Settlemint: a self-settling event market for NEAR

Dipankar Sarkar
doom2quake

August 2026

Abstract

An on-chain event market that resolves a real-world question has a settlement problem that predates crypto: someone decides the outcome, someone pays every winner, and everyone else must be able to check that the payout matched the outcome and that nothing was stranded. Settlemint is a NEAR-native contract that makes settlement mechanical and provably complete. A resolver opens a binary market, stakers commit yoctoNEAR while it is open, and when the outcome is known the resolver records it once and every winner is paid pro rata in a single resumable pass. The last winning claimant sweeps the rounding remainder, so escrow lands at exactly zero and no stake is stranded, and a permissionless void escape hatch lets every staker recover their own stake if the resolver never resolves. This paper describes the settlement state machine, its conservation argument, and the milestone-1 implementation: a NEAR Rust contract tested to 23 unit checks and compiled to a 205 KB testnet WASM, and a Python mirror of the same arithmetic held in cross-language parity by a further 32 tests. All work is NEAR testnet only; no mainnet, no users, no audit.

1 Introduction

When a prediction market, an event contract, or an agent-run payout settles, a participant who lost typically has no way to tell an honest resolution from a rigged one, and no guarantee that every winner was paid to the last unit. Two distinct failures hide inside "the market settled." First, a *liveness* failure: if the resolver key goes silent, escrow can sit locked with no escape. Second, an *accounting* failure: pari-mutuel payout is a division, and naive floor division drops a remainder, so the sum of payouts silently falls short of the sum of stakes and some winner is quietly short-changed.

Settlemint targets both failures directly on NEAR. It is deliberately narrow: it does not attempt to decide subjective outcomes (that is a later milestone), only to make the *settlement* mechanical, complete, and checkable once an outcome is recorded. This paper covers the milestone-1 core: the on-chain state machine, its conservation property, and the tested implementation.

2 Why NEAR

Two NEAR features make this design fit the chain rather than merely run on it.

Scoped access keys. NEAR accounts separate full-access keys from named function-call access keys [2]. An operator agent that settles markets can hold a key restricted to the `resolve` and `settle` methods of the settlement contract, so a compromised agent can advance markets but cannot move funds out of an account. The blast radius of a lost key is exactly the two methods, not the whole balance.

Predictable low fees and yocto accounting. NEAR settles in yoctoNEAR (10^{-24} NEAR) with predictable gas [5], so the one-resolve-plus-one-settle-pass pattern is cheap enough to run at cadence, and the dust that a pari-mutuel split produces is a genuine integer remainder the contract must account for rather than a floating-point artifact. The contract is written in Rust against `near-sdk` [4] and compiled to WASM; it is a reimplementation of the state machine for the NEAR runtime, not a transliteration of EVM bytecode.

NEAR Protocol Rewards [3] scores measured on-chain activity (contract calls, unique wallets) directly, so an agent that settles many small testnet markets produces exactly the honest, measured activity the programme is built to reward.

3 The settlement state machine

An event is a record with a named resolver, a close timestamp, a phase, an outcome, the two side pools, the escrow still held, and the winning-side stake not yet claimed. It moves through four phases.

```
Open create_event names a resolver and closes_at; take_position adds yocto
Resolved resolver records the outcome, only at/after close; winning pool frozen
Settled every winner paid pro rata; reached only when escrow == 0
Voided resolver silent past close + 7 days; each staker withdraws own stake
```

Resolver-only, post-close resolution. Only the named resolver may call `resolve`, and only at or after `closes_at`. Resolving early would let the resolver cut off an open market mid-flight, so the contract forbids it.

Pro-rata payout with a dust sweep. On a resolved market, an account that staked m on the winning side, where the winning pool is W and the total pool is T, is owed

$$\text{payout} = \left\lfloor \frac{T \cdot m}{W} \right\rfloor,$$

except that the *last* unclaimed winner receives the entire remaining escrow instead of its floor share. Because escrow is decremented by every payout and the final winner takes whatever is left, the rounding remainder is paid out rather than stranded.

Refund cases. If the winning side had no stake, or the market was voided, each account is refunded exactly its own total stake. Voiding is permissionless after `closes_at + 7 days`, which is why a lost or silent resolver key cannot lock funds.

4 Conservation

Let S be the sum of all stakes on an event. The contract maintains an `escrow` field equal to the yoctoNEAR still held, initialised to S as positions are taken and decremented by each payout. Two invariants hold.

1. **No overpayment.** Each account is marked `claimed` before its transfer is scheduled, so no account is paid twice, and each floor share is at most its exact proportional share, so the running total of floor shares never exceeds S.

2. **No underpayment.** The last unclaimed winner is paid the entire remaining escrow, which is S minus the sum of the earlier floor shares. Hence the total paid equals S exactly, and `escrow` reaches 0.

A market is reported `Settled` only once `escrow` reaches zero, so a partly-paid market is never mistaken for a final one while a winner is still owed money. The implementation asserts this with a test that settles one of two winners and checks the phase is still `Resolved` with non-zero escrow.

5 Implementation and evidence

The contract. `contract/src/lib.rs` is the whole mechanism in one file, depending only on `near-sdk`. It compiles to a 205 KB release WASM with `cargo build --release --target wasm32-unknown-unk` which is the artifact a testnet deploy uploads.

On-chain tests. `cargo test` runs **23** unit tests on the NEAR `unit-testing` harness, covering every phase transition, the access-control checks, the pro-rata split, the dust sweep, the no-winner and void refunds, and the not-finalised-until-escrow-zero property.

Off-chain mirror and cross-language parity. `settlemint/model.py` reimplements the identical arithmetic in Python. `tests/test_parity.py` asserts the exact payout literals the Rust tests assert, so a one-line drift in either implementation fails a test. The Python suites total **32** passing tests, including a run-fixture test that checks the recorded run in `docs/run.json` conserves value to the last yocto and reproduces exactly, and an adapter test that proves the live NEAR client refuses any mainnet endpoint.

Suite	Command	Result
NEAR Rust core	`cargo test`	23 passing
Python model, parity, fixtures	`pytest tests -q`	32 passing
Deployable artifact	`cargo build --release`	205 KB WASM

Table 1: Milestone-1 evidence, run 2026-08-26, all offline and keyless.

6 Limitations

All work is NEAR testnet only; the live client rejects mainnet endpoints in code. The recorded run is an offline fixture that executes the settlement arithmetic but touches no network, so no public block-explorer receipt exists yet; this repo ships no funded key. The verifiable-resolution hash commitment, the operator agent on a scoped access key, and the forkable template are later milestones and are not in this release. There are no users, no revenue, no partnerships, and no third-party audit; the 55 passing tests are the authors' own. These limits are restated in `docs/LIMITATIONS.md`.

7 Related work

Pari-mutuel and market-scoring-rule mechanisms have a long history [1], and decentralized prediction markets such as Augur [6] put resolution and payout on-chain. Settlemint is narrower and

more mechanical: it does not decide subjective outcomes in this milestone, it makes the *settlement* step complete and conservative on NEAR, with a liveness escape hatch and an integer dust sweep as first-class properties.

References

[1] Robin Hanson. Combinatorial information market design. *Information Systems Frontiers*, 5(1):107–119, 2003.

[2] NEAR Foundation. Near documentation: Access keys. https://docs.near.org/concepts/protocol/access-keys, 2026. Full-access and function-call access keys.

[3] NEAR Foundation. Near protocol rewards. https://nearprotocolrewards.com, 2026. Metric-based, non-dilutive rewards for measured GitHub and on-chain activity.

[4] NEAR Foundation. near-sdk-rs: Rust sdk for near smart contracts. https://github.com/near/near-sdk-rs, 2026. Version 5.6.0.

[5] NEAR Foundation. Nomicon: the near protocol specification. https://nomicon.io, 2026. Runtime, gas, and yoctoNEAR accounting.

[6] Jack Peterson, Joseph Krug, Micah Zoltu, Austin K. Williams, and Stephanie Alexander. Augur: a decentralized oracle and prediction market platform. https://arxiv.org/abs/1501.01042, 2019. arXiv:1501.01042.