Contents

Meridian: Quantum Futarchy

A Formal Specification for Capital-Efficient Prediction Market Governance on Monad

Version: 2.0.0-draft **Date**: February 2026 **Status**: Pre-implementation specification

---

Table of Contents

---

1. Problem Statement

1.1 Classical Futarchy (MetaDAO Model)

MetaDAO's futarchy creates two conditional markets per proposal – a pass market and a fail market – each requiring dedicated liquidity. This architecture produces four structural problems:

P1: Capital fragmentation. A trader with D tokens facing N proposals can allocate at most D/N per market. Thin markets produce noisy prices. MetaDAO's Proposal 6 demonstrated this: a single actor spent $250,000 to manipulate a thin pass market. The defense relied on vigilant counter-traders rather than structural guarantees.

P2: Linear proposal cost. Each proposal requires fresh conditional token minting and AMM liquidity. MetaDAO mitigates this by borrowing ~50% of its spot pool for each proposal, but this creates concentration risk when multiple proposals are active simultaneously.

P3: Binary evaluation. Each proposal is evaluated independently against the status quo (pass vs. fail). There is no mechanism to compare competing proposals directly. If five ideas address the same problem, they must be evaluated in five separate proposal cycles.

P4: Fail market nihilism. MetaDAO's fail market is structurally uninteresting to trade – conditional tokens simply redeem 1:1 for the underlying if the proposal fails. This produces noisy, low-volume fail markets that yield unreliable price signals.

1.2 What Quantum Markets Solve

Paradigm's quantum markets (Yukseloglu & Larbi, 2025) introduce a capital efficiency primitive: deposit once, receive full trading power across all proposals within a decision. Only the winning proposal settles; all other markets revert.

However, Paradigm's paper is a conceptual architecture piece, not a governance specification. Their reference implementation uses a 30-second TWAP with an all-time-high tracking mechanism that is inadequate for adversarial governance. Specific gaps:

- No formal solvency proof (the reference implementation has a potential gap in seed liquidity accounting where YES_supply = 2 ∗ vUSD_locked during initialization)
- The marketMax only tracks the all-time highest YES price, not a time-weighted closing price
- No minimum liquidity or trading volume requirements per proposal
- No cross-proposal arbitrage constraints

1.3 What This Specification Does

This document specifies **Quantum Futarchy**: the synthesis of quantum market capital efficiency with Hanson's futarchy governance mechanism, designed for deployment on Monad. Concretely:

- Multiple proposals compete within a single Decision (quantum superposition)
- Traders deposit once and receive full trading credits across all proposals
- Each proposal's market produces a welfare prediction via CPMM price discovery
- The proposal with the highest time-weighted welfare prediction wins (wave function collapse)
- Losing markets fully revert; the winning market settles normally
- All math is specified with formal solvency proofs
- The design accounts for Monad's 400ms blocks, gas-limit charging, and parallel execution model

---

2. Mechanism Overview

2.1 One-Paragraph Summary

A Decision poses a governance question. Any number of Proposals compete to answer it. Traders deposit the base asset (MON) into the Decision once and receive virtual trading credits replicated across every Proposal market. Each market is a virtual CPMM that prices YES/NO outcome tokens, producing an implied welfare prediction. After a configurable trading period, a TWAP-based oracle reads the final welfare predictions and selects the winning Proposal. The winning market settles like a standard prediction market (correct traders profit, incorrect traders lose). All losing markets revert completely – every trade is unwound and every participant recovers their full deposit. The system is zero-sum among participants in the winning market and fully capital-conserving across all losing markets.

2.2 Quantum Property

The defining property: a deposit of D MON into a Decision with N Proposals gives the trader D MON of trading power in **each** of the N markets simultaneously. This is possible because at most one market settles. The deposit exists in superposition until collapse.

Metric	Classical (MetaDAO)	Quantum (Meridian)
Capital per market	D/N	D
Capital efficiency	1/N (degrades)	1 (constant)
Adding a proposal	Requires liquidity	Zero marginal cost
Comparison model	Each vs. status quo	All vs. each other
Fail market needed	Yes (noisy)	No (eliminated)

2.3 Why This Works

The capital efficiency claim is not "free money." It works because of a fundamental constraint: **only one proposal can win per Decision.** A trader's deposit backs at most one settlement. The N−1 losing markets contribute zero net liability because every trade in them is unwound. The trader's realized exposure is exactly one market's worth of risk, regardless of how many markets they traded in.

2.4 Meridian as a Governance Intelligence Machine

Prediction markets tell you the *probability* of events. Meridian tells you the *impact* of governance actions – before they happen.

Each proposal market produces a real-time, market-priced answer to the question: **"If this proposal is implemented, what does the crowd expect the welfare outcome to be?"** The welfare price is not a vote count; it is a continuously updating signal backed by real capital, produced by traders who profit only when they are right.

This is structurally identical to what Lightcone calls "impact markets" applied to governance. Consider a Decision with three proposals:

Proposal	Welfare Price	Interpretation
Hire Alice as Lead	72%	Crowd expects strong positive impact
Hire Bob as Lead	48%	Crowd expects marginal/neutral impact
Status Quo	35%	Crowd expects decline without action

The *deltas* between these prices are the information product. They are not opinions; they are positions backed by money with settlement consequences. The delta between Alice (72%) and Bob (48%) is a 24-point market-priced signal that Alice is the superior hire – information that no governance vote, analyst, or committee can produce with comparable rigor.

Furthermore, probability is irrelevant to the trader's decision. A trader buying YES on "Hire Alice" is not betting on whether Alice *will* be hired (that is the governance mechanism's job). They are expressing conviction that *if she is hired*, the welfare metric improves. If Alice's proposal loses, the trader's position fully reverts – they pay nothing. The premium exists only in the universe where the governance action actually occurs.

This separation of impact from probability is what makes quantum futarchy a fundamentally new information primitive for governance: market-derived intelligence about what a DAO *should* do, not what it *will* do.

---

3. Formal Definitions

3.1 Decision

A Decision D is a tuple:

```
D = (id, creator, title, welfareMetricName, deadline, baseAsset, status, proposals
     totalDeposits, resolutionMode, welfareOracleAddress, measurementPeriod,
     minImprovement, M_baseline, M_actual, outcome, disputeWindow, disputeBond)
```

Where: - id: Unique identifier (uint256) - creator: Address of the Decision creator - title: Human-readable governance question - welfareMetricName: Description of what the markets predict (e.g., "Token Price USD") - deadline: Block number after which trading closes and collapse can occur - baseAsset: The deposited asset (MON) - status: One of {OPEN, COLLAPSED, MEASURING, RESOLVED, DISPUTED, SETTLED} - proposals[]: Ordered list of Proposals - totalDeposits: Sum of all deposits into this Decision - resolutionMode: One of {MODE_A, MODE_B, MODE_C} (see Section 8) - welfareOracleAddress: Address of IWelfareOracle adapter (Mode B/C only, zero for Mode A) - measurementPeriod: Blocks to wait after collapse before reading outcome (Mode B/C only, 0 for Mode A) - minImprovement: Minimum welfare improvement threshold, in basis points (default 0) - M_baseline: Welfare metric value snapshot at collapse (set by collapse()) - M_actual: Welfare metric value at resolution (set by resolve()) - outcome: Resolution outcome {UNRESOLVED, YES, NO} (set by resolve() or resolveDispute()) - disputeWindow: Blocks after resolution during which disputes can be filed (Mode B/C only) - disputeBond: Minimum bond required to file a dispute (Mode B/C only)

3.2 Proposal

A Proposal P is a tuple:

```
P = (id, decisionId, proposer, title, description, yesReserve, noReserve, k, totalV
```

Where: - id: Unique identifier within the Decision - decisionId: Parent Decision - proposer: Address that submitted the Proposal - title, description: Human-readable metadata - yesReserve: Current YES token reserve in the virtual CPMM - noReserve: Current NO token reserve in the virtual CPMM - k: CPMM invariant (yesReserve * noReserve) - totalVirtualMinted: Total vMON minted for this Proposal across all users

3.3 Position

A Position tracks a user's state within a specific Proposal market:

```
Position(user, decisionId, proposalId) = (yesBalance, noBalance, vMonSpent)
```

Where: - yesBalance: YES tokens held by the user for this Proposal - noBalance: NO tokens held by the user for this Proposal - vMonSpent: Total virtual MON spent in this Proposal's market (cost basis)

3.4 Deposit

A Deposit tracks a user's global state within a Decision:

```
Deposit(user, decisionId) = (amount, claimed[proposalId])
```

Where: - amount: Total MON deposited - claimed[proposalId]: How much has been claimed as vMON for each Proposal

---

4. Token Architecture

4.1 Token Hierarchy

```
MON (real, deposited once)
   |
   +-- Decision deposit balance (global per decision per user)
         |
         +-- Proposal 0: vMON_0 --> YES_0 + NO_0
         +-- Proposal 1: vMON_1 --> YES_1 + NO_1
         +-- Proposal 2: vMON_2 --> YES_2 + NO_2
         +-- ...
         +-- Proposal N: vMON_N --> YES_N + NO_N
```

4.2 Virtual MON (vMON)

Each Proposal has its own independent virtual MON denomination. vMON is **not** an ERC-20 token – it is an internal accounting unit tracked in contract storage. This avoids gas costs of deploying N token contracts per Decision.

Minting rule: For a user with deposit D in a Decision, the claimable vMON for Proposal i is:

```
claimable_i = D - claimed[i]
```

If the user deposits more MON, D increases, and additional vMON becomes claimable for every Proposal – including already-existing ones. This allows late deposits.

Key property: vMON is per-Proposal. vMON_0 and vMON_1 are entirely separate accounting domains. A user claiming vMON in Proposal 0 does NOT reduce their claimable vMON in Proposal 1.

4.3 YES/NO Outcome Tokens

Within each Proposal market, YES and NO tokens represent conditional claims:

```
1 vMON —> mint —> 1 YES + 1 NO (always)
1 YES + 1 NO —> redeem —> 1 vMON (always)
```

This is the standard conditional token split. Like vMON, these are internal accounting entries, not separate ERC-20 contracts.

Invariant 1 (Token Conservation): For any Proposal i at any point in time:

```
YES_supply_i = NO_supply_i
```

Proof: mintYesNo creates them in equal pairs. redeemYesNo destroys them in equal pairs. CPMM swaps redistribute tokens between users and the pool but never create or destroy tokens.

Invariant 2 (vMON Conservation): For any Proposal i:

```
vMON_circulating_i + vMON_locked_i = vMON_minted_i
```

Where vMON_locked_i = YES_supply_i (each YES/NO pair was created by locking 1 vMON).

---

5. Virtual CPMM Mechanics

5.1 Pool Model

Each Proposal market is a constant-product market maker operating on the YES/NO token pair:

```
yesReserve * noReserve = k
```

The CPMM does not hold real assets. It operates on virtual balances tracked in contract storage. "Depositing into the CPMM" means transferring internal accounting entries.

5.2 Implied Welfare

The predicted welfare for Proposal i is derived from the YES token price:

```
welfare_i = noReserve_i / (yesReserve_i + noReserve_i)
```

This produces a value in [0, 1] (stored as basis points [0, 10000] for integer arithmetic). A higher welfare means the market believes this Proposal leads to better outcomes.

Intuition: YES tokens represent "this proposal is good." If YES is expensive (high demand), yesReserve is low relative to noReserve, pushing welfare up.

5.3 Trading: Buy YES

A user wants to express belief that Proposal i is good. They buy YES tokens by spending vMON.

Step 1: Claim vMON. User claims d vMON for Proposal i (debited from their Decision deposit's unclaimed balance for this Proposal).

Step 2: Mint YES+NO pair. The d vMON mints d YES and d NO tokens.

Step 3: Sell NO to pool. The d NO tokens are sold to the CPMM:

```
YES_received = yesReserve * d / (noReserve + d)
```

Step 4: Update reserves.

```
yesReserve_new = yesReserve − YES_received = yesReserve * noReserve / (noReserve +
noReserve_new = noReserve + d
```

Step 5: User receives. The user keeps the d YES from minting PLUS the YES_received from the swap.

```
total_YES = d + YES_received = d + yesReserve * d / (noReserve + d)
          = d * (yesReserve + noReserve + d) / (noReserve + d)
```

Combined formula (Buy YES with d vMON):

```
YES_out = d * (yesReserve + noReserve + d) / (noReserve + d)
```

Execution price:

```
price_per_YES = d / YES_out = (noReserve + d) / (yesReserve + noReserve + d)
```

5.4 Trading: Buy NO

Symmetric to Buy YES:

```
NO_out = d * (yesReserve + noReserve + d) / (yesReserve + d)
```

5.5 Trading: Sell YES

A user holding q YES tokens wants to sell them back. This is the reverse of buying: the user sells YES to the pool for NO, then redeems YES+NO pairs for vMON.

The amount of vMON recovered requires solving a quadratic. Let a be the NO tokens received from selling YES to the pool. Then a pairs can be redeemed.

From the CPMM: selling q_sell YES tokens:

```
NO_out = noReserve * q_sell / (yesReserve + q_sell)
```

But the user wants to maximize redemption. They sell some YES to get NO, then redeem matching pairs. Let a be the number of pairs redeemed. The user must sell (q − a) YES to the pool to receive a NO tokens.

```
a = noReserve * (q − a) / (yesReserve + (q − a))
a * (yesReserve + q − a) = noReserve * (q − a)
a * yesReserve + a*q − a^2 = noReserve*q − noReserve*a
a^2 − a*(yesReserve + q + noReserve) + noReserve*q = 0
```

Solving via quadratic formula (taking the smaller root):

```
a = [(yesReserve + noReserve + q) − sqrt((yesReserve + noReserve + q)^2 − 4 * noRe
```

vMON returned = a (the number of pairs redeemed).

5.6 Fees

A swap fee phi (e.g., 0.003 = 0.3%) is applied to the NO tokens entering the pool during a Buy YES trade:

```
effective_d = d * (1 − phi)
YES_out = effective_d * (yesReserve + noReserve + effective_d) / (noReserve + effec
fee_retained = d * phi  (stays in the pool, increasing k)
```

Fees accrue to the pool, benefiting the protocol (since there are no external LPs in virtual pools). Fee revenue is retained in the pool's reserves and effectively redistributed to the Decision's total deposit pool at settlement.

5.7 Price Impact

For a Buy YES trade of size d:

```
marginal_price_before = noReserve / (yesReserve + noReserve)
execution_price       = (noReserve + d) / (yesReserve + noReserve + d)
price_impact          = execution_price − marginal_price_before
                      = d * yesReserve / [(yesReserve + noReserve)(yesReserve + no
```

Worked example: Pool at (100, 100), buy YES with d = 10:

```
marginal_price_before = 100/200 = 0.500
execution_price = 110/210 = 0.5238
price_impact = 10 * 100 / (200 * 210) = 0.0238 (2.38%)
YES_out = 10 * 210/110 = 19.09
```

5.8 Pool Initialization

When a Proposal is added to a Decision, its virtual CPMM is initialized with synthetic reserves:

```
yesReserve_0 = L
noReserve_0 = L
k = L^2
```

Where L is the **virtual liquidity depth** parameter (configured per Decision or globally). This creates a 50/50 starting probability (welfare = 0.5).

These synthetic reserves are not backed by real deposits. They represent the protocol's subsidy to bootstrap price discovery. The protocol's maximum loss from this subsidy is bounded (see Section 8.4).

---

6. Capital Accounting

6.1 The Superposition Ledger

The core data structure is a three-level accounting system:

```
Level 1: deposits[decisionId][user] = D            (real MON)
Level 2: claimed[decisionId][proposalId][user]      (vMON claimed per proposal)
Level 3: positions[decisionId][proposalId][user]    (YES/NO balances per proposal)
```

Deposit rule: When a user deposits D MON into Decision d:

```
deposits[d][user] += D
// No vMON is minted yet. The user's claimable balance increases across ALL propos
```

Claim rule: When a user claims vMON for Proposal i:

```
claimable = deposits[d][user] - claimed[d][i][user]
claimed[d][i][user] += claimable
vMON_balance[d][i][user] += claimable
```

This is the superposition step: a single deposit of D MON can be fully claimed as vMON in Proposal 0, fully claimed again in Proposal 1, etc.

Trade rule: When a user trades in Proposal i's market:

```
// Buy YES with amount `a` of vMON
vMON_balance[d][i][user] -= a
positions[d][i][user].yesBalance += YES_out
positions[d][i][user].vMonSpent += a
```

6.2 Withdrawal Before Collapse

A user may withdraw MON from a Decision before it collapses. The withdrawable amount must account for potential settlement obligations.

Conservative withdrawal rule: A user can withdraw at most:

```
withdrawable = deposits[d][user] - max_possible_loss(user, d)
```

Where max_possible_loss is the maximum amount the user could lose if any single Proposal wins and their position in that Proposal settles adversely.

For each Proposal i where the user has traded:

```
loss_i = max(0, positions[d][i][user].vMonSpent - settlement_value_if_NO_wins(posit

max_possible_loss = max(loss_i for all i where user has positions)
```

Since only one Proposal can win, the user faces at most one Proposal's loss. The maximum across all Proposals is the binding constraint.

Simplified rule (recommended for v1): Disallow withdrawal once the user has any active position in any Proposal. This eliminates complex max-loss calculations at the cost of capital flexibility.

6.3 Late Deposits and New Proposals

A user who deposits additional MON after already trading in some Proposals can: 1. Claim additional vMON in Proposals they haven't traded in (full new amount) 2. Claim additional vMON in Proposals they've already traded in (the delta between new deposit and previously claimed amount) 3. Trade with the new vMON normally

A new Proposal added to an active Decision makes all existing deposits automatically claimable for that Proposal. No action required from existing depositors – they simply claim vMON for the new Proposal when ready to trade.

---

7. Decision Lifecycle

7.1 State Machine

Mode A (Proportional TWAP):

```
OPEN ──> COLLAPSED ──> SETTLED
```

Mode B (Outcome-Based):

```
OPEN ──> COLLAPSED ──> MEASURING ──> RESOLVED ──> SETTLED
                                          |
                                  (DISPUTED ──> RESOLVED)
```

OPEN: Deposits accepted. Proposals can be added. Trading active on all Proposal markets.

COLLAPSED: Trading frozen. Winning Proposal selected based on TWAP welfare. Baseline welfare metric snapshot (Mode B only). No more deposits or trades.

MEASURING (Mode B only): Capital locked. The winning proposal's governance action takes effect in the real world. Wait measurementPeriod blocks for outcome to materialize.

RESOLVED (Mode B only): Oracle reports actual welfare metric. Outcome is YES (metric improved) or NO (metric declined). Dispute window opens. If no dispute within window, auto-finalizes.

DISPUTED (Mode B only): A challenger has posted a dispute bond contesting the oracle outcome. Enters dispute resolution flow (guardian multisig, optimistic oracle, or multi-oracle consensus). Resolves back to RESOLVED with either the original or overridden outcome.

SETTLED: Users can claim payouts. Winning market settles (proportionally in Mode A, binary in Mode B); losing markets revert.

7.2 Phase: OPEN

Creating a Decision:

```
createDecision(title, welfareMetricName, durationInBlocks, virtualLiquidity, minPr
  —> Decision created with status OPEN
  —> deadline = block.number + durationInBlocks
```

Adding a Proposal:

```
addProposal(decisionId, title, description)
  —> Requires msg.value >= minProposalBond
  —> Initializes virtual CPMM with (L, L) reserves
  —> Bond returned if proposal receives >= minTradeVolume before collapse
  —> Bond slashed if proposal receives zero trades (spam deterrent)
```

Depositing:

```
deposit(decisionId)
  —> msg.value transferred to contract
  —> deposits[decisionId][msg.sender] += msg.value
```

Trading:

```
buyYes(decisionId, proposalId, amount, minYesOut)
  —> Claims vMON if needed
  —> Executes buy-YES via CPMM
  —> Reverts if YES_out < minYesOut (slippage protection)
  —> Updates TWAP oracle
```

7.3 Phase: COLLAPSED

Triggered by calling collapse(decisionId) after block.number > deadline.

Collapse procedure:

1. Read TWAP welfare for each Proposal (see Section 9)
2. Select the Proposal with the highest TWAP welfare
3. If no Proposal exceeds a minimum welfare threshold, the Decision resolves to "status quo" (all markets revert, all deposits returned)
4. Record winningProposalId
5. Set status to COLLAPSED

7.4 Phase: SETTLED

Users call settle(decisionId) to claim their payout.

For the winning Proposal: The user's position settles as a standard prediction market. Their payout depends on whether they held YES or NO tokens and the resolution outcome.

For all losing Proposals: All trades are unwound. The user recovers their full deposit as if they had never traded in those markets.

Settlement calculation (detailed in Section 8):

```
payout = deposits[d][user] + pnl_from_winning_proposal(user)
```

Where pnl_from_winning_proposal can be positive (profit) or negative (loss), but payout >= 0 always.

---

8. Settlement and Solvency

8.1 The Two Resolution Questions

Settlement involves two distinct questions that must not be conflated:

Question 1: Which proposal wins? Determined by highest TWAP welfare at the deadline. This is always immediate and oracle-free. The TWAP mechanism (Section 9) reads the time-weighted average of each proposal's YES price and selects the leader. Losing proposals fully revert.

Question 2: How do YES/NO tokens settle within the winning proposal? This is where resolution mode matters. The contract supports three modes via a resolutionMode flag on each Decision.

8.2 Resolution Mode A: Proportional TWAP (Hackathon/MVP)

This is the primary mechanism for demo and early deployment. No oracle. No waiting period. Resolves instantly at collapse.

Core idea: The winning proposal's final TWAP welfare price W becomes the payout ratio. YES tokens pay W per token. NO tokens pay (1 − W) per token.

```
W = twap_welfare(winningProposal)      // e.g., 0.72

YES_payout_per_token = W               // 0.72 vMON per YES
NO_payout_per_token  = 1 - W           // 0.28 vMON per NO
```

Why this works: Traders are betting on where the TWAP converges, not on a guaranteed outcome. If you buy YES at 0.40 and the TWAP settles at 0.72, you profit (bought at 0.40, redeemed at 0.72). If you buy YES at 0.90 and TWAP settles at 0.72, you lose. There is genuine price discovery because the payout is uncertain until the TWAP finalizes.

Why buying YES everywhere is NOT free: Unlike the broken "auto-resolve YES" approach, proportional TWAP means YES does not always pay 1.0. If you buy YES at 0.80 and the TWAP ends at 0.65, you lose 0.15 per token. The risk is real. NO tokens also have positive expected value when the welfare price is overheated. Both sides of the book have natural demand.

PnL for the winning proposal:

```
pnl_w(user) = (positions[w][user].yesBalance * W)
            + (positions[w][user].noBalance * (1 - W))
            - positions[w][user].vMonSpent
```

For all non-winning proposals:

```
pnl_i(user) = 0    (for all i != w)
```

All trades unwind. Full deposit recoverable.

Total payout:

```
payout(user) = deposits[d][user] + pnl_w(user)
```

If the user never traded in the winning proposal, pnl_w = 0 and they receive their full deposit.

The demo story: "The crowd traded for 10 minutes. The market converged on Proposal A at 74% welfare. Traders who bought YES below 0.74 profited. Traders who bought YES above 0.74 overpaid and lost. The DAO now has a market-priced signal that Proposal A is the highest-impact action."

8.3 Resolution Mode B: Outcome-Based (Production / True Hanson Futarchy)

This is the real deal. The proposal passes, then you wait some period (days, weeks), measure a real welfare metric (token price, TVL, revenue), and pay out bets based on whether the metric actually went up. This produces the highest-quality governance signal of any mechanism known, but it takes time to resolve and you cannot demo it in 3 minutes.

8.3.1 Extended State Machine Mode B introduces a MEASURING phase between COLLAPSED and SETTLED:

```
OPEN —> COLLAPSED —> MEASURING —> RESOLVED —> SETTLED
                                      |
                             (DISPUTED —> RESOLVED)
```

- **OPEN**: Trading active. Same as Mode A.
- **COLLAPSED**: Winning proposal selected by TWAP. Governance action is executed or signaled. Baseline metric is snapshot.
- **MEASURING**: Capital locked. The real world plays out. The winning proposal's action takes effect. No trading, no withdrawals.
- **RESOLVED**: Oracle reports the actual welfare metric. Outcome is YES or NO. If disputed, enters dispute flow before finalizing.
- **SETTLED**: Users claim payouts. Same as Mode A.

8.3.2 Welfare Metric and Baseline The welfare metric must be defined at Decision creation. It is a quantifiable, oracle-readable value. Examples:

Decision	Welfare Metric	Oracle Source
"Hire Alice as Protocol Lead"	Token price (30-day TWAP post-collapse)	Pyth / Chainlink price feed
"Allocate 500K to Liquidity Mining"	Protocol TVL (measured 14 days post-collapse)	DefiLlama oracle or custom indexer
"Migrate to Monad from Ethereum"	Daily active addresses (measured 30 days post-migration)	On-chain counter contract
"Increase Staking APY to 12%"	Total staked tokens (measured 7 days post-implementation)	Staking contract view function

Baseline capture: At the moment of collapse, the contract (or a keeper) snapshots the current welfare metric value M_baseline. This is the reference point.

```
createDecision(
  title,
  welfareMetricName,
  welfareOracleAddress,       // address of the oracle contract
  welfareOracleSelector,      // function selector to call for metric value
  measurementPeriod,          // blocks to wait after collapse
  durationInBlocks,
  virtualLiquidity,
  minProposalBond,
  resolutionMode              // MODE_A | MODE_B | MODE_C
)
```

At collapse:

```
collapse(decisionId)
  --> Select winning proposal via TWAP
  --> M_baseline = IWelfareOracle(oracleAddress).getMetric()
  --> Store M_baseline on-chain
  --> Set status to COLLAPSED
  --> Set measuringDeadline = block.number + measurementPeriod
```

8.3.3 Measurement and Resolution After measurementPeriod blocks, anyone can call resolve():

```
resolve(decisionId)
  --> Requires block.number > measuringDeadline
  --> Requires status == COLLAPSED (or MEASURING)
  --> M_actual = IWelfareOracle(oracleAddress).getMetric()
  --> If M_actual >= M_baseline:
        outcome = YES    // The proposal delivered -- welfare improved or held
  --> If M_actual < M_baseline:
        outcome = NO     // The proposal failed -- welfare declined
  --> Store outcome on-chain
  --> Set status to RESOLVED
  --> Start disputeWindow (optional, see 8.3.5)
```

Threshold design: The simplest threshold is M_baseline itself – did the metric improve at all? For more nuance, the Decision creator can set a minImprovement parameter (e.g., "metric must improve by at least 5%"):

```
If M_actual >= M_baseline * (1 + minImprovement):
  outcome = YES
Else:
  outcome = NO
```

Default minImprovement = 0 (any improvement counts).

8.3.4 Payout Formulas Binary resolution. The winning proposal's YES/NO tokens settle based on the outcome:

```
If outcome = YES:
  YES pays 1.0 vMON per token
  NO  pays 0.0 vMON per token

  pnl_w(user) = positions[w][user].yesBalance − positions[w][user].vMonSpent

If outcome = NO:
  YES pays 0.0 vMON per token
  NO  pays 1.0 vMON per token

  pnl_w(user) = positions[w][user].noBalance − positions[w][user].vMonSpent
```

For all non-winning proposals: Same as Mode A. All trades unwind, pnl = 0.

Total payout:

```
payout(user) = deposits[d][user] + pnl_w(user)
```

Why this changes the trading game: In Mode A, traders bet on where the crowd converges (Schelling game). In Mode B, traders bet on whether the proposal will actually deliver real-world results. This is fundamentally harder, more valuable, and attracts a different class of participant – analysts, insiders, domain experts, and AI agents who model real outcomes.

A trader buying YES on "Allocate 500K to Liquidity Mining" is expressing the belief: "If we do this, TVL will go up." If they're right (TVL rises), they profit. If they're wrong (TVL drops despite the allocation), they lose. The market aggregates these beliefs into a price that IS the crowd's expected impact – genuine futarchy.

8.3.5 Dispute Resolution Oracle-resolved markets need a safety valve. If the oracle reports a clearly wrong value (feed malfunction, stale data, manipulation), there must be a mechanism to challenge the result.

Dispute flow:

```
RESOLVED ──> dispute(decisionId, bond)
  ──> Requires block.number < resolvedAt + disputeWindow
  ──> Requires msg.value >= disputeBond
  ──> Set status to DISPUTED
  ──> Escalate to dispute resolution mechanism
```

Dispute resolution options (configurable per Decision):

Option 1: DAO Vote (Simple) A guardian multisig or token-weighted vote can override the oracle outcome within the dispute window. Best for early production where trust in a small council is acceptable.

```
resolveDispute(decisionId, overrideOutcome)
  ──> Requires msg.sender == guardianMultisig
  ──> Override outcome to YES or NO
  —> Refund dispute bond to challenger
  —> Set status to RESOLVED
```

Option 2: Optimistic Oracle (UMA-style) The disputer proposes an alternative outcome. If no counter-dispute within a challenge period, the alternative is accepted. If counter-disputed, escalates to a decentralized jury (UMA DVM, Kleros, or similar).

Option 3: Multi-Oracle Consensus Query multiple independent oracles (e.g., Pyth + Chainlink + on-chain TWAP). Outcome is the majority. Disputes are only valid if oracles disagree.

Recommendation: Option 1 for v1 (guardian multisig). Migrate to Option 2 for v2. The contract interface is the same regardless – resolveDispute(decisionId, outcome) – only the access control changes.

Dispute parameters:

Parameter	Recommended Value	Rationale
disputeWindow	48 hours (in blocks)	Enough time for the community to notice a bad oracle report
disputeBond	5% of total Decision deposits	High enough to deter spam, low enough to be accessible
bondRefund	Full refund if dispute succeeds, slashed if fails	Skin in the game for disputers

If no dispute is filed within the window, the outcome auto-finalizes and status moves to SETTLED.

8.3.6 Capital Lockup and Early Exit Capital is locked from collapse through measurement and resolution. This is the fundamental tradeoff of Mode B – accuracy requires patience.

Lockup timeline (example with recommended parameters):

```
Day 0:           Decision created, trading begins
Day 2:           Deadline hits, collapse() called, baseline snapshot
Day 2 - Day 16:  Measurement period (14 days). No trading, no withdrawals.
Day 16:          resolve() callable. Oracle reports M_actual.
Day 16 - Day 18: Dispute window (48 hours).
Day 18:          If no dispute, status = SETTLED. Users claim payouts.

Total capital lockup: ~18 days from collapse.
```

Early exit (future extension): A secondary market for positions in MEASURING-phase Decisions could allow users to exit early by selling their claim to another user. This is not in v1 scope but the position data structure supports it (positions are per-user, transferable in principle).

8.3.7 Oracle Interface The contract interacts with the welfare oracle via a minimal interface:

```
interface IWelfareOracle {
  // Returns the current welfare metric value
  // Scaled to 18 decimals (e.g., token price in USD * 1e18)
  function getMetric() external view returns (uint256);

  // Returns the timestamp of the last metric update
  function lastUpdated() external view returns (uint256);
}
```

Staleness check: The resolve function should revert if the oracle data is stale:

```
resolve(decisionId)
  --> require(oracle.lastUpdated() >= measuringDeadline, "Oracle data stale")
```

This prevents resolution with outdated data. If the oracle is down, the Decision stays in MEASURING until fresh data is available (or the guardian intervenes).

Adapter pattern: Different oracle providers (Pyth, Chainlink, custom) have different interfaces. Each gets a thin adapter contract that implements IWelfareOracle. The Decision stores the adapter address, not the raw oracle address.

```
PythAdapter implements IWelfareOracle
  --> Reads Pyth price feed
  --> Converts to 18-decimal uint256
  --> Returns via getMetric()

ChainlinkAdapter implements IWelfareOracle
  --> Reads Chainlink AggregatorV3
  --> Scales to 18 decimals
  --> Returns via getMetric()

OnChainMetricAdapter implements IWelfareOracle
  --> Reads any on-chain view function (TVL, staked amount, etc.)
  --> Returns via getMetric()
```

8.3.8 Solvency Proof (Mode B) **Theorem**: Under binary resolution, the winning market is zero-sum among participants. Total payouts equal total deposits (plus or minus virtual liquidity subsidy).

Proof:

For the winning proposal w, every vMON that entered the market was used to mint YES/NO pairs. Each pair minted from 1 vMON produces exactly 1 YES + 1 NO.

If outcome = YES: all YES tokens pay 1, all NO tokens pay 0.

```
sum(payouts_w) = sum(yesBalance_w) = YES_supply_w
```

Since YES_supply_w = total vMON minted into token pairs, and all vMON either circulates as vMON or is locked in token pairs:

```
vMON_circulating_w + YES_supply_w = vMON_minted_w
```

Total payouts from the winning market = vMON_circulating_w + YES_supply_w = vMON_minted_w.

Since vMON_minted_w <= D_total, and non-winning proposals have pnl = 0:

```
sum(all payouts) = D_total + sum(pnl_w) = D_total + 0 = D_total
```

QED. Same zero-sum property as Mode A, just with binary instead of proportional distribution.

Note: If outcome = NO, the proof is symmetric (NO pays 1, YES pays 0). The sum is identical.

8.3.9 Worked Settlement Example (Mode B) **Setup**: Decision "Should we allocate 100K MON to liquidity mining on DEX X?"

- Welfare metric: Protocol TVL (measured via on-chain adapter)
- Measurement period: 14 days (~3,024,000 blocks at 400ms)
- Virtual liquidity: L = 100 per Proposal
- Two proposals: "Allocate" vs "Status Quo"

Participants:

- Alice deposits 50 MON, buys 30 vMON of YES on "Allocate" at avg price 0.55. Gets 53.2 YES tokens.
- Bob deposits 30 MON, buys 20 vMON of NO on "Allocate" at avg price 0.45. Gets 42.8 NO tokens.
- Charlie deposits 20 MON, buys 10 vMON of YES on "Status Quo" at avg price 0.51. Gets 19.5 YES tokens.

Collapse: "Allocate" has highest TWAP welfare (0.62). "Allocate" wins.

- M_baseline = current TVL = 2,400,000 MON
- Governance action signaled: allocate 100K MON to DEX X liquidity mining
- Charlie's trade in "Status Quo" unwinds (pnl = 0)

14 days later: The liquidity mining program ran. TVL grew.

- M_actual = 3,100,000 MON
- M_actual (3.1M) >= M_baseline (2.4M) –> outcome = **YES**

Settlement:

- Alice: pnl = 53.2 * 1.0 - 30 = **+23.2**. Payout = 50 + 23.2 = **73.2 MON**
  - (Alice correctly predicted the allocation would grow TVL)
- Bob: pnl = 42.8 * 0.0 - 20 = **-20.0**. Payout = 30 - 20 = **10.0 MON**
  - (Bob bet against the allocation, was wrong)
- Charlie: pnl = 0. Payout = **20.0 MON**
  - (Never traded in winning proposal)

Check: 73.2 + 10.0 + 20.0 = 103.2 MON. Deposits = 100 MON. The 3.2 MON excess comes from the protocol's virtual liquidity subsidy (bounded by L).

Alternate scenario – the allocation fails:

If TVL had instead dropped to 2,000,000 MON: - M_actual (2.0M) < M_baseline (2.4M) –> outcome = **NO** - Alice: pnl = 53.2 * 0.0 - 30 = **-30.0**. Payout = 50 - 30 = **20.0 MON** - Bob: pnl =

42.8 * 1.0 - 20 = **+22.8**. Payout = 30 + 22.8 = **52.8 MON** - Charlie: pnl = 0. Payout = **20.0 MON**

Bob profits because the crowd was wrong about the allocation's impact. This is the accountability mechanism at work – the market punishes collective overconfidence.

8.3.10 Why Mode B Is the Endgame Mode A is a coordination game. Mode B is a truth machine.

In Mode A, the welfare price reflects what the crowd *thinks* the crowd thinks – a Keynesian beauty contest. It is useful, it is fast, and it demonstrates the mechanism. But it is self-referential.

In Mode B, the welfare price reflects what the crowd *knows* about the real world. Traders are rewarded for predicting actual outcomes, not for predicting other traders' predictions. This is Hanson's original vision: governance by information aggregation, where the information is about reality, not about the market itself.

The progression is clear: Mode A to get adoption, Mode B to get truth. The contract architecture supports both from day one.

8.4 Resolution Mode C: Hybrid (Fast Feedback Loop)

A middle ground for early production use when full oracle resolution is too slow but proportional TWAP is too self-referential.

Mechanism: Use a short, measurable proxy metric that resolves in hours rather than weeks. Examples:

- "Which logo should we use?" – Welfare metric = on-chain poll result 1 hour after collapse
- "Which chain should we deploy to?" – Welfare metric = community snapshot vote within 24 hours
- "Should we increase emissions?" – Welfare metric = TVL change over 48 hours post-implementation

The contract architecture is identical to Mode B but with shorter measurement periods and simpler oracles (even a multisig can suffice for early governance).

8.5 Solvency Proof (Mode A)

Theorem: Under proportional TWAP resolution, total payouts across all users equal total deposits into the Decision (excluding virtual liquidity subsidy). The winning market is zero-sum among participants.

Proof:

Let D_total = sum of all user deposits = total real MON in the contract.

For the winning proposal w with final TWAP welfare W:

Each token pair (1 YES + 1 NO) was minted from 1 vMON. At settlement, that same pair pays out:

$$\text{YES\_payout} + \text{NO\_payout} = W + (1 - W) = 1 \text{ vMON}$$

Therefore, every vMON that entered the winning market as a token mint exits as exactly 1 vMON in payouts, regardless of W. The value is conserved – it is merely redistributed between YES and NO holders based on where W lands relative to their entry prices.

For non-winning proposals, all trades unwind: pnl = 0.

Therefore:

```
sum(payouts) = D_total + sum(pnl_w) = D_total + 0 = D_total
```

The contract holds exactly D_total real MON and must pay out exactly D_total. QED.

Note: This proof is cleaner than the binary resolution case because proportional payout guarantees YES + NO = 1 by construction. There is no scenario where the wrong side pays 0 and creates a surplus or deficit among participants.

8.6 Virtual Liquidity Solvency

The virtual CPMM is initialized with synthetic reserves (L, L) not backed by real deposits. This means YES/NO tokens in the pool at initialization represent a protocol subsidy.

Protocol exposure: The protocol's virtual position is equivalent to an LP position of (L, L). At settlement, if the price has moved from 0.5 to some extreme, the protocol's position has experienced divergence loss.

Maximum protocol loss from virtual liquidity (Mode A):

The protocol holds virtual YES and NO tokens in the pool. At settlement with final reserves (x_f, y_f) where x_f * y_f = L^2:

```
Protocol payout = x_f * W + y_f * (1 - W)
```

Where W = y_f / (x_f + y_f) (the welfare price from reserves).

Protocol initially contributed L (value at 50/50 initialization). The divergence loss is bounded:

Worst-case protocol loss: L (the entire virtual liquidity contribution). This occurs when W approaches 0 or 1.

This must be funded from either: - A protocol treasury reserve - Fees collected from trading (fees increase k, increasing the protocol's reserves) - The proposal bond (if sized appropriately)

Recommendation: Set L conservatively. A virtual liquidity of L = 1000 * 10^18 (1000 MON equivalent) with a trading fee of 0.3% should be self-funding after moderate trading volume.

8.7 Worked Settlement Example (Mode A)

Setup: Decision with 3 Proposals. Virtual liquidity L = 100 per Proposal. Resolution Mode A (proportional TWAP).

- Alice deposits 50 MON
- Bob deposits 30 MON
- Charlie deposits 20 MON
- Total deposits: 100 MON

Trading:

- Alice buys 20 vMON of YES on Proposal A at avg price 0.52. Gets 38.1 YES tokens.
- Alice buys 10 vMON of YES on Proposal B at avg price 0.51. Gets 19.5 YES tokens.
- Bob buys 15 vMON of NO on Proposal A at avg price 0.52. Gets 28.85 NO tokens.
- Charlie buys 10 vMON of YES on Proposal C at avg price 0.51. Gets 19.5 YES tokens.

Collapse: Proposal A has highest TWAP welfare at W = 0.65.

Settlement:

Proposal A settles proportionally: YES pays 0.65, NO pays 0.35.

- Alice: pnl_A = (38.1 * 0.65) - 20 = 24.77 - 20 = **+4.77**
  - Payout = 50 + 4.77 = **54.77 MON**
  - (Alice bought YES cheap and it settled above her entry – profit)
- Bob: pnl_A = (28.85 * 0.35) - 15 = 10.10 - 15 = **-4.90**
  - Payout = 30 - 4.90 = **25.10 MON**
  - (Bob bought NO but welfare ended at 0.65, so NO only paid 0.35 – loss)
- Charlie: pnl_A = 0 (never traded in Proposal A)
  - Payout = 20 + 0 = **20.00 MON**

Note: Alice's trade in Proposal B unwinds (pnl = 0). Charlie's trade in Proposal C unwinds (pnl = 0).

Check: 54.77 + 25.10 + 20.00 = 99.87 MON.

The 0.13 MON discrepancy is the net gain to the protocol's virtual liquidity position (fees collected exceeded divergence loss in this example). In cases where divergence loss exceeds fees, the protocol reserve covers the difference, always bounded by L.

---

9. TWAP Oracle Design

9.1 Why TWAP

Spot price at the deadline is trivially manipulable: a single large trade in the final block can swing the winner. A Time-Weighted Average Price smooths over a window, requiring sustained capital commitment to manipulate.

MetaDAO's experience confirms this: their lagging TWAP design prevented $250,000 of manipulation capital from overriding genuine price discovery.

9.2 Oracle Architecture

For each Proposal i, the oracle maintains:

```
OracleState_i = {
  priceCumulative: uint256,    // running sum of (price * blocks_elapsed)
  lastUpdateBlock: uint256,    // last block this was updated
  lastObservation: uint256,    // last recorded welfare price (lagged)
}
```

9.3 Observation Update (Lagging Price)

On every trade in Proposal i's market, before the trade executes:

```
function updateOracle(proposalId):
  blocks_elapsed = block.number - oracle.lastUpdateBlock
  if blocks_elapsed == 0: return  // one update per block max

  // Get raw welfare from CPMM
  raw_welfare = noReserve / (yesReserve + noReserve)

  // Apply lagging: observation can move at most MAX_CHANGE per block
  delta = raw_welfare - oracle.lastObservation
  clamped_delta = clamp(delta, -MAX_CHANGE_PER_BLOCK, MAX_CHANGE_PER_BLOCK)
  new_observation = oracle.lastObservation + clamped_delta * blocks_elapsed

  // Clamp to valid range
  new_observation = clamp(new_observation, 0, 10000)

  // Accumulate
  oracle.priceCumulative += oracle.lastObservation * blocks_elapsed
  oracle.lastObservation = new_observation
  oracle.lastUpdateBlock = block.number
```

9.4 Welfare Reading at Collapse

At collapse, the TWAP welfare for Proposal i over the observation window [start_block, end_block] is:

```
twap_welfare_i = (priceCumulative_end - priceCumulative_start) / (end_block - start
```

Where start_block = deadline − TWAP_WINDOW and end_block = deadline.

The Proposal with the highest twap_welfare_i wins.

9.5 Parameter: MAX_CHANGE_PER_BLOCK

This controls the tradeoff between manipulation resistance and price responsiveness.

On Monad with 400ms blocks: - 1 block = 0.4 seconds - 2,500 blocks = 1000 seconds (~17 minutes)

If MAX_CHANGE_PER_BLOCK = 4 basis points (0.04%): - Price can move at most 0.04% per block - To move price from 50% to 60% (1000 bps) requires 250 blocks (100 seconds) - An attacker must sustain a manipulated pool price for 100+ seconds

If MAX_CHANGE_PER_BLOCK = 1 basis point (0.01%): - To move from 50% to 60% requires 1000 blocks (400 seconds = ~7 minutes) - More manipulation resistant but slower to reflect genuine price changes

Recommendation: MAX_CHANGE_PER_BLOCK = 2 basis points. This means a full 50% -> 100% swing requires 2500 blocks (~17 minutes). Combined with a TWAP window of 25,000 blocks (~2.8 hours), manipulation requires sustained commitment over hours, not seconds.

9.6 Parameter: TWAP_WINDOW

The observation window for computing the final TWAP at collapse.

Recommendation: TWAP_WINDOW = 25,000 blocks (~2.8 hours on Monad). This means the collapse decision is based on the average welfare over the final ~3 hours of trading. Short-term spikes are averaged out.

9.7 Anti-Cranking Protection

Unlike Solana (where TWAP requires "cranking" by external agents), Monad's EVM model allows the oracle to self-update on every trade. However, if a Proposal receives no trades for an extended period, the oracle becomes stale.

Stale oracle handling: At collapse, if a Proposal's oracle has not been updated within TWAP_WINDOW / 2 blocks, its welfare is set to the initial value (0.5) or excluded from consideration. This prevents a dormant Proposal with a stale high-welfare reading from winning.

---

10. Security Analysis

10.1 Threat Model

We assume: - Adversaries have significant capital (up to 50% of total Decision deposits) - Adversaries can execute transactions in every Monad block (400ms) - Adversaries can observe the mempool (no global mempool on Monad, but block leaders receive transactions directly) - Multiple adversaries may collude - Smart contract bugs are out of scope (addressed by audits)

10.2 Attack: Last-Block Price Manipulation

Vector: Large trade in the final block before deadline to swing a Proposal's welfare.

Defense: The lagging TWAP oracle (Section 9.3) limits price movement to MAX_CHANGE_PER_BLOCK per block. A single-block trade can move the observation by at most 2 bps. With a TWAP window of 25,000 blocks, the impact on the average is 2/25000 = 0.00008 bps – negligible.

Residual risk: None, if the TWAP window is sufficiently long.

10.3 Attack: Sustained TWAP Manipulation

Vector: An attacker continuously trades to push a Proposal's welfare above competitors, sustaining the manipulated price over the entire TWAP window.

Defense (structural): To sustain a manipulated price, the attacker must continuously trade against arbitrageurs. In quantum futarchy, this defense is stronger than in classical futarchy because:

1. Counter-traders can deploy their FULL deposit against the manipulation (no capital fragmentation)
2. If the manipulation fails (wrong Proposal wins), the attacker's trades in the manipulated Proposal are unwound – but their counter-trades in OTHER Proposals are also unwound. The attacker loses only in the winning Proposal.

3. Counter-traders face zero opportunity cost: trading against manipulation in one Proposal does not reduce their ability to trade in others.

Defense (economic): Let C be the cost of manipulation (capital needed to sustain a price deviation of delta over the TWAP window). In a CPMM:

```
C ~ delta^2 * k / 4  (for small delta relative to reserves)
```

With virtual liquidity L = 1000 MON and real deposits of D_total = 10,000 MON:

```
k ~ (L + D_total/2)^2 ~ 6000^2 = 36,000,000
C_to_move_5% ~ 0.05^2 * 36M / 4 ~ 22,500 MON
```

At 10,000 MON total deposits, manipulating one Proposal by 5% requires ~2.25x the total pool size. This is prohibitively expensive.

Residual risk: Low for well-capitalized Decisions. High for thin Decisions with minimal deposits.

10.4 Attack: Proposal Spam

Vector: Creating hundreds of Proposals to dilute attention and obfuscate meaningful choices.

Defense: 1. **Proposal bond**: A configurable bond (e.g., 1 MON) required to create a Proposal. Returned if the Proposal receives at least MIN_TRADE_VOLUME in trading activity. Slashed if zero trades (pure spam). 2. **UI-level filtering**: Frontend shows only Proposals ranked by trading volume or welfare deviation from 0.5. 3. **Maximum Proposals per Decision**: A hard cap (e.g., 20) prevents unbounded gas costs during collapse.

10.5 Attack: Cross-Proposal Manipulation

Vector: An attacker suppresses welfare on competing Proposals (by buying NO) while pumping their preferred Proposal (by buying YES). Since capital is shared, this costs nothing extra.

Defense: This is actually the intended mechanism. Buying NO on a Proposal means "I think this Proposal is bad" – it IS the governance signal. If an attacker spends capital suppressing good Proposals, counter-traders profit by buying YES on those Proposals cheaply.

The key insight: the attacker's capital is finite. Pushing down welfare on Proposal B while pushing up welfare on Proposal A requires trading in both markets. Counter-traders can oppose both manipulations simultaneously using the same deposit.

Residual risk: If the attacker has overwhelming capital and no counter-traders participate, they can dictate outcomes. This is a liveness assumption shared with all market-based governance: markets require informed participants.

10.6 Attack: Sandwich/Frontrunning

Vector: MEV extraction by frontrunning trades in Proposal markets.

Defense: 1. **Slippage protection**: Every trade includes a minOut parameter. If frontrunning would push the price beyond the user's tolerance, the transaction reverts. 2. **Monad's no-global-mempool**: Transactions go directly to upcoming block leaders, reducing the exposure window compared to Ethereum's public mempool. 3. **Fee impact**: With a 0.3% fee, the profitability of sandwiching small trades is minimal.

10.7 Attack: Sybil Proposal Collision

Vector: Attacker creates multiple near-identical Proposals to split counter-trader attention.

Defense: Counter-traders can trade in ALL Proposals at no additional capital cost (the quantum property). If the attacker creates 5 similar Proposals and pumps one, counter-traders can sell YES on all 5 simultaneously.

10.8 Minimum Activity Threshold

A Proposal with very few trades could win if its welfare happens to be high due to noise rather than genuine information.

Defense: Require a minimum trade volume threshold for a Proposal to be eligible for winning:

```
eligible_i = (totalVolumeTraded_i >= MIN_VOLUME_THRESHOLD)
```

If no Proposal meets the threshold, the Decision resolves to status quo (all deposits returned).

Recommended: MIN_VOLUME_THRESHOLD = 5% of totalDeposits.

10.9 Tiebreaking

If two Proposals have identical TWAP welfare at collapse:

Rule: The Proposal with higher total trading volume wins. If still tied, the earlier Proposal (lower ID) wins. This incentivizes early and active market participation.

---

11. Contract Architecture (Monad-Optimized)

11.1 Design Principles

1. **Singleton pattern**: A single MeridianCore contract holds all deposits and manages all Decisions. This avoids factory-pattern gas overhead and simplifies accounting.
2. **Internal accounting**: YES/NO/vMON are tracked as storage mappings, not as separate ERC-20 contracts. This saves ~30K gas per Proposal creation (no contract deployment).
3. **Storage isolation**: Per-Decision and per-Proposal state is stored in separate mapping branches to maximize Monad's parallel execution benefit.
4. **Claim-based settlement**: Users pull funds (call `settle()`) rather than the protocol pushing. This enables parallel execution during the settlement phase.

11.2 Storage Layout

```
// Level 1: Decision state
mapping(uint256 => Decision) decisions;
uint256 nextDecisionId;

// Level 2: Proposal state (per Decision)
mapping(uint256 => mapping(uint256 => Proposal)) proposals;
mapping(uint256 => uint256) proposalCount;
```

```
// Level 3: Oracle state (per Proposal)
mapping(uint256 => mapping(uint256 => OracleState)) oracles;

// Level 4: User state
mapping(uint256 => mapping(address => uint256)) deposits;
// [decisionId][user]
mapping(uint256 => mapping(uint256 => mapping(address => uint256))) claimed;
// [decisionId][proposalId][user]
mapping(uint256 => mapping(uint256 => mapping(address => Position))) positions; //

// Level 5: Settlement state
mapping(uint256 => mapping(address => bool)) settled;  // [decisionId][user]
```

Monad parallel execution note: Trades on different Proposals within the same Decision touch different proposals[d][i] storage slots. Monad's optimistic execution will parallelize these. Trades on the same Proposal will serialize (they modify the same yesReserve/noReserve slots). This is acceptable – within a single Proposal, trades MUST be ordered to maintain CPMM consistency.

11.3 Function Signatures (Pseudocode)

```
// ============ DECISION LIFECYCLE ============

function createDecision(
    title: string,
    welfareMetricName: string,
    durationInBlocks: uint256,
    virtualLiquidity: uint256,
    minProposalBond: uint256
) -> decisionId: uint256

function addProposal(
    decisionId: uint256,
    title: string,
    description: string
) payable -> proposalId: uint256
    // Requires msg.value >= minProposalBond
    // Initializes CPMM with (virtualLiquidity, virtualLiquidity)

// ============ CAPITAL ============

function deposit(decisionId: uint256) payable
    // Adds msg.value to deposits[decisionId][msg.sender]

function withdraw(decisionId: uint256, amount: uint256)
    // Only if user has no active positions (simplified rule)
    // Reduces deposits[decisionId][msg.sender]

// ============ TRADING ============
function buyYes(
    decisionId: uint256,
    proposalId: uint256,
    amount: uint256,        // vMON to spend
    minYesOut: uint256      // slippage protection
)
    // 1. Auto-claim vMON if needed
    // 2. Execute CPMM buy-YES
    // 3. Update oracle
    // 4. Emit Trade event

function buyNo(
    decisionId: uint256,
    proposalId: uint256,
    amount: uint256,
    minNoOut: uint256
)

function sellYes(
    decisionId: uint256,
    proposalId: uint256,
    yesAmount: uint256,     // YES tokens to sell
    minVmonOut: uint256     // minimum vMON to receive
)

function sellNo(
    decisionId: uint256,
    proposalId: uint256,
    noAmount: uint256,
    minVmonOut: uint256
)

// ============ RESOLUTION ============

function collapse(decisionId: uint256)
    // Callable by anyone after deadline
    // Reads TWAP welfare for each proposal
    // Selects winner (highest TWAP, above threshold, with min volume)
    // Sets status = COLLAPSED

function settle(decisionId: uint256)
    // Callable by any user with a deposit
    // Calculates payout based on winning proposal position
    // Transfers real MON to user
    // Sets settled[decisionId][msg.sender] = true

// ============ VIEW ============
function getDecision(decisionId) -> Decision
function getProposal(decisionId, proposalId) -> Proposal
function getPosition(user, decisionId, proposalId) -> Position
function getWelfare(decisionId, proposalId) -> uint256  // current spot welfare (bp
function getTwapWelfare(decisionId, proposalId) -> uint256  // TWAP welfare (bps)
function getUserDeposit(user, decisionId) -> uint256
function getClaimable(user, decisionId, proposalId) -> uint256
```

11.4 Events

```
event DecisionCreated(decisionId, creator, title, deadline)
event ProposalAdded(decisionId, proposalId, proposer, title)
event Deposited(user, decisionId, amount)
event Withdrawn(user, decisionId, amount)
event Trade(user, decisionId, proposalId, direction: YES|NO, amountIn, amountOut, n
event Collapsed(decisionId, winningProposalId, winningTwapWelfare)
event Settled(user, decisionId, payout, pnl)
```

11.5 Access Control

- createDecision: Permissionless (anyone can create)
- addProposal: Permissionless (requires bond)
- deposit/withdraw: Permissionless
- buyYes/buyNo/sellYes/sellNo: Permissionless (requires deposit)
- collapse: Permissionless (callable by anyone after deadline)
- settle: Permissionless (each user settles their own position)

No admin keys. No upgradeability (in v1). Governance parameters are set at Decision creation time and are immutable for that Decision.

---

12. Gas and Storage Optimization

12.1 Monad-Specific Considerations

Gas-limit charging: On Monad, users pay for the gas limit, not gas used. The frontend must estimate gas tightly for each operation. Overestimating wastes real MON.

Operation	Estimated Gas	Notes
createDecision	~200K	String storage dominates
addProposal	~150K	Initializes CPMM + oracle
deposit	~50K	Single SSTORE
buyYes	~120K	Oracle update + CPMM + position update
collapse	~50K + 30K * N	Reads N proposals' oracles
settle	~80K	Reads position + transfers MON

Cold storage repricing: Monad charges ~4x Ethereum for cold SLOAD/SSTORE. The first access to any storage slot in a transaction costs 8,100 gas (vs. 2,100 on Ethereum). Subsequent accesses to the same slot cost 100 gas (warm).

Optimization strategies:

1. **Pack related data into single slots**: Decision status + proposalCount + totalDeposits can fit in one 256-bit slot.
2. **Batch oracle updates**: If the same user trades in multiple Proposals in one transaction, update all oracles together (the Decision-level storage is already warm).
3. **Avoid unnecessary SLOADs**: Cache values in memory within a function rather than re-reading from storage.

12.2 Storage Packing

```
// Pack Decision into 3 slots instead of 8
struct Decision {
    // Slot 1: 160 + 48 + 48 = 256 bits
    address creator;         // 160 bits
    uint48 deadline;         // 48 bits (block number, sufficient for ~89 years at
    uint48 createdAtBlock;   // 48 bits

    // Slot 2: 128 + 16 + 8 + 16 + 16 = 184 bits (fits in 256)
    uint128 totalDeposits;   // 128 bits (up to 3.4 * 10^38 wei)
    uint16 proposalCount;    // 16 bits (up to 65,535 proposals)
    uint8 status;            // 8 bits (OPEN=0, COLLAPSED=1, SETTLED=2)
    uint16 winningProposalId; // 16 bits
    uint16 virtualLiquidity;  // 16 bits (scaled, e.g., in units of 0.01 MON)

    // Slot 3: string pointers (title, welfareMetricName stored separately)
}
```

12.3 Parallel Execution Optimization

Key insight: Monad parallelizes transactions that touch different storage slots. Design storage so independent operations use independent slots:

- Different users depositing into the same Decision: deposits[d][user_A] and deposits[d][user_B] are different slots. **Parallelizable.**
- Different users trading in different Proposals: positions[d][0][user_A] and positions[d][1][user_B] are different slots. proposals[d][0] and proposals[d][1] are different slots. **Parallelizable.**
- Different users trading in the SAME Proposal: Both modify proposals[d][i].yesReserve. **Serialized.** This is correct – CPMM state must be consistent.
- Multiple users settling: Each reads proposals[d][w] (shared, read-only) and writes deposits[d][user] (independent). **Mostly parallelizable** (shared read, independent writes).

12.4 Collapse Gas Optimization

Collapse must iterate over all Proposals to find the winner. With a cap of 20 Proposals per Decision, this is bounded at ~20 * 30K = 600K gas – well within Monad's 30M transaction gas limit.

For larger Proposal counts (future versions), consider a commit-reveal pattern where the oracle state is finalized incrementally before collapse.

---

13. Parameter Recommendations

13.1 Decision Parameters

Parameter	Recommended Value	Rationale
durationInBlocks	432,000 (~2 days)	Sufficient for price discovery; not so long that capital lockup is burdensome
virtualLiquidity	1,000 MON	Bootstraps price discovery without excessive protocol subsidy
minProposalBond	0.1 MON	Low enough for permissionless proposals; enough to deter spam
maxProposals	20	Bounds gas cost at collapse
minVolumeThreshold	5% of totalDeposits	Prevents noise-driven wins on untouched Proposals

13.2 Market Parameters

Parameter	Recommended Value	Rationale
swapFee	30 bps (0.3%)	Standard DeFi fee; high enough to deter wash trading
minTradeSize	0.001 MON	Prevents dust trades that waste gas

13.3 Oracle Parameters

Parameter	Recommended Value	Rationale
MAX_CHANGE_PER_BLOCK	2 bps	50%->100% takes ~17 min; manipulation-resistant yet responsive

Parameter	Recommended Value	Rationale
TWAP_WINDOW	25,000 blocks (~2.8h)	Long enough to average out manipulation; short enough for meaningful signal
STALE_THRESHOLD	12,500 blocks (~1.4h)	Proposals untouched for half the window are excluded

13.4 Status Quo Default

Every Decision automatically includes a "Status Quo" Proposal (Proposal 0) that represents "do nothing." Its welfare is initialized at 0.5 and can be traded like any other Proposal. If no Proposal (including Status Quo) meets the minimum volume threshold, the Decision is abandoned and all deposits returned.

This ensures Hanson's design property: the DAO only acts when the market judges action superior to inaction.

---

14. Open Design Decisions

14.1 Resolution Mode Selection

Resolved in Section 8. Three modes are specified: - **Mode A (Proportional TWAP)**: Hackathon/MVP. Instant resolution, no oracle. YES pays W, NO pays 1−W. - **Mode B (Outcome-Based)**: Production. True Hanson futarchy with oracle-measured welfare metric. - **Mode C (Hybrid)**: Early production. Short-lived proxy metrics with simple oracles.

Recommendation: Ship Mode A. The contract supports all three via a resolutionMode flag on each Decision. The remaining open question is Mode B's oracle selection (Pyth, Chainlink, UMA, or DAO committee) – defer until production.

14.2 Withdrawal Policy

Option A (Simple): No withdrawals once any position is open. User must wait for settlement.

Option B (Flexible): Withdrawable amount = deposit minus max potential loss across all Proposals. Requires computing worst-case settlement for each Proposal – gas-intensive but capital-friendly.

Recommendation: Option A for v1. The capital lockup duration (2 days recommended) is short enough that the simplicity benefit outweighs the flexibility cost.

14.3 Cross-Decision Capital

In the current design, each Decision has its own deposit pool. A user who wants to participate in 5 Decisions must deposit separately into each one.

Future extension: A global deposit that works in superposition across Decisions (not just across Proposals within a Decision). This is theoretically possible if Decisions have non-overlapping settlement periods, but adds significant accounting complexity.

Recommendation: Defer to v3. Per-Decision deposits are sufficient for initial deployment.

14.4 AI Agent API

Per Paradigm's thesis, quantum markets are well-suited for AI agent participation. The smart contract interface is agent-friendly by default (no UI dependencies). A dedicated off-chain API could provide:

- Real-time welfare predictions for all Proposals
- Suggested trade sizes given target welfare impact
- Risk analysis (max loss scenarios)
- Automated arbitrage against manipulation

Recommendation: Build the on-chain contracts first. The API layer can wrap the view functions with no contract changes.

14.5 Proposal Execution

When a Proposal wins, what happens on-chain?

Option A (Advisory): The Decision result is purely informational. A multisig or existing governance process executes the winning Proposal's action.

Option B (Executable): Each Proposal includes an executable payload (calldata + target address). The collapse() function executes it atomically. This is the MetaDAO model.

Recommendation: Option A for v1 (simpler, less attack surface). Option B for v2 (requires time-lock and guardian mechanisms).

---

15. Frontend Design Principles

The frontend is not a secondary concern – it is where the mechanism becomes legible. A futarchy protocol that feels like a DeFi dashboard has already failed. Meridian should feel like a **live crowd decision engine**: Polymarket's clarity meets Monad's speed.

15.1 Decision-First, Not Chart-First

The hero element is the **question and the crowd's current answer**, not a price chart. Every screen should be answerable in one glance: *"What is the crowd saying right now?"*

A proposal card shows: - The proposal title as a clear human question - A single dominant percentage (the current YES welfare price) - A directional indicator (trending up/down over last N blocks) - A colored bar representing the crowd split

Charts exist but are secondary – expandable, not default. The primary interface communicates **conviction**, not price action.

15.2 Three-Second Interaction

The critical path from “I see a proposal” to “I have a position” must complete in under 3 seconds. This is non-negotiable for demo environments and hackathon judging.

The interaction flow: 1. **See** – Proposal card with YES/NO and current crowd signal (0.5s to comprehend) 2. **Tap** – Single tap on YES or NO (0.3s) 3. **Confirm** – Pre-filled amount (0.1 MON default), one button to execute (1s for wallet confirmation) 4. **Feedback** – Instant visual confirmation with the bar animating to reflect the new state (~0.5s on Monad)

No modal chains. No token approval steps (use permit or pre-approved allowances). No amount selection screens – default amount with an optional expander for power users.

15.3 Motion as Proof of Speed

Monad’s sub-second finality is invisible unless the UI makes it visible. **Animation is the proof.** When a trade lands:

- The welfare percentage bar animates smoothly to its new value
- A ripple effect emanates from the point of interaction
- The proposal card briefly pulses to signal state change
- Other users see the bar move in real-time via WebSocket/subscription

This is the “holy shit” moment: a user taps YES, and the entire room watching the screen sees the bar move within 400ms. Monad’s speed becomes tangible. The UI should target 60fps animations that begin optimistically on transaction submission and confirm on block inclusion.

Key technical requirement: subscribe to on-chain events (or use an indexer with sub-second latency) so that every connected client sees every trade reflected live. Polling is not acceptable – use WebSocket subscriptions to an RPC node or a lightweight indexer.

15.4 Social Proof and Liveness

An empty-feeling app kills momentum at a hackathon demo. The UI must radiate activity:

- **Live trade feed**: A scrolling ticker of recent trades (“0xab…cd bought 0.5 MON of YES on Proposal A – 2s ago”) visible on every screen
- **Participant count**: “47 traders active” displayed prominently
- **Activity heatmap**: Proposals with recent trading volume glow warmer; stale proposals fade
- **Block heartbeat**: A subtle pulse indicator synced to Monad’s ~400ms block time, showing the chain is alive

The goal: when 30 people at a hackathon demo are all trading simultaneously, the screen should feel like a living organism – bars moving, trades flowing, numbers updating. The room’s collective intelligence becomes visible.

15.5 Visual Design Direction

- **Less DeFi dashboard, more live poll.** Think election night coverage, not Uniswap.
- **High contrast.** YES = green, NO = red, with enough contrast for projector visibility at demo booths.
- **Large type.** Proposal titles and percentages should be readable from 3 meters away on a laptop screen.

- **Minimal chrome.** No sidebar navigation, no complex headers. The proposals ARE the interface.
- **Dark mode default.** Projectors and demo environments favor dark backgrounds with bright data.

Logo: Compass Rose The Meridian logo is a compass rose – a navigational instrument that finds true north. The metaphor: Meridian finds truth through market consensus, navigating governance decisions toward the correct bearing.

SVG specification:

```
<svg width="26" height="26" viewBox="0 0 26 26" fill="none">
  <!-- Outer ring -->
  <circle cx="13" cy="13" r="12" stroke="rgba(212,168,83,0.4)" strokeWidth="1" />
  <!-- Inner ring -->
  <circle cx="13" cy="13" r="8" stroke="rgba(212,168,83,0.2)" strokeWidth="0.5" />
  <!-- Crosshair axes -->
  <path d="M13 1 L13 25" stroke="rgba(212,168,83,0.15)" strokeWidth="0.5" />
  <path d="M1 13 L25 13" stroke="rgba(212,168,83,0.15)" strokeWidth="0.5" />
  <!-- Vertical needle (primary, high opacity) -->
  <path d="M13 5 L15 13 L13 21 L11 13 Z" fill="#D4A853" fillOpacity="0.8" />
  <!-- Horizontal needle (secondary, low opacity) -->
  <path d="M5 13 L13 11 L21 13 L13 15 Z" fill="#D4A853" fillOpacity="0.3" />
  <!-- Center point -->
  <circle cx="13" cy="13" r="1.5" fill="#D4A853" />
</svg>
```

Design anatomy: - **Two concentric circles**: Outer (40% opacity) and inner (20% opacity) rings represent the two resolution layers – the market ring (which proposal wins) and the oracle ring (did it deliver) - **Crosshair axes**: Faint cardinal lines at 15% opacity. The grid of possible governance outcomes. - **Vertical diamond needle**: The dominant north-south axis at 80% opacity. This is the crowd's conviction – the primary signal pointing toward the best proposal. - **Horizontal diamond needle**: The secondary east-west axis at 30% opacity. The weaker cross-signal, representing the information that has not yet converged. - **Center point**: Solid gold dot. The point of collapse where all information converges into a single decision.

Brand color: #D4A853 (Meridian Gold). Used at varying opacities to create depth without additional hues. The gold evokes both value (capital at stake) and instruments of navigation (brass compasses, astrolabes).

Usage rules: - Always render on dark backgrounds (#0A0A0F or darker) - Minimum size: 24x24px. At smaller sizes, drop the inner ring and crosshairs. - The logo should animate subtly on page load: the needle rotates from a random bearing to true north (pointing up), symbolizing the market converging on truth.

15.6 The Multiverse Metaphor

Quantum superposition is a technical term. **Parallel universes** is a story people immediately understand. The UI language should lean hard into the multiverse framing, inspired by Lightcone's approach to impact markets.

Each proposal is a universe. When a user deposits, their capital enters every universe simultaneously. The UI should make this tangible:

- **Deposit screen**: "Your 1 MON now exists in 3 parallel universes." Show the deposit splitting into N glowing threads, one per proposal.
- **Proposal cards**: Each card is a window into a universe. The welfare price is the crowd's verdict on *what happens in this universe* – what the outcome looks like if this proposal becomes reality.
- **Settlement animation**: When the Decision collapses, show the winning universe solidifying while losing universes fade and dissolve. Capital flows back from dissolved universes into the user's wallet. This is the "wave function collapse" made visual.

The language throughout the app should reinforce the metaphor: - Not "Market A, Market B" but "Universe: Hire Alice" / "Universe: Hire Bob" - Not "Your position" but "Your position in this universe" - Not "Settlement" but "Reality collapsed" - Not "Revert" but "Universe dissolved – capital returned"

This framing does three things: (1) makes the capital superposition mechanic instantly intuitive, (2) differentiates Meridian from every other DeFi product on sight, and (3) gives hackathon judges and spectators a sticky mental model they will remember.

15.7 Surfacing the Information Product

Meridian is not just a governance tool. It is an **information machine** that produces market-priced intelligence about the impact of governance actions. The UI must surface this information product explicitly – it is the thing that makes people stop and stare.

The delta display. For every Decision with multiple proposals, show the spread between the leading and trailing welfare prices. This delta IS the product:

```
Decision: "Who should lead Protocol Development?"

  Hire Alice                72%
  Hire Bob                  48%
  Status Quo                35%

  Market Signal: Alice leads Bob by 24 points.
  The crowd is mass-pricing Alice as the highest-impact hire.
```

Historical impact chart. Show how the crowd's assessment of each proposal's impact has evolved over time. This is the secondary chart view (not the hero, per 15.1) – but it tells a powerful story: *the collective intelligence of the crowd, repriced every 400ms, converging on a governance answer.*

Shareable intelligence. Every Decision should produce a shareable card (image or link preview) that shows the current crowd verdict. Example: "Meridian says: the crowd prices Hire Alice at 72% welfare impact, 24 points ahead of Hire Bob. 142 traders, 850 MON in play." This is Lightcone's insight applied to governance – the information itself becomes the viral asset, not the trading interface.

"What the crowd knows" banner. At the top of the app, a rotating ticker of active Decisions and their current crowd signals. This positions Meridian as a live governance intelligence feed, not an app you open when you want to trade. People should bookmark it to *watch*, even if they never trade.

---

16. References

1. Hanson, R. (2000/2013). "Shall We Vote on Values, But Bet on Beliefs?" George Mason University.
2. Yukseloglu, A. & Larbi, S. (2025). "Quantum Markets: A Capital Efficient Mechanism for Scaling Futarchy." Paradigm Research. https://www.paradigm.xyz/2025/06/quantum-markets
3. Paradigm reference implementation. https://github.com/Sofianel5/quantum-markets
4. MetaDAO Program Architecture. https://docs.metadao.fi/implementation/program-architecture
5. MetaDAO Price Oracle. https://docs.metadao.fi/implementation/price-oracle
6. MetaDAO CLOB-to-AMM Proposal. https://hackmd.io/@joebuild/proposal-clob-to-amm
7. Monad Developer Documentation. https://docs.monad.xyz/developer-essentials/differences
8. Monad Opcode Pricing. https://docs.monad.xyz/developer-essentials/opcode-pricing
9. Monad Base Fee Design. https://www.category.xyz/blogs/redesigning-a-base-fee-for-monad
10. Decker, N. "Futarchy, Distortions from Hedging, and Other Thoughts." https://nicholasdecker.substack.com/distortions-from-hedging
11. Frontiers (2025). "Futarchy in Decentralized Science: Empirical and Simulation Evidence." https://www.frontiersin.org/journals/blockchain/articles/10.3389/fbloc.2025.1650188/full
12. Maloney & Mulherin (2003). "The Complexity of Price Discovery in an Efficient Market." Journal of Corporate Finance.
13. White, D. (2024). "Multiverse Finance." Paradigm Research. Conceptual framework for conditional asset markets across parallel outcome universes.
14. Lightcone Protocol (Aradtski & Defiance, 2025). Impact markets: trading the impact of events on real assets, decoupling probability from impact. Inspiration for Meridian's governance intelligence framing.

Appendix A: Full CPMM Formula Reference

A.1 Core Invariant

```
yesReserve * noReserve = k
```

A.2 Implied Welfare (YES Price)

```
welfare = noReserve / (yesReserve + noReserve)
```

Range: (0, 1), stored as basis points [0, 10000].

A.3 Buy YES with `d` vMON

```
YES_out = d * (yesReserve + noReserve + d) / (noReserve + d)
```

Post-trade reserves:

```
yesReserve_new = k / (noReserve + d)
noReserve_new = noReserve + d
```

Note: k is preserved (before fees). With fees, k_new > k.

A.4 Buy NO with `d` vMON

```
NO_out = d * (yesReserve + noReserve + d) / (yesReserve + d)
```

A.5 Sell YES: Receive `a` vMON for `q` YES tokens

```
a = [(yesReserve + noReserve + q) - sqrt((yesReserve + noReserve + q)^2 - 4 * noRe
```

A.6 Price Impact (Buy YES, size `d`)

```
impact = d * yesReserve / [(yesReserve + noReserve) * (yesReserve + noReserve + d)]
```

A.7 Fee-Adjusted Buy YES

```
d_eff = d * (10000 - feeBps) / 10000
YES_out = d_eff * (yesReserve + noReserve + d_eff) / (noReserve + d_eff)
```

Remaining d − d_eff is retained in the pool (increases k).

A.8 LP Divergence Loss (Virtual Liquidity)

For a virtual LP position initialized at (L, L) with k = L^2:

If the market moves to welfare p (from initial 0.5):

```
LP_value = 2 * L * sqrt(p * (1 - p))
```

At p = 0.5: LP_value = 2L ∗ 0.5 = L (no loss). At p = 1.0: LP_value = 0 (maximum loss = L).

Appendix B: Comparison with Paradigm Reference Implementation

Aspect	Paradigm Ref. Impl.	Meridian Spec
AMM	Uniswap V4 concentrated liquidity	Custom CPMM (simpler, cheaper)
TWAP window	30 seconds	25,000 blocks (~2.8 hours)
Welfare tracking	All-time max YES price	Time-weighted average over window
Resolution	External ECDSA signer	Self-referential (MVP) or oracle (prod)
Token standard	Separate ERC-20 per proposal	Internal accounting (gas optimized)

Aspect	Paradigm Ref. Impl.	Meridian Spec
Solvency	Potential gap in seed accounting	Formal proof with virtual liquidity bound
Manipulation resistance	Minimal (30s window)	Lagging TWAP + min volume + TWAP window
Target chain	Generic EVM	Monad-optimized

Appendix C: Monad-Specific Technical Constraints

Constraint	Value	Impact on Design
Block time	400ms	Oracle updates every block; TWAP parameters in blocks not seconds
Block gas limit	200M	Collapse with 20 proposals fits comfortably
Tx gas limit	30M	No single operation exceeds this
Cold SLOAD	8,100 gas	First storage access per slot is expensive; minimize cold reads
Cold SSTORE	8,100+ gas	Pack related data into fewer slots
Warm access	100 gas	Subsequent reads/writes to same slot are cheap
Gas charging	Gas limit, not usage	Frontend must estimate tightly
Max contract size	128KB	Single contract can hold all logic
Finality	~800ms	Near-instant settlement confirmation
No global mempool	N/A	Reduced frontrunning exposure
No blob txs	N/A	All data on-chain or via events
No historical state	N/A	Need indexer for historical market data