## **1) Config deltas**

**config/venues.json** (append safely if keys already exist)

{

"uniswapV3": { "quoter": "<addr>", "factory": "<addr>" },

+ "uniswapV2": { "router": "<addr>", "factory": "<addr>" },

+ "curve": { "registries": ["<addr>"], "knownPools": [] },

+ "oneinch": { "quoterUrl": "https://api.1inch.dev/swap/v6.0", "enabled": false }

}

**config/policies.json**

{

"slippageBpsMax": 100,

+ "oracle": { "twapWindowSec": 300, "driftBps": 50 },

+ "quoting": { "timeoutMs": 1500, "retry": { "attempts": 3, "backoffMs": 300 } },

+ "routing": { "gasPriceOverridesGwei": null }

}

**.env.example**

UNIV3\_OBS\_CARDINALITY=64

ORACLE\_DRIFT\_BPS=50

QUOTER\_TIMEOUT\_MS=1500

RETRY\_MAX\_ATTEMPTS=3

RETRY\_BACKOFF\_MS=300

## **2) Contracts (new)**

**contracts/src/oracle/UniV3Twap.sol** (minimal, safe math via TickMath; imports as in your repo)

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

interface IUniswapV3Pool {

function slot0() external view returns (uint160 sqrtPriceX96,int24,int24,uint16,uint16,uint16,uint8,bool);

function observe(uint32[] calldata secondsAgos) external view

returns (int56[] memory tickCumulatives, uint160[] memory secondsPerLiquidityCumulativeX128);

}

library TickMath { function getSqrtRatioAtTick(int24 tick) internal pure returns (uint160) {} }

contract UniV3Twap {

event OracleChecked(address indexed pool, uint32 window, uint160 twap, uint160 spot, uint256 driftBps);

function twapX96(address pool, uint32 window) public view returns (uint160) {

require(window > 0, "TWAP:window=0");

uint32;

secondsAgos[0] = window; secondsAgos[1] = 0;

(int56[] memory tickCumulatives,) = IUniswapV3Pool(pool).observe(secondsAgos);

int56 delta = tickCumulatives[1] - tickCumulatives[0];

int24 twapTick = int24(delta / int56(int32(uint32(window))));

return TickMath.getSqrtRatioAtTick(twapTick);

}

function driftBps(address pool, uint32 window) external view returns (uint256) {

(uint160 spot,, , , , , ,) = IUniswapV3Pool(pool).slot0();

uint160 twap = twapX96(pool, window);

uint256 num = spot > twap ? spot - twap : twap - spot;

uint256 bps = (num \* 10000) / twap;

return bps;

}

}

**contracts/src/oracle/ObservationConfigurator.sol**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

interface IUniswapV3PoolObs { function increaseObservationCardinalityNext(uint16 cardinality) external; }

contract ObservationConfigurator {

address public owner;

modifier onlyOwner(){ require(msg.sender==owner,"!owner"); \_; }

constructor(){ owner = msg.sender; }

function bump(address pool, uint16 newSize) external onlyOwner {

IUniswapV3PoolObs(pool).increaseObservationCardinalityNext(newSize);

}

}

**contracts/src/policy/PolicyGuards.sol** (diff)

+ error OracleDrift(uint256 drift, uint256 max);

+ error OracleUnavailable();

+ uint32 public oracleTwapWindow;

+ uint16 public oracleMaxDriftBps;

+ address public univ3Oracle;

+ event OracleConfigUpdated(uint32 window, uint16 maxBps, address oracle);

function setOracleConfig(uint32 window, uint16 maxBps, address oracle) external onlyOwner {

oracleTwapWindow = window; oracleMaxDriftBps = maxBps; univ3Oracle = oracle;

emit OracleConfigUpdated(window, maxBps, oracle);

}

function \_validateUniv3Step(address pool) internal view {

if (univ3Oracle == address(0) || oracleTwapWindow == 0) return; // backward compat

(bool ok, bytes memory data) = univ3Oracle.staticcall(

abi.encodeWithSignature("driftBps(address,uint32)", pool, oracleTwapWindow)

);

if (!ok || data.length == 0) revert OracleUnavailable();

uint256 drift = abi.decode(data, (uint256));

if (drift > oracleMaxDriftBps) revert OracleDrift(drift, oracleMaxDriftBps);

}

## **3) Core Exec (TS): Oracle client & multi-venue quoting**

**New:** packages/core-exec/src/oracle/univ3.ts

import { PublicClient } from "viem";

import { readContract } from "../viem/read";

export async function spotSqrtX96(client: PublicClient, pool: `0x${string}`): Promise<bigint> {

const [sqrtPriceX96] = await client.readContract({ /\* slot0() ABI \*/ } as any);

return sqrtPriceX96 as bigint;

}

export async function twapSqrtX96(client: PublicClient, oracle: `0x${string}`, pool: `0x${string}`, window: number) {

return await readContract(client, oracle, "twapX96", [pool, BigInt(window)]) as bigint;

}

export function driftBps(spot: bigint, twap: bigint): number {

const num = spot > twap ? spot - twap : twap - spot;

return Number((num \* 10\_000n) / twap);

}

**New:** packages/core-exec/src/quote/types.ts

export type QuoteContext = { publicClient: any; chainId: number; timeoutMs: number; };

export class QuoteError extends Error { constructor(public code: 'TIMEOUT'|'REVERT'|'UNAVAILABLE', msg: string){ super(msg);} }

export type QuoteFn = (ctx: QuoteContext, step: any, amountIn: bigint) => Promise<bigint>;

**New:** per-venue quoters (minimal stubs; fill with your existing ABIs)

* quote/univ3.ts – QuoterV2
* quote/univ2.ts – Router getAmountsOut
* quote/curve.ts – Pool get\_dy
* quote/oneinch.ts – HTTP quoter (behind interface), default disabled

**New:** packages/core-exec/src/quote/multiVenue.ts

import { QuoteFn, QuoteError } from "./types";

import { quoteUniv3 } from "./univ3";

import { quoteUniv2 } from "./univ2";

import { quoteCurve } from "./curve";

export const dispatch: Record<string, QuoteFn> = {

univ3: quoteUniv3, univ2: quoteUniv2, curve: quoteCurve

};

export async function quote(ctx, step, amountIn: bigint) {

const fn = dispatch[step.venue];

if (!fn) throw new QuoteError('UNAVAILABLE', `Unknown venue ${step.venue}`);

return fn(ctx, step, amountIn);

}

**New:** packages/core-exec/src/util/retry.ts

export async function withRetry<T>(fn: () => Promise<T>, attempts=3, backoffMs=300): Promise<T> {

let last; for (let i=0;i<attempts;i++){ try { return await fn(); } catch(e){ last=e; } await new Promise(r=>setTimeout(r, backoffMs\*(i+1))); }

throw last;

}

## **4) Gas-aware scoring & candidate selection**

**New:** packages/core-exec/src/gas/gasPrice.ts

export async function getGasPriceWei(client, overrideGwei?: number): Promise<bigint> {

if (overrideGwei!=null) return BigInt(overrideGwei) \* 10n\*\*9n;

return await client.getGasPrice();

}

**New:** packages/core-exec/src/gas/estimate.ts

export async function estimateStepGas(client, step): Promise<bigint> {

// quick heuristic or per-venue constants if estimateCall unavailable

return BigInt(step.gasHint ?? 150000);

}

**New:** packages/core-exec/src/routing/score.ts

export function netOut(expectedOut: bigint, gasUsed: bigint, gasPriceWei: bigint, outPerWei: bigint = 0n): bigint {

// if outPerWei==0 (unknown), treat gasCost as zero-impact in "out token" terms

const gasCostOut = outPerWei === 0n ? 0n : gasUsed \* gasPriceWei \* outPerWei;

return expectedOut - gasCostOut;

}

**New:** packages/core-exec/src/routing/explain.ts

export type Explain = { chosen: any; discarded: Array<{candidate:any; reason:string}> };

**packages/core-exec/src/RouteBuilder.ts** (diff: candidate generation → scoring → choose)

+ import { quote } from "./quote/multiVenue";

+ import { withRetry } from "./util/retry";

+ import { getGasPriceWei } from "./gas/gasPrice";

+ import { estimateStepGas } from "./gas/estimate";

+ import { netOut } from "./routing/score";

+ import { loadPolicy } from "./policy"; // your existing loader

export async function buildBestRoute(ctx, logicalRoute, amountIn) {

const policy = loadPolicy();

const gasPrice = await getGasPriceWei(ctx.publicClient, policy.routing.gasPriceOverridesGwei ?? undefined);

const candidates = expandVenueCandidates(logicalRoute, ctx.venues); // produce per-step venue variants

const results = [];

for (const c of candidates) {

try {

const out = await withRetry(() => simulateQuotePath(ctx, c, amountIn, policy), policy.quoting.retry.attempts, policy.quoting.retry.backoffMs);

const gas = await estimateRouteGas(ctx, c);

const score = netOut(out.expectedOut, gas, gasPrice, out.outPerWei ?? 0n);

results.push({ c, score, out, gas });

} catch (e) {

results.push({ c, error: String(e) });

}

}

const ok = results.filter(r => !r.error && r.out?.twapOk);

if (!ok.length) throw new Error("NO\_QUOTE: all candidates failed TWAP or quoting");

ok.sort((a,b)=> (b.score > a.score ? 1 : -1));

return { chosen: ok[0], discarded: results.filter(r=>r!==ok[0]).map(r=>({candidate:r.c, reason: r.error ? r.error : "worse-net"})) };

}

async function simulateQuotePath(ctx, candidate, amountIn, policy){

// for each step: quote -> twap check (policy) -> propagate amount

// returns { expectedOut, twapOk, outPerWei? }

}

## **5) CLI**

**packages/cli/src/commands/route.ts** (diff)

program

.command('simulate')

.option('--max-slippage-bps <n>')

+ .option('--venue-candidates <csv>', 'e.g. univ3,univ2,curve')

+ .option('--gas-price-gwei <n>')

.action(simulateRoute);

+program

+ .command('plan-best')

+ .requiredOption('--route-file <path>')

+ .option('--venue-candidates <csv>')

+ .option('--gas-price-gwei <n>')

+ .action(planBest);

+program

+ .command('execute-best')

+ .requiredOption('--route-file <path>')

+ .option('--venue-candidates <csv>')

+ .option('--gas-price-gwei <n>')

+ .action(executeBest);

**Output JSON additions (simulate/plan/execute):**

{

"netOut": "123456789",

"gasUsedEst": "210000",

"usedFallbacks": true,

"discardedCandidates": [{ "venuePlan":"...", "reason":"TIMEOUT|worse-net|TWAP" }]

}

## **6) Telemetry & SLOs**

**New:** packages/core-exec/src/telemetry/metrics.ts

export const metrics = {

quoteLatencyMs: /\* histogram \*/ null,

quotesFailed: /\* counter with {venue,reason} \*/ null,

simSuccess: /\* counter \*/, simFailure: /\* counter{reason} \*/,

oracleDriftBps: /\* histogram \*/

};

// wire to your metrics impl or simple ndjson sink

**New doc:** docs/ops/slo.md (p95 quote latency ≤ 1.5s; sim success ≥ 99%; etc.)

**Dashboard JSON:** docs/ops/dashboards/m4.json (Grafana export stub).

## **7) Tests**

**TS unit/integration**

* packages/core-exec/test/oracle.univ3.spec.ts
* packages/core-exec/test/quote.multivenue.spec.ts
* packages/core-exec/test/routing.score.spec.ts
* packages/cli/test/plan-best.spec.ts
* packages/core-exec/test/retry.helper.spec.ts

**Foundry (fork preferred)**

* contracts/test/oracle/Twap.observe.t.sol
* contracts/test/policy/Guards.oracle.t.sol
* contracts/test/execution/Executor.events.t.sol
* contracts/test/oracle/ObservationConfigurator.t.sol

**Example TS test (drift math):**

import { driftBps } from "../../src/oracle/univ3";

test("driftBps", ()=> {

expect(driftBps(1000n, 1000n)).toBe(0);

expect(driftBps(1100n, 1000n)).toBe(100); // 10%

});

## **8) CI**

**.github/workflows/ci.yml** (add jobs)

* ts-oracle (unit)
* ts-routing (multi-venue)
* foundry-oracle-fork (artifacts, non-required if flaky)
* Cache viem/ABI builds; job timeouts aligned to SLOs

## **9) Rollout (Sepolia → Prod)**

1. Deploy UniV3Twap + ObservationConfigurator.
2. Timelock: setOracleConfig(window=300, maxBps=50, oracle=<UniV3Twap>).
3. Bump pool observation cardinality to UNIV3\_OBS\_CARDINALITY for hot pools.
4. Stage end-to-end:  
   * route plan-best on 3 test routes (univ3-only, univ2-only, mixed),
   * Verify chosen candidate is the expected netOut winner,
   * Confirm TWAP reverts when pools intentionally drifted (manipulated on fork).
5. Enable selected fallback venues in venues.json gradually.
6. Monitor dashboards for p95 quote latency and sim success; adjust retry/backoff if needed.

## **10) Backward compatibility**

* If univ3Oracle==0 or oracleTwapWindow==0: guard **skips** TWAP (M3 behavior).
* venues.json may omit new venues; CLI flags default to safe values.

## **11) Risk notes & mitigations**

* **Oracle unavailability** → OracleUnavailable() revert; runbook: temporarily disable oracle config via timelock while investigating.
* **RPC brownouts** → retries/backoff/fallback venues; runbook documented.
* **Gas scoring uncertainty** (no outPerWei) → treat gas cost as neutral; call out in CLI explain output.

## **12) Acceptance (DoD)**

* TWAP rejections correctly block risky pools (unit + fork tests).
* Multi-venue candidate exploration and **gas-aware** scoring picks a better netOut path in at least one demonstrable case.
* Timeouts trigger fallbacks; pipeline still returns an executable candidate when at least one venue quotes.
* Metrics populate; dashboards meet SLOs during a brief staged load.
* Governance can adjust oracle window/drift and observation cardinality via timelock.

## **13) Commit plan (squashable)**

1. feat(contracts): add UniV3Twap & ObservationConfigurator
2. feat(guards): enforce oracle drift with config & events
3. feat(exec): oracle client, retry helper, multi-venue quoting
4. feat(exec): gas price/estimate + netOut scoring + explain
5. feat(cli): plan-best & execute-best, flags, enriched output
6. chore(config): venues/policies/env defaults for M4
7. feat(telemetry): metrics & SLO docs + dashboard stub
8. test(ts): oracle, multi-venue, scoring, retry
9. test(foundry): twap, guards, events, obs configurator
10. ci: add oracle/routing/fork jobs, cache
11. docs: runbooks & README updates
12. release: bump to 0.4.0-m4 + notes