# **M4 — Oracle-enforced TWAP, multi-venue quoting, gas-aware routing, resiliency & SLOs (≈ 180 steps)**

**Goal:** turn the M3 MVP into a production-capable UC-4 system: **real TWAP enforcement**, **multi-venue** price discovery, **gas-aware** path selection, **retries/failover**, stronger **observability** with SLOs, and **release hardening**.  
 **Assumptions:** M3 is merged (native sentinel supported; sim + timelock + circuit breakers online).

## **A. Branch, baselines & guardrails**

1. Create branch feature/m4-oracles-multivenue-gas.
2. Tag repo pre-m4-cut at the last green on main.
3. Add .env.example entries: UNIV3\_OBS\_CARDINALITY=64, ORACLE\_DRIFT\_BPS=50, QUOTER\_TIMEOUT\_MS=1500, RETRY\_BACKOFF\_MS=300, RETRY\_MAX\_ATTEMPTS=3.
4. Add config/venues.json placeholders for **additional venues** you plan to support in M4 (e.g., univ2, curve, oneinch-quoter) with network addresses.
5. Add config/policies.json keys: "oracle": {"twapWindowSec": 300, "driftBps": 50}, "quoting": {"timeoutMs": 1500, "retry": {"attempts": 3, "backoffMs": 300}}, "routing": {"gasPriceOverridesGwei": null}.
6. Create docs/dev/m4-overview.md with scope, risks, and links to M3 docs.

## **B. Solidity — Uniswap V3 on-chain TWAP enforcement**

1. Add file contracts/src/oracle/UniV3Twap.sol.
2. Implement function twapX96(address pool, uint32 window) external view returns (uint160 avgSqrtPriceX96) using UniswapV3 observe().
3. Implement function driftBps(address pool, uint32 window) external view returns (uint256 bps) comparing spot vs TWAP.
4. Emit event OracleChecked(address pool, uint32 window, uint160 twap, uint160 spot, uint256 driftBps) (for off-chain telemetry).
5. In PolicyGuards.sol, add storage: uint32 oracleTwapWindow, uint16 oracleMaxDriftBps.
6. Add setters setOracleTwapWindow(uint32) and setOracleMaxDriftBps(uint16) with onlyOwner + events.
7. In PolicyGuards.validateStep(...) (or equivalent policy hook), if venue is UniswapV3:  
   * Resolve pool from fee + token pair,
   * Call UniV3Twap.twapX96(pool, oracleTwapWindow),
   * Compute drift vs current slot0().sqrtPriceX96,
   * Revert with error OracleDrift(uint256 drift, uint256 max) if drift > oracleMaxDriftBps.
8. Add error OracleUnavailable(); and revert if observe() fails.
9. Unit tests:  
   * Build fork test where observations cardinality is increased,
   * Validate acceptable drift passes; exaggerated drift reverts.

## **C. Solidity — Pool observation cardinality management**

1. Add contracts/src/oracle/ObservationConfigurator.sol with bumpObservationCardinality(address pool, uint16 newSize) (owner-only).
2. Wire governance (timelock) to call this on hot pools during rollout.
3. Test increasing cardinality on fork; assert slot0().observationCardinality grows.

## **D. TypeScript — Oracle client & policy glue**

1. Create packages/core-exec/src/oracle/univ3.ts:  
   * getSpotSqrtX96(pool), getTwapSqrtX96(pool, window), calcDriftBps(spot, twap).
2. Create packages/core-exec/src/oracle/index.ts interface:  
   * getDriftBpsForStep(step, window): Promise<number>.
3. Update simulator to **use real TWAP**:  
   * Replace M3 stub in twap.ts to call the oracle client,
   * Return twapOk = driftBps <= max.
4. Add tolerant error handling: if oracle call times out → mark twapOk=false with reason.

## **E. Multi-venue quoting abstraction (TS)**

1. Add packages/core-exec/src/quote/types.ts with:  
   * QuoteContext { publicClient, chainId, timeoutMs },
   * QuoteFn = (ctx, step, amountIn) => Promise<bigint>.
2. Implement per-venue modules:  
   * quote/univ3.ts (existing),
   * quote/univ2.ts (router getAmountsOut),
   * quote/curve.ts (pool get\_dy or get\_dy\_underlying),
   * quote/oneinch.ts (HTTP quoter; stubbed with adapter or mockable interface).
3. Create packages/core-exec/src/quote/multiVenue.ts:  
   * Given a **step** with a venue tag and payload, dispatch to correct QuoteFn.
4. Standardize errors via QuoteError with code: 'TIMEOUT'|'REVERT'|'UNAVAILABLE'.

## **F. Gas-aware routing & total cost scoring**

1. Add packages/core-exec/src/gas/gasPrice.ts: best-effort gas price fetcher (RPC eth\_gasPrice, with env override).
2. Add packages/core-exec/src/gas/estimate.ts:  
   * Quick estimator per step type (based on ABI selector), allow per-venue constants fallback.
3. Create packages/core-exec/src/routing/score.ts:  
   * scoreRoute({expectedOut, gasUsed, gasPrice}) => netOut = expectedOut - (gasUsed \* gasPrice \* tokenOutPerWei),
   * Include tokenOut/eth conversion via last-hop price or a USD oracle **optional** (if not available, assume 1:1 for WETH or skip scoring).
4. Update RouteBuilder pipeline:  
   * For a given logical route template, generate **venue variants** (e.g., UNIV3 vs UNIV2 at hop),
   * Simulate each variant (quotes) + gas estimate,
   * Choose max netOut variant that passes oracle guard and policy limits.
5. Add packages/core-exec/src/routing/explain.ts to produce a human/JSON explanation: chosen path, discarded candidates with reasons (timeout, worse net, oracle drift).

## **G. Resiliency: retries, fallback venues, partial degrade**

1. Implement withRetry helper in packages/core-exec/src/util/retry.ts with jitter and max attempts from policy.
2. In multi-venue quoting, wrap each venue call with withRetry.
3. If a step’s primary venue fails (timeout/unavailable), try **fallback** venue list (from venues.json order).
4. If all quotes for any step fail: return simulation.failure = 'NO\_QUOTE' early.
5. Annotate simulation result with usedFallbacks: boolean and failures[].
6. In CLI, present a clear warning when fallbacks used.

## **H. Stronger slippage: per-step minOut & bundle minOut**

1. Update derivePerStepMinOut to compute **both** per-step and bundle-level minOut according to maxSlippageBps.
2. Ensure executor enforces per-step minOut via payload or sidecar constraints (depending on your existing model).
3. Add tests where one step meets minOut but bundle exceeds limit → revert as expected.

## **I. CLI extensions**

1. Extend route simulate to accept:  
   * --venue-candidates univ3,univ2,curve,
   * --fallbacks-per-step 2,
   * --gas-price-gwei 12 (override).
2. Output new JSON fields: netOut, gasUsedEst, usedFallbacks, discardedCandidates[].
3. Add route plan-best --route-file:  
   * Generates and **materializes** the chosen variant as an executable route JSON.
4. Add route execute-best --route-file shortcut that runs plan-best then executes.

## **J. Contracts — Executor execution safety nets**

1. Add **replay-nonce** check per bundle if not present: mapping(bytes32 routeId => uint256 nonce); bump on success.
2. Add event BundleFailed(bytes32 routeId, uint8 stepIndex, bytes reason).
3. Surface revert reasons for off-chain logging (limited length).

## **K. Observability & SLOs**

1. Define **SLOs** in docs/ops/slo.md:  
   * Quote latency p95 ≤ 1.5s,
   * Simulation success ≥ 99% on healthy RPC,
   * Execution revert rate ≤ 1% (policy reverts excluded),
   * Oracle check timeouts ≤ 0.5%.
2. Extend telemetry (packages/core-exec/src/telemetry/metrics.ts) with counters/histograms:  
   * quote\_latency\_ms{venue}, quotes\_failed\_total{venue,reason},
   * sim\_success\_total, sim\_failure\_total{reason},
   * bundle\_reverts\_total{reason}, oracle\_drift\_bps\_hist.
3. Add labels for used\_fallback=true/false.
4. Export /metrics (dev) and a file sink events.ndjson (prod-like artifact).
5. Add a Grafana dashboard JSON (dev) under docs/ops/dashboards/m4.json.

## **L. Test data & fixtures**

1. Update config/fixtures/tokens.sepolia.json with pools usable for TWAP on UniswapV3 (liquidity > 0).
2. Add per-venue mock ABIs for local tests (Curve/UniV2 minimal).
3. Add one or two volatile pools to test high drift.

## **M. TypeScript unit & integration tests**

1. packages/core-exec/test/oracle.univ3.spec.ts: spot vs twap vs drift math.
2. packages/core-exec/test/quote.multivenue.spec.ts: happy path + timeouts + fallbacks.
3. packages/core-exec/test/routing.score.spec.ts: netOut scoring with gas conversion.
4. packages/cli/test/plan-best.spec.ts: candidate generation → selection correctness.
5. packages/core-exec/test/retry.helper.spec.ts: jitter/backoff and attempt count.

## **N. Foundry tests (fork preferred)**

1. contracts/test/oracle/Twap.observe.t.sol: TWAP correctness on fork.
2. contracts/test/policy/Guards.oracle.t.sol: revert on excessive drift.
3. contracts/test/execution/Executor.events.t.sol: emits BundleFailed with reason on bad step.
4. contracts/test/oracle/ObservationConfigurator.t.sol: cardinality bump works and is timelock-guarded.

## **O. Quoter timeouts & RPC hygiene**

1. Implement per-venue **timeout** wrapper using AbortController (or viem client timeout).
2. Add configurable RPC method bans or fallbacks (if primary RPC rate-limits).
3. Document recommended RPC pool size for staging vs CI.

## **P. Security hardening (M4 delta)**

1. Add **invariants** doc docs/security/m4-invariants.md:  
   * Oracle checks precede swaps for each step,
   * Native unwrap only at **final** hop,
   * MinOut never increases mid-bundle.
2. Run static analysis again; diff against M3 report.
3. Ensure **no** external calls before validation (checks-effects-interactions).
4. Validate all governance setters are timelocked and evented.

## **Q. Performance & gas baselines**

1. Create forge snapshot cases including:  
   * Multi-venue path using UniswapV3,
   * Same path using UniV2 (to compare gas),
   * With/without native unwrap.
2. Add table in docs/dev/gas.md showing delta vs M3; call out tradeoffs.

## **R. Rollout plan (staging → prod)**

1. Add script/DeployOracleContracts.s.sol to deploy UniV3Twap & ObservationConfigurator.
2. Update script/Configure.s.sol to:  
   * Set oracleTwapWindow, oracleMaxDriftBps,
   * Register oracle address in guards (if you keep a registry).
3. Stage on Sepolia:  
   * Bump observation cardinality for the core pools,
   * Confirm TWAP returns finite values.
4. Run route plan-best on a few candidate routes; confirm multi-venue selection behaves as expected.
5. Execute a handful of bundles; capture telemetry artifacts.

## **S. Backward compatibility**

1. Keep M3 behavior if oracle config is **unset**: policy skips TWAP, logs a warning.
2. Allow venues.json to omit venues not yet supported; candidates list must intersect availability.
3. CLI retains M3 flags; new flags default to safe values.

## **T. Failure drills & runbooks**

1. docs/runbooks/oracle-failures.md:  
   * How to detect oracle timeouts,
   * How to temporarily disable oracle checks (timelock action) while keeping other guards.
2. docs/runbooks/quoting-degrade.md:  
   * How to re-order fallback venues or disable one.
3. docs/runbooks/retries-tuning.md:  
   * Guidance on backoff tuning under RPC brownouts.

## **U. CI pipeline upgrades**

1. Update .github/workflows/ci.yml:  
   * New job ts-oracle (unit tests),
   * New job ts-routing (multi-venue scoring),
   * Fork job foundry-oracle-fork gated as non-required if flaky (publish artifacts).
2. Add job-level timeouts aligned to SLOs (e.g., 4m per test matrix shard).
3. Cache ABI & viem build outputs to reduce CI time.

## **V. Docs & examples**

1. Extend README.md:  
   * “Choosing the best path with route plan-best”
   * Example output JSON annotated.
2. Add examples/routes/eth-usdc-multivenue.json with candidates.
3. Add docs/content/uc4-multiswap-permissions.mdx section:  
   * Oracle drift guard rationale & diagrams (spot vs twap).

## **W. Acceptance criteria (stakeholder sign-off)**

1. Simulation rejects high-drift pools per policy and surfaces reason in CLI.
2. Multi-venue candidate generation selects a better **netOut** path at least in one demo case.
3. Timeouts trigger fallbacks; execution continues if at least one candidate quotes.
4. Per-step minOut + bundle minOut enforced in executor; failing paths revert with meaningful event.
5. Telemetry records quote latency histograms, and dashboard shows p95 within SLO.
6. Governance can bump observation cardinality and change drift window/limit via timelock.

## **X. Release prep**

1. Bump versions to 0.4.0-m4 across affected packages.
2. Write m4-release-notes.md with upgrade/migration steps and known limitations.
3. Tag m4-oracles-multivenue-gas.
4. Merge after full CI green + staging acceptance artifacts attached.

### **Key code skeletons (drop-in starters)**

**UniV3 TWAP (Solidity)**

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

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

int56[] memory tickCumulatives;

uint160[] memory secondsPerLiquidityCumulativeX128;

uint32;

secondsAgos[0] = window;

secondsAgos[1] = 0;

(tickCumulatives, secondsPerLiquidityCumulativeX128) = IUniswapV3Pool(pool).observe(secondsAgos);

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

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

return TickMath.getSqrtRatioAtTick(twapTick);

}

**TS: drift bps**

export function driftBps(spotX96: bigint, twapX96: bigint): number {

const num = spotX96 > twapX96 ? spotX96 - twapX96 : twapX96 - spotX96;

const bps = Number((num \* 10\_000n) / twapX96);

return bps;

}

**TS: multi-venue dispatcher**

export async function quote(ctx: QuoteContext, step: Step, amountIn: bigint): Promise<bigint> {

switch (step.venue) {

case "univ3": return quoteUniv3(ctx, step, amountIn);

case "univ2": return quoteUniv2(ctx, step, amountIn);

case "curve": return quoteCurve(ctx, step, amountIn);

default: throw new QuoteError("UNAVAILABLE", `Unknown venue ${step.venue}`);

}

}

**TS: retry**

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

let lastErr: unknown;

for (let i = 0; i < attempts; i++) {

try { return await fn(); } catch (e) { lastErr = e; }

await new Promise(r => setTimeout(r, backoffMs \* (i + 1)));

}

throw lastErr;

}

### **“Definition of Done” for M4**

* **Policy-enforced TWAP** live on staging; excessive drift reverts earlier than swap execution.
* **Multi-venue** quoting and **gas-aware** scoring choose the best executable variant; fallbacks in place.
* **Resiliency**: timeouts and retries tuned; degraded but successful runs observed in staging.
* **SLOs**: dashboards show p95 quote latency ≤ target and simulator success ≥ 99% during a staged load test.
* **Governance**: timelock successfully adjusts oracle windows/limits and pool observation cardinality.
* **Docs & runbooks**: operators can diagnose oracle failures and reconfigure fallbacks without code changes.
* **CI**: new oracle/multi-venue tests pass; fork tests publish artifacts; release notes prepared.