Market-Based Oracle Resolution and Evolutionary Agent Economy via Smart Contracts

Batayan E. Sheep

October 7, 2025

Abstract

We present an AI-oriented smart contract architecture that resolves the oracle problem via market-based scoring and enables a survival-of-the-fittest dynamic among autonomous AI agents. Our framework formalizes reward allocation as an inverse-error rule with desirable properties, couples it with replicator dynamics for evolutionary selection, and demonstrates feasibility with an Ethereum (L2-first) implementation using pluggable verifiers (Pyth, UMA, Chainlink). We further extend the design with ZKML proofs, an off-chain Agent Farm for strategy mutation/selection, and cross-chain reputation. Simulations highlight capital concentration, persistent top-performers, and rapid under high-frequency task cycles. We discuss applications in decentralized forecasting, LLM evaluation, and AI labor markets.

1 Introduction

Smart contracts excel at on-chain verifiability but are constrained by the *oracle problem*—blockchains cannot natively access or validate external data [6]. This limits adoption in AI-centric tasks where outputs are informative but not deterministically checkable on-chain. We propose a market-based oracle resolution coupled with evolutionary incentives: agents stake, commit encrypted predictions, reveal, and are paid in proportion to the inverse of their (relative) error against a resolved truth. Non-reveals or fraud are penalized via slashing. Over repeated tasks, selection pressure concentrates capital and opportunity in the most accurate strategies, yielding a decentralized AI agent economy [1].

2 Mathematical Framework

2.1 Oracle Resolution by Market-Based Scoring

Let $p_i \in \mathbb{R}$ be agent i's prediction and p the resolved truth (e.g., a BTC/USD close retrieved by an oracle such as Pyth [2]). Define relative error

$$\varepsilon_i = \left| \frac{p_i - p}{p} \right|. \tag{1}$$

Let R be the reward pool and $\epsilon > 0$ a smoothing constant (units aligned with the scaling of p). We allocate rewards by

$$r_i = R \cdot \frac{1}{\varepsilon_i + \epsilon} \Bigg/ \sum_j \frac{1}{\varepsilon_j + \epsilon}. \tag{2}$$

Properties. (i) **Monotonicity:** If $\varepsilon_i < \varepsilon_k$ then $r_i > r_k$. (ii) **Continuity:** r_i is continuous in each p_j. (iii) **Boundedness:** $0 \le r_i \le R$, $\sum_i r_i = R$. (iv) **Noise robustness:** ϵ regularizes near-ties and jitter; larger ϵ smooths payouts, smaller ϵ sharpens selection. Variants such as softmax $r_i \propto \exp(-\beta\varepsilon_i)$ allow tuning of selection pressure β.

2.2 Evolutionary Dynamics

Let $x_i(t)$ denote agent i's capital (or population share) and f_i its expected payoff (fitness). The population evolves under replicator dynamics [4]:

$$\dot{x}_i \;=\; x_i\big(f_i - \bar{f}\big), \qquad \bar{f} = \sum_j \frac{x_j}{\sum_k x_k} f_j. \tag{3}$$

Under repeated tasks with (2), higher-accuracy strategies yield $f_i > \bar{f}$, expanding their share. Slashing (for non-reveal or mismatch) acts as negative shocks that accelerate of unreliable agents.

3 Simulation Results

We simulate N=100 agents over T=50 generations. Each agent has a latent accuracy $\alpha_i \sim \text{Beta}(5, 2)$; predictions add Gaussian noise scaled by $(1 - \alpha_i)$. Rewards follow (2). We observe: (i) capital concentration into top-5% (Pareto-like), (ii) persistent top performers despite stochasticity, (iii) mean capital grows sub-exponentially while the maximum grows superlinearly.[1]

4 Applications in the AI Agent Economy

Decentralized forecasting. Agents forecast prices, weather, macro indicators; truth is resolved by Pyth/UMA; incentives favor calibration and honesty [2, 5]. **LLM evaluation.** Outputs (summaries, translations) are peer-evaluated and scored by a market mechanism rather than a centralized curator. **Autonomous labor markets.** Agents stake, execute micro-tasks, accumulate reputation/capital, and graduate into higher-stake tasks; poor performers exit. These align with early visions of programmable machine economies [1].

5 Conclusion

Our architecture resolves external truth by economic means and implements evolutionary selection among AI agents. The design is implementable on Ethereum L2s with pluggable verifiers and practical safeguards. Extensions to ZKML, cross-chain reputation, and multi-domain task markets point to a trustless, scalable, AI-native economic substrate [3].

6 Ethereum Implementation Details

6.1 Networks, Tokens, Time

L2-first. Use Arbitrum/Optimism/Base for low gas and high task cadence. **Reward token.** Prefer ERC-20 stablecoins for payouts; stake can be ETH or ERC-20. **Time.** Use `block.timestamp` (UTC); add guard bands (10–30s) to commit/reveal windows.

6.2 Contract Topology and Storage

Core contracts. `AgentRegistry` (stake/unstake/slash), `TaskMarket` (post → commit → reveal → settle), `Verifier` (Pyth/UMA/Chainlink adapters), optional `Reputation`. Use dependency injection so `TaskMarket` holds a verifer address upgradable under timelock governance.

[1]Figures can be embedded as: `\includegraphics[width=.8\textwidth]{agent_capital_over_time.png}`, `winner_distribution.png`.

Task state (packed).

```
struct Task {
  address client;
  address rewardToken;   // ERC20 or address(0) for native
  uint64  tCommit;       // unix seconds
  uint64  tReveal;       // unix seconds
  uint96  reward;        // token units
  bytes32 verifierKey;   // e.g., Pyth feedId or UMA queryId
  bool    settled;
}
```

Submissions. Sparse mapping: `commitHash = keccak256(abi.encode(value, salt))`; on reveal provide `(value, salt)`. Index with clean events for analytics.

6.3 Commit–Reveal and Collusion Mitigations

Commit obfuscates values during the commit window; reveal checks hash equality. Keep commit windows short; optionally obscure participant set size until reveal; penalize suspiciously identical reveals (correlation penalties) in post-processing.

6.4 Oracle Adapters

Pyth (pull). A relayer pushes price updates near settlement; contract reads `getPriceUnsafe(feedId)` and normalizes to signed 1e8.

UMA (optimistic). For arbitrary truths (VWAP, median-of-venues, task-specific rubric), a proposer submits a value; undisputed proposals finalize; disputes escalate to voters. Cache final value on-chain and expose via `resolve(bytes)` [5].

Chainlink. Read-only feeds for vetted assets; simpler ops, less flexible.

6.5 Settlement and On-Chain Math

Weights. $w_i = 1/(\varepsilon_i + \epsilon)$ using fixed-point. Avoid heavy transcendental functions. **Payout.** Pro-rata: $r_i = R \cdot w_i / \sum_j w_j$. For high N, prefer *pull* claims (`claim(taskId)`) over *push* loops. Top-1 mode is a gas-light variant that sharpens selection.

6.6 Security and Correctness

Access control. Ownable/AccessControl; verifier swaps behind a TimelockController. **Pausable.** Emergency stop. **Reentrancy.** Guard settlements/withdrawals. **Oracle freshness.** Enforce `publishTime` max-age. **Slashing.** Rule-based and non-discretionary; send to a fee vault or pool. **Rounding.** Fixed 1e8 price scale; reason about bounds before `unchecked`.

6.7 Gas Optimizations

Struct packing; `immutable` addresses; `calldata` for external params; custom errors; batch `verifyAndSettleMany(ids)`; prefer L2 rollups for frequency.

6.8 Automation, Indexing, and Ops

Keepers. Gelato/Chainlink Automation for timed `updatePriceFeeds` and settlements. **Indexing.** The Graph or custom indexer to compute reputation, win-rates, error histograms. **Dashboards.** REST/GraphQL for leaderboards and cohort analysis.

6.9 ZKML Integration

Goal. Prove that a revealed value p_i was produced by a specific model M on input x without revealing M or x (privacy-preserving auditability).

Workflow. (1) Off-chain: agent computes (p_i, π_i) where π_i is a zkSNARK proof for statement "$p_i = M(x)$ under commitment C_M". (2) On-chain: submit p_i with `proof` to `TaskMarket.reveal()`; the `ZkVerifier` adapter verifies π_i and marks the submission "attested". (3) Settlement may weight attested submissions higher or gate large tasks for attested agents only.

Interface (sketch).

```
interface IZkVerifier {
  function verify(bytes calldata proof, bytes32 modelCommitment,
                  int256 output) external view returns (bool);
}
```

Use EZKL/Giza-style compilers off-chain to produce π_i; on-chain verification keys are stored in `ZkVerifier`. Gas can be reduced via aggregator contracts or proof recursion.

6.10 Agent Farm (Strategy Evolution)

Manager. Off-chain service that (i) spawns agent variants (hyperparameters, prompts, models), (ii) enrolls them (stake, commit, reveal), (iii) logs payoffs/errors, (iv) applies selection & mutation.

Cycle. Each epoch: compute EMA score $\rho_i \leftarrow \alpha s_i + (1-\alpha)\rho_i$ where $s_i = 1/(\varepsilon_i + \epsilon)$. Keep top-$q$%; clone+mutate; retire bottom agents (withdraw or accept slashing).

Risk control. Cap per-agent concurrent tasks; throttle correlation (avoid all agents converging to one guess); portfolio-style budget allocator maximizes expected payoff under variance constraints.

6.11 Multi-Task and Cross-Chain Extensions

Task types. Add `taskType` and route to per-type verifiers (price, classification, LLM QA with optimistic evaluation). **Cross-chain.** Mirror markets on multiple L2s; synchronize reputation via CCIP/LayerZero; preserve identity via ERC-6551 or attestations.

6.12 Testing and Formal Checks

Foundry/Hardhat. Fuzz commit/reveal boundaries; invariants (payout conservation $\sum r_i \leq R$, no settlement pre-deadline, slash cannot underflow stake). **Scribble/Certora.** Specify properties (hash binding, oracle-age guard). **Fork tests.** Verify Pyth/UMA addresses and freshness enforcement on live forks.

Acknowledgments

We thank the open-source oracle and rollup communities whose tooling and documentation enable practical deployments.

References

[1] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum White Paper*, 2014. https://ethereum.org/en/whitepaper/.

[2] Yu Chen et al. Pyth network: Bringing real-world data on-chain, 2021. https://pyth.network/.

[3] Nicola Gennaioli, Yueran Ma, and Andrei Shleifer. Expectations and investment. *NBER Macroeconomics Annual*, 29(1):379–431, 2014.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1992.

[5] UMA Protocol. Uma oracle design and mechanism, 2023. https://docs.umaproject.org.

[6] Rui Zhang, Rui Xue, and Ling Liu. A survey on oracle design in blockchain systems. In *Proceedings of the VLDB Endowment*, volume 13, pages 2350–2363, 2020.