

FairFun

Liquid Sponsorship: Continuous Yield Distribution for Fully-Liquid Tokens on Solana

Version 3.0 — Current Implementation

Program ID: HsydRBzU6Bcw6ku3h4K6JqimRTxTeCfvZQL6yDBvAi4A

Network: Solana Mainnet — Anchor Framework

ABSTRACT

FairFun introduces Liquid Sponsorship — a continuous, proportional revenue-distribution layer for tokens on Solana. It solves a problem that existing approaches (MasterChef staking, liquid staking tokens, and snapshot airdrops) each fail to solve simultaneously: distributing yield to holders without requiring them to lock, stake, or time their position. Distribution weight is determined by Gravity, computed at the moment each treasury deposit arrives — no continuous off-chain engine, no USD price oracle, and no sampling error. Any existing PumpFun or SPL token can register with FairFun in a single transaction. New yield-bearing memecoins can launch directly on FairFun using Meteora DBC with a flat price curve, replacing speculative price games with a transparent community economy. The \$FAIRFUN ecosystem token adds a cross-project raffle: its trading fees purchase participating project tokens and gift them to randomly selected \$FAIRFUN holders.

Contents

- I The Retail Problem: Earning by Holding**
 - I.A** Why Timing-Only Models Fail
 - I.B** The Memecoin Revenue Opportunity
 - I.C** The Ideal: Three Properties That Must Coexist
 - II Approach 1: Global Index Distribution (Staking Required)**
-

II.A	MasterChef on Ethereum
II.B	Why Solana's Parallel Account Model Changes Everything
II.C	Token-2022 Transfer Hooks
II.D	Permissionless Program Instructions
III	Approach 2: Liquid Staking Tokens
III.A	The Exchange Rate Model
III.B	Trade-offs and Fragmentation
IV	Approach 3: Gravity (No Staking, Full Liquidity)
IV.A	The Continuous Integral: Theoretical Ideal
IV.B	The Riemann Approximation and Its Practical Limits
IV.C	Deposit-Event Gravity: The Correct Architecture
IV.D	Snapshot vs. Gravity: Scenario Analysis
IV.E	The Trust Trade-off
V	Approach Comparison
VI	The \$FAIRFUN Ecosystem Token and Gift Raffle
VI.A	Role of the Ecosystem Token
VI.B	The Cross-Project Gift Raffle Mechanic
VII	FairFun Revenue Architecture
VII.A	Treasury, Deposits, and JitoSQL Productivity
VII.B	Protocol Fee and Claim Mechanics
VIII	Registering Tokens and Launching Yield-Bearing Memecoins
VIII.A	Registering an Existing Token
VIII.B	Creating a Yield-Bearing Memecoin with Meteora DBC
VIII.C	The Community Economy Model
IX	Flexible Claim Modes
X	System Architecture
XI	On-Chain Security Model
XII	Extended Mechanics: Raffles and Collections
XII.A	Raffle Systems
XII.B	Multi-Coin Collections
XIII	Pre-Pump Pools, Coordinated Launch, and the Liquid Staking Stack

- XIII.A The Pre-Pump Coordination Problem
 - XIII.B The LaunchPool Contract
 - XIII.C Program Sniper: Coordinated Buy at Launch
 - XIII.D Threshold-Locked Token Vault
 - XIII.E Primitive 2 — Liquid Staking Tokens (sTokens)
 - XIII.F Primitive 3 — Automatic Gravity Registration
 - XIII.G The Complete Launch Stack and Lifecycle
-

XIV Real-World Applications

- XIV.A The Community Bundled Launch
 - XIV.B \$EEVEE — The Living Evolution Protocol
 - XIV.C The Eight Evolutions and the Gene System
 - XIV.D Birth Triggers and New Token Deployment
 - XIV.E FairFun Primitive Stack in \$EEVEE
-

XV The Complete Claim Flow

I. The Retail Problem: Earning by Holding

Memecoins attract retail participants because they are simple: buy a token and participate in a community. But this model has a structural flaw that limits long-term participation: the only way to profit is to sell at the right moment. Most retail participants cannot do this reliably, and the ones who can do it by extracting value from those who cannot.

I.A — Why Timing-Only Models Fail

Without yield, holding a token is economically equivalent to holding cash in a potentially depreciating currency. The rational strategy is always to minimize hold duration and exit before others. This produces predictable boom-bust cycles. Projects that want long-term supporters have no mechanism to reward them: a holder who believed in the project for a year receives no premium over someone who arrived thirty seconds before a lucky exit.

- Retail has no edge on timing. Whales and bots have better information, faster execution, and larger capital buffers.
- Without yield, holding has a negative expected value after fees. The only positive-EV strategy is to flip quickly.
- Projects cannot distinguish supporters from traders. Both look identical on-chain unless duration is measured.

I.B — The Memecoin Revenue Opportunity

Despite their reputation for speculation, memecoins generate substantial real revenue. Every trade on an AMM produces fees. High-volume tokens generate millions in annual fee revenue. Projects with active communities build products, sell services, and create content that generates recurring income. Under the current model, none of this flows back to the holders who provided the market stability that made it possible.

THE CORE MISALIGNMENT

Traders generate a project's revenue. Long-term holders deserve to benefit from it. A mechanism that routes revenue to conviction rather than activity changes the incentive structure of the entire category.

I.C — The Ideal: Three Properties That Must Coexist

- **No staking required.** Tokens stay in the holder's own wallet, fully tradable at any moment.
- **No timing required.** Rewards accumulate automatically. No snapshot date, no claim window.

— **Duration-weighted.** Long-term holders earn more than short-term ones regardless of entry price.

The rest of this paper surveys existing approaches that achieve some but not all three properties, then explains how FairFun's Gravity mechanism achieves all three simultaneously.

II. Approach 1: Global Index Distribution (Staking Required)

The most widely deployed on-chain yield distribution approach uses a global reward index that advances each time revenue arrives, combined with a per-user debt that records how much of the index the user has already collected.

II.A — MasterChef on Ethereum

MASTERCHEF GLOBAL INDEX

$$\text{accRPS} + = \frac{\text{Revenue}}{\text{totalStaked}}$$

accRewardPerShare — cumulative rewards per unit of staked token, advances on each deposit

totalStaked — sum of all tokens currently locked in the staking contract

Pending_w — $\text{Stake}_w \times \text{accRewardPerShare} - \text{rewardDebt}_w$

rewardDebt_w — $\text{Stake}_w \times \text{accRewardPerShare}$ at time of last user interaction

This is elegant and gas-efficient but has two hard requirements: tokens must be staked in the contract (the protocol must know each user's balance), and totalStaked must be cheaply queryable as a running total.

II.B — Why Solana's Parallel Account Model Changes Everything

Solana processes transactions in parallel across CPU cores. Every account a transaction touches must be declared in its account list before execution. A transaction cannot iterate over an unbounded set of accounts at runtime.

THE PARALLEL ACCOUNT CONSTRAINT

On Ethereum, a contract can loop over all stakers in storage. On Solana, you must know every account you will touch before the transaction is submitted. A MasterChef port that tried to push updates to 10,000 holders in one transaction would fail: you cannot include 10,000 unknown wallet addresses in a single account list. All Solana-native designs must therefore be pull-based: holders fetch their own state lazily by submitting a transaction that touches only their own accounts.

II.C — Token-2022 Transfer Hooks

Token-2022 supports transfer hooks: a custom program invoked on every token transfer. This provides one Solana-compatible path to a global index.

Transfer hook: update index on every token movement

```
fn transfer_hook(ctx: Context, amount: u64) -> Result<()> {  
    sender.pending += sender.staked * (global_index - sender.debt);  
    sender.staked -= amount; sender.debt = global_index;  
    receiver.staked += amount; receiver.debt = global_index; // no past credit  
    Ok(())  
}
```

Advantages: fully on-chain, automatic, no trusted backend for balances. **Limitations:** extra compute on every swap and send; protocols that don't support Token-2022 hooks break composability; an on-chain price oracle with ~5-second latency introduces imprecision; tokens still require a per-wallet PDA making this functionally equivalent to staking.

II.D — Permissionless Program Instructions

Without Token-2022, a permissionless checkpoint instruction that anyone can call achieves a similar result — but tokens must still be tracked in a program-owned vault, making this a staking model with a pull-based update mechanism.

III. Approach 2: Liquid Staking Tokens

Liquid staking tokens (LSTs) improve UX: instead of a separate reward claim, the holder receives a wrapper token whose redemption rate grows over time. No claiming required — just hold the wrapper.

III.A — The Exchange Rate Model

LST EXCHANGE RATE MECHANICS

$$\text{Rate}(t) = \frac{\text{totalTOKEN}(t)}{\text{totalSTOKEN}}$$

totalTOKEN(t) — all TOKEN in vault plus accrued yield at time t

totalSTOKEN — fixed sToken supply (does not change with yield)

Holder value — sTokenBalance x ExchangeRate(t) — grows as yield accumulates

FairFun uses this model internally: deposited SOL converts to JitoSOL inside the treasury PDA, earning staking yield and MEV while waiting to be claimed, then unwinds to native SOL at settlement time.

III.B — Trade-offs and Fragmentation

- **Deposit required.** Holder sends TOKEN to a vault and receives sToken. The original TOKEN is no longer in their wallet.
- **Liquidity fragmentation.** Two markets (TOKEN and sToken) split AMM depth.
- **Reentry penalty.** A holder who exits and re-enters starts at the current exchange rate with no credit for past yield.

THE FUNDAMENTAL LIMIT

LSTs require giving up custody of TOKEN in exchange for sToken. This is a staking model with a liquid receipt, not full liquidity. The holder cannot hold the original TOKEN in their wallet while simultaneously earning yield on it.

IV. Approach 3: Gravity (No Staking, Full Liquidity)

Gravity is a measurement of value-duration. It accumulates for every wallet that holds a token, without any deposit, staking instruction, or vault interaction. Tokens never leave the holder's wallet. The gravity score records the total economic weight the holder has provided to the project over the history of their position.

IV.A — The Continuous Integral: Theoretical Ideal

The ideal gravity score for wallet w at time T is the continuous integral of its USD-denominated token value over the holding duration:

GRAVITY: THE CONTINUOUS IDEAL

$$G_w(T) = \int_0^T \text{Bal}_w(t) \cdot P(t) dt$$

Bal_w(t) — token balance of wallet w at time t

Price(t) — USD price of the token at time t

G_w(T) — cumulative gravity score — total USD-minutes of holding

This integral cannot be computed exactly on a blockchain. Balance and price are piecewise-constant functions that change at discrete on-chain events. Any implementation must approximate it.

IV.B — The Riemann Approximation and Its Practical Limits

The most direct approximation is a left Riemann sum over fixed-width epochs: sample balance and price once per minute and add the product to a running score.

RIEMANN SUM APPROXIMATION ($\Delta T = 60$ S)

$$G_w(T) \approx \sum_i [\text{Bal}_w(t_i) \cdot P(t_i) \cdot \Delta t]$$

t_i — start of epoch i (t₀ = 0, t₁ = 60 s, t₂ = 120 s ...)

Δt — epoch width: 60 seconds

Error bound — O(Δt x max rate of change of Bal x Price) — sub-epoch changes are missed

While mathematically sound, this approach carries three operational requirements that limit its practicality:

- **Persistent off-chain engine.** A process must run every 60 seconds, reading every holder's balance and the current price, 24 hours a day. Downtime means missed epochs and incorrect gravity.
- **USD price oracle dependency.** Multiplying by USD price requires a price feed (Pyth, Switchboard). The oracle introduces latency, potential manipulation, and a trust dependency on an external data source.
- **Computational cost at scale.** For a token with 100,000 holders, each epoch requires 100,000 balance reads. At 60-second intervals this is 86.4 million reads per day — significant infrastructure cost.

IV.C — Deposit-Event Gravity: The Correct Architecture

A simpler and more economically coherent architecture emerges from a direct observation: what matters for revenue distribution is not whether a holder was present at an arbitrary timestamp, but whether they were present at the specific moments when value flowed into the treasury. Gravity should measure presence at revenue events, not presence during arbitrary time intervals.

Under deposit-event gravity, the score advances only when a deposit arrives. At that moment, every holder's balance fraction is recorded and added to their cumulative score:

DEPOSIT-EVENT GRAVITY (FAIRFUN'S MODEL)

$$G_w = \sum_d \frac{\text{Bal}_w(t_d)}{\text{Supply}(t_d)}$$

Bal_w(t_d) — holder's token balance at the exact moment deposit d arrives

Supply(t_d) — total token supply at that moment

Bal_w / Supply — balance fraction: a pure ratio in [0, 1], no price oracle needed

G_w — cumulative sum of balance fractions across every deposit event

Why This Architecture Is Superior

- **No price oracle required.** The balance fraction Bal/Supply is a pure ratio computed entirely from on-chain token account data. No Pyth, no Switchboard, no external dependency.
- **No persistent off-chain engine.** Gravity only advances at deposit events, which are discrete on-chain transactions. The computation runs once per deposit, not once per minute.
- **Directly tied to revenue.** A holder's gravity grows when and only when the project earns money. If no revenue arrives for a week, gravity scores do not change — which is economically correct: no value flowed in, so no weight should accrue.
- **On-chain verifiability.** Deposits are on-chain events. The balance at deposit time is queryable from the token's on-chain state. This means gravity is computable from public on-chain data without any trusted engine.
- **Anti-gaming.** With epoch-based gravity, a sophisticated actor could try to hold through predicted high-fee periods and exit during low-fee periods. With deposit-event gravity, the deposit timing is determined by external trading activity — unpredictable and ungameable. The optimal strategy is simply to hold continuously.

THE GRAVITY INSIGHT

Time-based gravity asks: 'how long have you been here?' Deposit-event gravity asks: 'how many of the project's revenue moments were you present for?' The second question is the economically correct one. A holder who was present for 100 deposit events provided 100 moments of market stability at the exact moments the project needed it. Their reward should reflect this, regardless of the calendar time that elapsed.

The Compounding Loyalty Effect

Because deposit-event gravity is cumulative, early holders who witness many deposit events naturally accumulate a higher score than late entrants with the same current balance. This creates durable loyalty incentives without lockups:

- Holder A joins at launch and witnesses 500 deposit events. Their gravity = $500 \times$ (their average balance fraction).
- Holder B joins later and witnesses only 50 deposit events. Identical current balance, but one-tenth the gravity.
- Holder A receives approximately 10x Holder B's share of the next deposit, despite holding the same number of tokens.

IV.D — Snapshot vs. Gravity: Scenario Analysis

Scenario	Snapshot result	Gravity result	Key insight
Equal balance, early vs. late (180 days vs. 1 day)	50% / 50%	99.4% / 0.6%	180 days = 180x more deposit events witnessed
9x larger balance, 1/90th the time (whale vs. loyalist)	90% / 10% by balance	16.7% / 83.3%	Long-term small holder beats late whale
Flash position (Solana slot, ~400 ms)	Full proportional share	< 0.01% of normal session	One deposit event vs. thousands accumulated
Known distribution date: buy before, sell after	Full reward, zero hold time	Share of only that one event	No timing advantage — gravity is a history, not a moment

Scenario B is the most instructive. A long-term holder C with 100 tokens present at 90 deposit events, versus whale D who acquires 900 tokens just before one deposit. By balance, D holds 90% of supply. By gravity: C has witnessed 90 events with her balance fraction; D has witnessed exactly one. Gravity awards C 83.3% of that deposit despite D holding nine times

more tokens.

IV.E — The Trust Trade-off

Gravity's defining advantage — tokens never leave the wallet — comes with one constraint: the off-chain engine reads wallet balances at deposit time and attests to gravity scores via a backend signature. Seven on-chain security guards ensure that even a compromised backend cannot pay out more than was deposited, decrease a holder's earnings, or leave the vault insolvent. As ZK coprocessors on Solana mature, this trust assumption can be eliminated by proving the gravity computation cryptographically, without changing the settlement contract.

V. Approach Comparison

Approach	Staking req.	Engine	Oracle	Liquidity	Best for
MasterChef (EVM)	Yes	On-chain iterate	No	Locked	EVM fixed-stake pools
Global index + Token-2022 hooks	Yes	Hook (per transfer)	On-chain	Locked	Composable Solana tokens
Global index + permissionless ix	Yes	Lazy pull	On-chain	Locked	Permissionless programs
Liquid staking tokens (LSTs)	Yes (deposit)	Off-chain	Off-chain	Semi (sToken)	Simple yield wrapping
Gravity: deposit-event (FairFun)	No	Event-driven	None	Full	Memecoins, DeFi, neobanks

Gravity with deposit-event calculation is the only approach that simultaneously requires no staking, no continuous engine, no price oracle, and preserves full token liquidity. The trade-off is a bounded backend trust assumption, constrained by seven on-chain invariants.

VI. The \$FAIRFUN Ecosystem Token and Gift Raffle

\$FAIRFUN is the native token of the FairFun protocol. It is itself a yield-bearing token registered on its own FairFun treasury and accumulates gravity like any other project token. Beyond standard gravity distribution, \$FAIRFUN has a unique cross-ecosystem mechanic that connects it to every project in the FairFun registry.

VI.A — Role of the Ecosystem Token

When new projects register with FairFun, they join a shared ecosystem. Each registered project benefits from FairFun's distribution infrastructure, but also contributes to the network effect of the whole. The \$FAIRFUN token is designed to capture a share of that network value and redistribute it to the ecosystem's most committed supporters.

- **Standard gravity yield.** \$FAIRFUN holders earn SOL from the protocol's own trading fee treasury, distributed via gravity like any other FairFun token.
- **Ecosystem exposure.** As more projects register and generate trading fees, the FairFun protocol's revenue grows — benefiting \$FAIRFUN holders whose treasury receives a protocol fee from every registered project.
- **Gift raffle participation.** \$FAIRFUN holders are eligible for a cross-project gift raffle described in section VI.B.

VI.B — The Cross-Project Gift Raffle Mechanic

The gift raffle creates a direct economic link between \$FAIRFUN and every registered project token. The mechanic operates as follows:

- **Fee collection.** A portion of \$FAIRFUN's AMM trading fees accumulates in a dedicated raffle reserve within the treasury.
- **Token purchase.** When the reserve reaches a threshold, the protocol uses those SOL fees to purchase tokens from a participating project. The target project can be rotated periodically, selected by governance, or chosen from a weighted pool that favours projects with the highest recent trading volume.
- **Raffle draw.** A gravity-weighted random selection identifies one \$FAIRFUN holder. Gravity score determines the probability of selection: a holder with twice the gravity has twice the chance of winning.
- **Gift delivery.** The purchased project tokens are sent directly to the winning \$FAIRFUN holder's wallet as an unsolicited gift. No action required.

RAFFLE SELECTION PROBABILITY

$$P(\text{win})_w = \frac{G_w(\$FAIRFUN)}{G_{\text{total}}(\$FAIRFUN)}$$

G_w(\$FAIRFUN) — gravity score of wallet w in the \$FAIRFUN pool

G_total(\$FAIRFUN) — total gravity across all \$FAIRFUN holders

Expected value — same as proportional distribution; variance is the engagement driver

WHY THIS ARCHITECTURE CREATES ECOSYSTEM NETWORK EFFECTS

Every new project that registers with FairFun becomes a potential raffle prize. Holding \$FAIRFUN gives exposure to the entire FairFun ecosystem: the longer you hold, the more raffle draws you are eligible for, and the higher your gravity-weighted probability of winning. Projects benefit because the raffle drives buying pressure on their token (the raffle treasury buys it from the open market). \$FAIRFUN holders benefit because winning gives them tokens in projects they might not have discovered. The ecosystem benefits because value circulates between all participants.

The raffle is not zero-sum. The SOL used to purchase project tokens comes from \$FAIRFUN trading activity, not from any holder's existing balance. Participating projects receive buy pressure. Winners receive tokens. Non-winners continue accumulating gravity for future draws. Every party is better off.

VII. FairFun Revenue Architecture

VII.A — Treasury, Deposits, and JitoSOL Productivity

Every token registered with FairFun has a dedicated on-chain treasury — a Program Derived Address (PDA) at seeds [b"rewards_treasury", token_mint]. This vault is the single destination for all project revenue. Any wallet can deposit SOL via the deposit instruction, which increments `pool.total_deposited` and emits a `TreasuryDeposited` event. Between deposit and distribution, treasury capital is routed into JitoSOL — Solana's leading liquid staking token — earning staking yield plus MEV rewards while waiting to be claimed. At claim time, the relevant portion unwinds to native SOL.

ON-CHAIN TOTAL EVER RECEIVED (COMPUTED AT CLAIM TIME)

$$\text{total}_{\text{onchain}} = \text{treasury.lamports()} + \text{pool.total_claimed}$$

treasury.lamports() — current SOL in the treasury PDA (includes JitoSOL yield)

pool.total_claimed — gross SOL already paid out to all holders

onchain_total — total SOL ever deposited — ground truth for all security checks

VII.B — Protocol Fee and Claim Mechanics

Revenue reaches holders through a pull-based claim. At each deposit event, the off-chain engine records the gravity contribution of every holder and updates their cumulative earned amount. When a holder claims, the backend signs a message attesting to their current lifetime earnings. The on-chain program verifies this, computes the unpaid delta, applies a 10% protocol fee, and transfers 90% net to the holder.

CLAIM COMPUTATION

$$\text{NetClaim} = (\text{CumEarned} - \text{AlreadyClaimed}) \times 90\%$$

CumulativeEarned — lifetime gross SOL earned, attested by the backend

AlreadyClaimed — gross previously paid, stored in UserClaim PDA

Protocol fee — 10% of the delta — retained in treasury, not separately routed

Example: a holder's cumulative earned grows from 5.0 SOL to 8.0 SOL. Claimable delta = 3.0 SOL. Holder receives 2.7 SOL; 0.3 SOL stays in the treasury as protocol revenue. UserClaim.claimed_amount is updated to 8.0 SOL (the gross, not the net).

VIII. Registering Tokens and Launching Yield-Bearing Memecoins

FairFun is open infrastructure. Any Solana token — existing or new — can join the protocol. There are two entry paths: registering an existing token to start gravity distribution, or launching a new token designed from the ground up as a yield-bearing community coin.

VIII.A — Registering an Existing Token

Any token with an existing mint address — PumpFun tokens, Raydium pairs, custom SPL or Token-2022 mints — can register with FairFun in a single admin-signed transaction. No token migration, no liquidity move, and no change to the token's existing market structure is required.

Register an existing token mint

```
// One-time setup by the project admin
register_pool(token_mint: Pubkey) -> Result<()>

// Creates two new PDAs:
// RewardPool: seeds=["rewards_pool", token_mint]
// Treasury: seeds=["rewards_treasury", token_mint]

// After registration, anyone can deposit:
deposit(amount: u64) -> Result<()> // routes SOL into JitoSOL

// Holders claim whenever they choose:
claim(cumulative_earned, observed_total_deposits, expires_at) -> Result<()>
```

What happens immediately after registration:

- The treasury PDA is created. Any SOL deposited to it begins earning JitoSOL yield immediately.
- The gravity engine begins tracking all holders at the first deposit event. Wallets that held before registration receive no retroactive gravity — the ledger starts at registration.
- Existing liquidity (Pump AMM, Raydium, Meteora) is unchanged. Holders use their wallets exactly as before.
- The project can start routing trading fees to the treasury manually, or integrate the deposit instruction into their fee routing contract for automatic deposits.

EXISTING PUMPFUN TOKENS

A PumpFun token that has already migrated to Raydium or remains on Pump AMM can register with FairFun without moving liquidity. The project simply calls `register_pool` with the token's existing mint address. Trading fees from the AMM can then be routed to the new treasury PDA via a CPI call or a manual periodic deposit. Holders start earning gravity from the first deposit, with no change to how they hold or trade the token.

VIII.B — Creating a Yield-Bearing Memecoin with Meteora DBC

The most powerful way to launch a FairFun-native yield-bearing token is to pair it with Meteora's Dynamic Bonding Curve (DBC) program using a flat (constant) price curve. This combination produces a fundamentally different kind of token: one whose value proposition is community membership and fee yield, not price speculation.

Why Meteora DBC with a Flat Curve

Meteora DBC supports configurable price curves. A flat curve sets a constant price per token regardless of buy or sell volume:

- **Constant price.** Every buyer — first or thousandth — pays exactly the same price per token. There is no 'buy early at low price and sell to later buyers at higher price' dynamic.
- **Configurable fees.** DBC charges a fee (typically 1–5%) on every buy and sell. These fees accumulate and can be routed to any destination — including a FairFun treasury PDA.
- **Full liquidity at launch.** Unlike Pump.fun's bonding curve, there is no graduation threshold. The token trades at its fixed price from the first transaction.

The Launch Flow

- 01 Create token mint**
Deploy a Token-2022 mint (or standard SPL). Set supply and decimals.
↓
- 02 Register with FairFun**
Call `register_pool(token_mint)`. Creates treasury PDA and gravity ledger.
↓
- 03 Create Meteora DBC pool**
Deploy a DBC pool with flat price curve and fee routing to the FairFun treasury PDA. Set fee rate (e.g. 2% per trade).
↓

04 First deposit / first trade

When the first buy occurs, the 2% fee deposits to the treasury. Gravity engine records holder balances at this deposit event.

↓

05 Ongoing distribution

Every subsequent buy or sell generates a fee deposit. Each deposit advances gravity scores and increases all holders' cumulative earnings.

↓

06 Holders claim

At any time, holders (or their delegates) submit a claim transaction and receive their accumulated SOL yield.

VIII.C — The Community Economy Model

The flat curve + FairFun combination is not merely a technical configuration. It represents a fundamentally different philosophy about what a memecoin is for.

THE SPECULATIVE MEMECOIN VS. THE COMMUNITY ECONOMY

Traditional memecoin: buy early at low price, wait for new buyers to push price up, sell to them at a higher price. Your gain is their loss. The token is a zero-sum timing game.

FairFun community coin: buy at a fixed price to join a community. Every subsequent buyer pays the same price you paid. But every subsequent buyer's fee goes to you (and all other holders) via gravity. Your gain comes from community growth, not from someone else's timing mistake. The token is a positive-sum membership economy.

The Economics of Community Membership

Under this model, the token price is the community's entry fee. The AMM fee is each member's ongoing contribution to the shared treasury. Gravity determines each member's share of that treasury based on how long they have been present at fee-generating moments.

- **New member joins.** They pay the fixed entry price. 2% of their purchase fee goes to all existing members via gravity distribution.
- **Existing member's return.** The longer they have held and the more fee events they have witnessed, the higher their gravity, and the larger their share of every future fee.
- **Community growth = direct yield.** When the community grows from 1,000 to 10,000 members, trading volume grows, fees grow, and all long-term holders' yield grows proportionally.

- **No need to sell.** The return on holding comes from fee yield, not price appreciation. A member can hold indefinitely and earn continuously without ever executing a sale.

This is the closed economy model for on-chain communities. The community's economic activity — buying, selling, trading, using — generates fees that flow back to the community's most loyal members. The token is not a bet on price appreciation; it is a membership stake in an economy where loyalty is measured and rewarded automatically, on-chain, without intermediaries.

Price Stability as an Emergent Property

- **No speculation pressure.** A flat curve removes the incentive to buy early for price gains. New members buy because they want to earn fees, not because they expect to sell higher.
- **Holders have no reason to sell.** Exiting forfeits future fee yield AND accumulated gravity advantage. The opportunity cost of selling grows with each fee event missed.
- **Fee yield provides a return floor.** Even if the token has no secondary market price appreciation, holders earn a yield from ongoing community activity. Holding has positive expected value independent of price.

IX. Flexible Claim Modes

Five claim instructions cover different settlement preferences. All enforce the same seven security guards.

Instruction	Signer	Output	Batch
claim	Holder (self)	SOL to holder	No
delegated_claim	Any third party	SOL to claimant	No
delegated_claim_many	Any third party	SOL to each claimant	Up to 8
delegated_claim_to_tokens	Any third party	Project tokens to claimant	No
delegated_claim_many_to_tokens	Any third party	Tokens, single AMM buy	Up to 8

Delegation and the UserDelegationSettings PDA

Any third party can claim on behalf of a holder whose `delegated_claims_enabled` flag is true (default on first initialization). The delegator pays transaction fees; the SOL always flows to the claimant. Holders disable delegation via `set_delegated_claims_enabled(false)`.

Batch Claims

Up to 8 claimants per transaction. Single backend signature authorizes the batch. Each entry must exceed 0.001 SOL gross claimable. `BatchRewardsClaimed` event emitted at batch completion.

Claim-to-Tokens via Pump AMM

The treasury wraps SOL to WSOL, executes a Pump AMM buy, transfers project tokens to the claimant's Token-2022 ATA, and closes the WSOL account. `min_base_amount_out` provides slippage protection. Batch variant aggregates all claimants into a single AMM purchase and distributes tokens pro-rata; integer dust remains in the treasury's token account.

X. System Architecture

FairFun is a hybrid system: an event-driven off-chain engine paired with a secure Anchor program on Solana.

Off-Chain Gravity Engine

Triggered by each treasury deposit event. Reads all holder balances from the token's on-chain account list, computes each wallet's balance fraction at that moment, and updates cumulative earned amounts. Because computation is event-driven rather than epoch-driven, the engine is idle between deposits and requires no continuous scheduling infrastructure.

On-Chain Account Structure

Account	Seeds	Purpose
GlobalConfig	b"rewards_config"	Admin pubkey and backend signer authority.
RewardPool	b"rewards_pool" + token_mint	Pool metadata: total_deposited, total_claimed, active.
Treasury	b"rewards_treasury" + token_mint	Zero-data SOL vault (JitoSOL-productive).
UserClaim	b"rewards_user_claim" + pool + user	claimed_amount = gross cumulative at last claim.
UserDelegationSettings	b"rewards_user_delegation_settings" + pool + user	Delegation flag per (pool, holder) pair.

Backend Signer and Message Format

Single-claim message (88 bytes, little-endian)

bytes 0-31 user pubkey (32 bytes)

bytes 32-63 pool pubkey (32 bytes)

bytes 64-71 cumulative_earned (u64 LE)

bytes 72-79 observed_total_deposits(u64 LE)

bytes 80-87 expires_at (i64 LE)

Batch-claim message (36 + N*56 bytes)

bytes 0-31 pool pubkey (32 bytes)

bytes 32-35 entry_count (u32 LE)

per entry (56 bytes): claimant pubkey | cumulative_earned | observed_total_deposits
| expires_at

Administrative controls: `set_pool_active` marks a pool active or inactive (inactive pools reject all claims but cannot be drained). `set_backend_authority` rotates the signer key. No admin instruction can transfer SOL out of the treasury.

XI. On-Chain Security Model

Seven guards are checked on every claim before any SOL transfer executes. Guards 1-5 run in `calculate_claimable`; guards 6-7 in each instruction handler.

Guard	Condition	Guarantee
1. Monotonicity	<code>cumulative_earned >= already_claimed</code>	Earnings never decrease.
2. Deposit cap	<code>cumulative_earned <= observed_total_deposits</code>	Cannot claim beyond backend-recorded deposits.
3. Truth check	<code>observed_total_deposits <= onchain_total</code>	Backend cannot inflate beyond on-chain reality.
4. Non-zero claim	<code>claimable > 0</code>	Dust/no-op transactions rejected early.
5. Pool solvency	<code>new_total_claimed <= onchain_total</code>	Pool cannot overpay across all holders.
6. Liveness	<code>clock <= expires_at AND pool.active</code>	Expired or paused claims rejected.
7. Vault solvency	<code>treasury.lamports() >= claimable</code>	Vault must hold SOL to cover the transfer.

For delegated paths: delegator cannot equal claimant, and `claimant.delegated_claims_enabled` must be true.

REPLAY AND SINGLE-USE AUTHORIZATION

Each signed message binds to a specific claimant, pool, `cumulative_earned`, and `expires_at`. After a claim, `claimed_amount = cumulative_earned`, so resubmitting the same message produces `claimable = 0` and fails guard 4. Expiry + monotonicity makes each signed authorization single-use.

XII. Extended Mechanics: Raffles and Collections

XII.A — Raffle Systems

A holder's gravity score can serve as a probabilistic claim ticket. Instead of a deterministic proportional share, a raffle distributes a prize to one randomly selected winner. Probability of selection equals gravity share of total.

RAFFLE SELECTION PROBABILITY

$$P(\text{win})_w = \frac{G_w}{G_{\text{total}}}$$

G_w — gravity of wallet *w* at time of raffle draw

G_{total} — total gravity across all eligible wallets

Expected value — identical to proportional distribution; variance is the engagement driver

Raffle prizes can be SOL from the treasury, purchased project tokens, NFTs, or access rights. On-chain randomness is sourced from a recent blockhash or a VRF oracle. Use cases:

- **Jackpots.** Large periodic prizes funded by a reserve drawn from treasury revenue.
- **NFT drops.** Gravity-weighted selection for exclusive collection releases, rewarding long-term community members.
- **Cross-ecosystem gifts.** The \$FAIRFUN gift raffle (Section VI.B) is an instance of this mechanic, funded by \$FAIRFUN trading fees.

XII.B — Multi-Coin Collections

A Collection groups multiple independent token pools into a shared gravity namespace. Gravity accumulated in any member token contributes to the holder's collection-level score.

COLLECTION GRAVITY: WEIGHTED AGGREGATE

$$G_w(C) = \sum_k w_k \cdot G_w(\text{token}_k)$$

G_w(token k) — gravity accumulated by wallet w in member token k

w_k — collection-defined weight for token k

G_w(C) — total collection gravity for wallet w

Share_w — $G_w(C) / G_{\text{total}}(C)$ — used for collection-level treasury distributions

- **Themed universes.** Multiple tokens in a shared IP or game world. Holding any token builds collection gravity earning from the whole universe's revenue.
- **Season passes.** New campaign token each season. Past-season gravity carries forward as a permanent baseline.
- **Launchpad portfolios.** A launchpad's collection token earns from all incubated projects' fee revenues.
- **Cross-project partnerships.** Two independent projects bridge gravity pools, rewarding community overlap.
- **Weight governance.** Weights apply prospectively, not retroactively. Past gravity accumulation is always honoured at the weight in effect when it was earned.

XIII. Pre-Pump Pools, Coordinated Launch, and the Liquid Staking Stack

This section describes a four-layer primitive stack that transforms how early supporters participate in a token launch. Each layer is independently composable: the LaunchPool, the Threshold-Locked Vault, the Liquid Staking Token, and the Automatic Gravity Registration. Together they allow a group of participants to coordinate a fair, sniper-resistant launch entry, receive immediate yield-bearing liquidity in exchange for their locked position, and begin accumulating gravity from the first moment — all within a single atomic on-chain flow.

XIII.A — The Pre-Pump Coordination Problem

When a new token launches on PumpFun, the initial bonding curve price is at its lowest. Every participant knows this, which creates a race: whoever buys first gets the cheapest tokens and the largest profit if price rises. This race has three destructive consequences:

- **Sniper bots win.** Automated bots with low-latency RPC connections buy at launch and immediately have a lower cost basis than any human participant. Retail enters at a disadvantage from block one.
- **Early buyers dump on later buyers.** Having bought cheaply, the rational strategy for each individual early buyer is to sell to later buyers at a higher price. This creates sell pressure precisely when the token needs buy support.
- **Community members compete instead of cooperate.** People who might collectively back a project are individually racing to front-run each other. The project's supporters become its largest source of sell pressure.

THE COORDINATION SOLUTION

If all early supporters pool their SOL before launch and execute a single coordinated buy at the moment of launch, they all receive the same price — no one is front-run by anyone else in the group. The pool contract becomes the first buyer, not an individual. And because all participants have the same cost basis, none has an individual incentive to dump on the others.

XIII.B — The LaunchPool Contract

A LaunchPool PDA is created before the token launches. Participants commit SOL to the pool during a funding window. The pool accumulates until a minimum target is met or a deadline is reached, then executes the coordinated buy.

LAUNCHPOOL PARTICIPANT SHARE

$$\text{Share}_p = \frac{\text{SOL}_p}{\text{TotalSOL}_{\text{pool}}}$$

SOL_p — SOL committed by participant p

TotalSOL_{pool} — total SOL in the pool at launch execution

Share_p — participant p's proportional claim on all tokens purchased

LaunchPool account structure:

LaunchPool PDA seeds and state

```
seeds: [b"launch_pool", token_mint, launch_id]

state {
  target_token_mint: Pubkey, // token that will be purchased at launch
  total_committed: u64, // lamports pooled so far
  participant_count: u32,
  status: enum { Funding | Launched | Cancelled },
  thresholds: Vec<(u64 /*market_cap_usd*/, u8 /*unlock_bps*/>,
  min_commitment: u64, // minimum per participant (e.g. 0.1 SOL)
  deadline: i64, // unix timestamp for funding window close
}
```

Key instructions:

- `commit_sol(amount)` — participant deposits SOL into the pool during the funding window. A UserCommitment PDA records their individual amount.
- `uncommit()` — participant withdraws their SOL before launch if they change their mind. Only available before the pool executes.
- `execute_launch()` — callable by anyone once the deadline passes or the target is met. Executes the coordinated buy and initiates the vault and sToken creation flow.

— `cancel_pool()` — if minimum target is not met by deadline, participants can reclaim their SOL.

XIII.C — Program Sniper: Coordinated Buy at Launch

When `execute_launch()` is called, the LaunchPool contract acts as the buyer in a single atomic transaction:

01 Detect launch

`execute_launch()` is callable once the token's PumpFun pool exists on-chain. The program verifies the Pump AMM pool address for the target mint.

↓

02 Execute coordinated buy

The LaunchPool PDA (as a program-owned signer) submits a buy instruction to the Pump AMM using all pooled SOL. The pool's transaction competes on the same footing as any other buyer — but as a single large order, not N individual competing transactions.

↓

03 Receive tokens

All purchased tokens are deposited directly into the ThresholdVault PDA, not into any individual participant's wallet. No participant ever holds raw tokens from this buy.

↓

04 Mint sTokens

The ThresholdVault mints sTokens to each participant proportional to their `Share_p`. This happens in the same transaction or in a subsequent participant-initiated claim instruction.

WHY ONE BUY IS BETTER THAN MANY

A single coordinated buy from one PDA avoids the internal front-running race. Every participant receives tokens at the same effective price because their shares are computed against the total purchase. A group of 200 participants sending individual transactions would fight each other on the bonding curve; as a pool, they execute once at the same slot.

XIII.D — Threshold-Locked Token Vault

All tokens purchased by the LaunchPool are held in a ThresholdVault PDA. Tokens cannot leave this vault until on-chain market cap milestones are verified. This prevents participants from immediately dumping their position and aligns their economic interests with the token's long-term price performance.

MARKET CAP VERIFICATION (CHECKED AT EACH UNLOCK ATTEMPT)

$$\text{MarketCap} = P_{\text{oracle}} \times \text{supply}_{\text{circ}}$$

oracle_price_usd — token USD price from Pyth or Switchboard, queried at unlock time

circulating_supply — on-chain total supply from the token mint account

MarketCap — compared against each threshold; if met, that tranche unlocks

Threshold structure example:

Tranche	Market cap threshold	Tokens unlocked	Cumulative unlocked
1	\$100,000	25% of vault	25%
2	\$500,000	25% of vault	50%
3	\$1,000,000 (graduation)	25% of vault	75%
4	\$5,000,000	Final 25%	100%

Any wallet can call `unlock_tranche(tranche_id)` once the on-chain market cap verifiably exceeds the threshold. The unlocked tokens do not go to participants directly — they increase the redemption value of sTokens. Participants access their unlocked tokens by redeeming sTokens at the improved exchange rate.

XIII.E — Primitive 2: Liquid Staking Tokens (sTokens)

Rather than waiting for market cap thresholds to receive tokens, participants receive sTokens immediately at launch. An sToken represents a proportional claim on the ThresholdVault's contents — a claim that becomes more valuable as each tranche unlocks.

STOKEN EXCHANGE RATE MECHANICS

$$\text{Rate}(t) = \frac{\text{unlocked_value}(t)}{\text{total_sTokens}}$$

unlocked_token_value(t) — market value of all tokens available for redemption at time t

total_sTokens — fixed sToken supply minted at launch (does not change with unlocks)

Rate(t) — SOL or token value of one sToken — grows as each tranche unlocks

This is the same exchange-rate growth model as JitoSOL: the sToken count is fixed, but each sToken becomes redeemable for more underlying value over time. Participants can:

- **Hold sTokens** and wait for exchange rate to grow as thresholds are reached.
- **Sell sTokens** on the secondary market. The buyer inherits the locked position and the exchange rate appreciation. The seller receives immediate liquidity without triggering any unlock.
- **Redeem sTokens** for unlocked tokens once tranches are available. Redemption is proportional to the current exchange rate.
- **Use sTokens as collateral** in DeFi protocols that accept Token-2022 assets.

Critically, this primitive is not exclusive to pre-pump participants. Any holder of the underlying token can deposit into the ThresholdVault at any time and receive sTokens representing their position. This creates an open liquid staking market for the token: locking earns gravity; the sToken receipt stays liquid.

STOKEN SECONDARY MARKET DYNAMICS

A participant who receives sTokens at launch but needs liquidity before any threshold is reached can sell their sTokens. The buyer pays a market price that reflects the discounted expected value of the locked tokens. The vault is undisturbed — no tokens unlock early. This creates a price discovery mechanism for the premium or discount on locked early positions, without forcing premature unlocks.

XIII.F — Primitive 3: Automatic Gravity Registration

The third layer connects sTokens to FairFun's gravity system. When the LaunchPool mints sTokens to participants, those sTokens are automatically registered in the FairFun gravity pool for the underlying token. No separate staking instruction is required. Gravity begins accumulating from the moment sTokens are received.

Implementation via Token-2022 Transfer Hook:

sToken transfer hook: gravity auto-registration

```
fn transfer_hook(ctx: Context, amount: u64) -> Result<()> {  
    // On every sToken transfer (including the initial mint to participants):  
    let deposit_event = ctx.accounts.gravity_pool.last_deposit;  
    // Record sender exit from gravity tracking at current deposit index  
    if sender.token_balance > 0 {  
        sender.gravity_checkpoint = deposit_event;  
    }  
    // Record receiver entry – gravity accrues from next deposit event forward  
    receiver.gravity_checkpoint = deposit_event;  
    receiver.token_balance += amount;  
    sender.token_balance -= amount;  
    Ok(())  
}
```

Because the sToken uses Token-2022 with a transfer hook, gravity registration is automatic and universal: every sToken holder — whether a pre-pump participant, a secondary market buyer, or a new deposit-to-vault user — begins accumulating gravity from their first sToken balance forward, with no action required.

Auto-Lock for Pre-Pump Participants:

For participants in the LaunchPool specifically, the `execute_launch()` transaction includes an additional step: it calls an instruction on the FairFun program that marks the participant's sToken position as 'launch-locked' in the gravity ledger. This means:

- Their gravity accumulation begins at block 0 of the token's life, giving them the maximum possible head start over any later entrant.
- The launch-lock flag records that this position originated from a LaunchPool, making it eligible for any LaunchPool-specific reward tiers or priority raffle entries that the project

configures.

- The flag is attached to the wallet, not the sToken. If they sell their sTokens, the buyer's position is not launch-locked (they receive gravity from their purchase date, not from day 0).

XIII.G — The Complete Launch Stack and Lifecycle

The four primitives compose into a single coherent launch stack. The following table summarizes each layer, its inputs, outputs, and the liquidity it provides:

Layer	Input	Output	Liquidity	Gravity
Primitive 0: LaunchPool	SOL commitments	Coordinated buy order	SOL locked until launch	None yet
Primitive 1: ThresholdVault	Purchased tokens	Tokens locked until MC thresholds	Token illiquid	None
Primitive 2: sToken	Vault share claim	sToken (freely tradeable)	Full (sToken liquid)	None yet
Primitive 3: Auto-gravity	sToken receipt	Gravity accumulation from day 0	Full	Active immediately

A Complete Lifecycle Example:

01 Funding window

200 participants each commit 0.5 SOL. LaunchPool accumulates 100 SOL total. Minimum target met; deadline countdown begins.

↓

02 Token launches on PumpFun

execute_launch() is called. LaunchPool PDA executes a single 100-SOL buy on the Pump AMM at the initial bonding curve price. All 100 SOL worth of tokens go to ThresholdVault PDA.

↓

03 sTokens minted

ThresholdVault mints 10,000 sTokens total (50 per participant proportional to their 0.5 SOL / 100 SOL = 0.5% share). sTokens are Token-2022 with transfer hook. Each participant receives 50 sTokens in their wallet.

↓

04 Auto-gravity registration

Transfer hook fires on mint. Each participant's gravity checkpoint is set to `deposit_index = 0`. Their gravity begins accumulating from the next fee deposit to the FairFun treasury. They are marked launch-locked in the gravity ledger.

↓

05 Day 1 through threshold 1

Token trades on PumpFun. Fee deposits flow to FairFun treasury. Every deposit advances sToken holders' gravity. Participants earn SOL yield with no tokens in hand and no market cap threshold reached yet.

↓

06 Threshold 1 reached (\$100K MC)

Anyone calls `unlock_tranche(1)`. 25% of vault tokens become redeemable. sToken exchange rate increases: 1 sToken now redeems for 25% more tokens than at launch. Participants choose: redeem, hold, or sell sTokens at improved rate.

↓

07 Participant A sells sTokens at threshold 2 (\$500K MC)

A sells 50 sTokens to Buyer B on secondary market at current premium. Transfer hook fires: A's gravity checkpoint frozen; B's gravity begins. ThresholdVault is undisturbed. No tokens leave the vault from this trade.

↓

08 Full unlock (graduation at \$1M MC)

All tranches unlocked. sToken exchange rate = full token value. Remaining holders redeem sTokens for tokens at the final rate. Gravity continues for any holder who elects to keep sTokens staked.

WHY THIS CHANGES EARLY PARTICIPATION ECONOMICS

Pre-pump participants in the traditional model face a dilemma: lock up SOL with no yield while waiting for thresholds, or dump early and lose their community standing. The FairFun stack eliminates this dilemma entirely. From the moment of launch, participants hold liquid sTokens that (a) earn gravity-weighted SOL yield from trading fees, (b) appreciate in exchange rate as thresholds unlock, and (c) carry a day-0 gravity advantage that compounds into a larger share of all future deposits. The rational strategy is to hold as long as possible — not because of lock-in, but because the economic return on holding grows with time.

XIV. Real-World Applications

This section describes two flagship use cases that demonstrate how FairFun's primitives compose into complete, production-ready applications. The first is the Community Bundled Launch — a mechanism for groups of friends to collaborate on a fair token deployment. The second is \$EEVEE, a living evolution protocol that builds a gamified training economy entirely on top of FairFun's infrastructure: pre-pump launches, locked token distribution, gravity-weighted yield, and multi-coin collections.

XIV.A — The Community Bundled Launch

The most common early-stage token dynamic is destructive: each member of a community races to buy first on the bonding curve, front-running their own friends. The winner profits at the expense of everyone who was one block slower. The community that should be cooperating to build the project competes against itself instead.

The Use Case

A group of friends, a Discord community, or a DAO decides to launch a token together. They each contribute SOL to a shared LaunchPool before the token exists. The pool accumulates during a funding window — days, hours, or minutes — with each participant's share recorded on-chain at contribution time. When the funding window closes:

01 Token deployed

The LaunchPool contract deploys the token mint and creates the PumpFun AMM pool in a single atomic transaction. Every participant funded the same deployment; every participant pays the same initial bonding curve price.

↓

02 Coordinated buy executed

The program immediately executes a buy transaction with all pooled SOL. Tokens arrive in the ThresholdVault PDA, never in any individual wallet. The group acts as a single first buyer, not N competing individuals.

↓

03 sTokens distributed

Each participant receives sTokens proportional to their SOL contribution. sTokens are tradeable immediately, but the underlying tokens unlock only as market cap milestones are reached — aligning every participant's incentive with the token's long-term performance.

↓

04 Gravity begins

Token-2022 transfer hooks register every sToken holder in the FairFun gravity pool. From block one, participants earn SOL yield from the token's trading fees. The group that launched the token is also its earliest gravity accumulators.

WHAT THIS CHANGES

In a traditional PumpFun launch, the first buyer captures the maximum upside and the community has no mechanism to coordinate. In a FairFun bundled launch, the community IS the first buyer. Every participant gets the identical entry price, the identical lock schedule, and the identical gravity head start. There are no insiders, no whitelist advantages, and no sniper tax. The economic relationship between the project and its earliest supporters is fair by construction.

XIV.B — \$EEVEE: The Living Evolution Protocol

\$EEVEE is a flagship application that demonstrates every FairFun primitive working together in a single coherent product. It is a living training game built on Solana where buying the base \$EEVEE token drives a gamified evolution engine. Over time, the engine deploys entirely new tokens — the Eeveelutions — and distributes their supply to the community that built the incubator, using FairFun's bundled launch and gravity infrastructure.

Overview

- Every buy of \$EEVEE earns the buyer training credits proportional to their SOL spent.
- When credits reach 0.1 SOL, a spin ticket is created and enters a FIFO Trainers Queue.
- Tickets take turns spinning three wheels, each showing one of eight Eeveelution portraits.
- The spin result — derived from a future blockhash so neither party can predict or manipulate it — determines an outcome: training, catch, or birth trigger.
- All spin activity charges an incubator that tracks genetic DNA across eight elemental genes.
- When a birth threshold is reached, the incubator deploys a new Eeveelution token via FairFun's bundled launch and distributes its supply to \$EEVEE holders.

The Training Economy: Credit, Ticket, Queue

Buying \$EEVEE does not immediately produce a spin. Instead, each buy earns training credits that accumulate toward the next spin threshold:

TRAINING CREDIT ACCUMULATION

$$\text{Credit}_w + = \frac{\text{BuyLamports}}{0.1 \text{ SOL in lamports}}$$

BuyLamports — SOL spent on this purchase, in lamports (1 SOL = 1,000,000,000)

TrainingCredit_w — running credit balance for wallet w (denominated in 0.1 SOL units)

Threshold — when credit $\geq$ 1.0 unit (= 0.1 SOL), one spin ticket is created; credit resets

A buy of 0.37 SOL produces 3 full tickets and leaves 0.07 SOL residual credit in the wallet's trainer account. Each ticket is a separate, atomic entity in the queue. A buy of 0.01 SOL contributes 0.01 SOL toward the next ticket but does not create one yet.

Buy size	Tickets created	Residual credit
0.01 SOL	0	0.01 SOL
0.10 SOL	1	0.00 SOL
0.37 SOL	3	0.07 SOL
1.00 SOL	10	0.00 SOL

FIFO Fairness: The Trainers Queue

Spin tickets enter a first-in-first-out queue. The processor takes exactly one ticket at a time from the queue head, resolves its outcome, settles any reward, and only then advances to the next ticket. This guarantees that no participant — no matter how large their buy — can jump ahead of earlier participants.

Queue state machine: one active ticket at a time

```

loop {
  1. Poll confirmed $EEVEE buys → update trainer credits
  2. For each credit  $\geq$  0.1 SOL → create spin ticket, push to queue tail
  3. If no active ticket AND queue non-empty:

```

```

a. Dequeue oldest ticket (status: queued → processing)
b. Wait for future blockhash (committed entropy slot)
c. hash(ticket_id + blockhash) → derive 3 wheel indexes [0..7]
d. Persist reels result → advance incubator gene weights
e. If catch or birth condition → create settlement job
f. Update trainer XP and stats
g. Mark ticket resolved → advance queue
}

```

XIV.C — The Eight Evolutions and the Gene System

Each wheel face corresponds to one of the eight Eeveelutions, each carrying a distinct elemental gene. Every spin, regardless of match outcome, contributes to the incubator's running DNA state by incrementing the gene weights of the three rolled elements.

Evolution	Element	Gene weight	Wheel index
Vaporeon	Water	water	0
Jolteon	Electric	electric	1
Flareon	Fire	fire	2
Espeon	Psychic	psychic	3
Umbreon	Dark	dark	4
Leafeon	Grass	grass	5
Glaceon	Ice	ice	6
Sylveon	Fairy	fairy	7

Spin Outcomes

Wheels	Outcome	Effect
All different	Training	No token reward. Trainer earns +1 XP. Each rolled gene increments its incubator weight by 1.

Wheels	Outcome	Effect
Two matching	Catch	Trainer catches that evolution. Protocol spends from claimable creator fee pool to buy that evolution token on the open market and send it to the trainer. Reward is solvent-aware: capped by available fee pool.
Three matching	Master Catch / Birth Trigger	Trainer receives special status (+50 XP). If incubator threshold is met, a birth event is initiated: a new Eeveelution coin is deployed via FairFun bundled launch.

The DNA Incubator

The incubator is not a simple progress bar toward a fixed next evolution. It is an eight-dimensional genetic state that accumulates across all spins from every trainer. When a birth event fires, the new coin's identity is determined by the dominant gene combination at that moment:

INCUBATOR DNA SPACE

$$\text{DNAMask} = \cup \{g \in G \mid \text{weight}_g > \text{threshold}_g\}$$

G — set of 8 genes: {water, electric, fire, psychic, dark, grass, ice, fairy}

weight_g — cumulative spin count that produced gene g on any wheel

threshold_g — minimum weight required for gene g to activate in the DNA mask

DNAMask — bitmask of active genes — $2^8 = 256$ possible combinations

This means the community's collective spin history shapes what evolution is born next. If trainers have been rolling mostly water and ice results, the incubator's dominant genes are water and ice — and the next birth produces a water-ice hybrid evolution, not one of the eight base forms. The game space of 256 possible combinations ensures that no two evolution seasons are identical.

Trainer XP and Leaderboard

Action	XP awarded	Leaderboard weight
Spin completed	+1 XP	Base participation
Catch (2-match)	+10 XP	Active hunter
Triple match	+50 XP	Elite trainer
Birth deployed	+100 XP	Genesis contributor

XIV.D — Birth Triggers and New Token Deployment

When a three-of-a-kind spin occurs AND the incubator's claimable pool has accumulated sufficient SOL above a reserve threshold, a birth event is triggered. This is the moment \$EEVEE's entire FairFun infrastructure snaps into action:

01 Birth condition met

Triple-match spin occurs. Incubator checks: `claimable_pool_lamports >= next_birth_target_lamports`. If true, birth proceeds. If not, the birth is deferred to the next triple-match that satisfies the threshold.

↓

02 DNA mask read

The incubator's current DNAMask is read. The dominant gene combination determines the new evolution's element(s), name, and visual identity. This is locked at birth time — the DNA cannot be changed retroactively.

↓

03 Token deployed via FairFun Bundled Launch

The birth engine calls FairFun's LaunchPool infrastructure. A new token mint is created, a PumpFun AMM pool is initialized, and the incubator fund executes a coordinated buy. All acquired tokens enter a ThresholdVault.

↓

04 Supply distributed to \$EEVEE holders

The ThresholdVault mints sTokens representing the new evolution's locked supply. sTokens are distributed to \$EEVEE holders proportional to their gravity score — long-term holders receive a larger share than recent entrants.

↓

05 Gravity registration begins

sToken holders are automatically registered in the new evolution's FairFun gravity pool via the Token-2022 transfer hook. They begin earning yield from the new token's trading fees from the moment of distribution.

↓

06 Incubator resets

Gene weights are reduced (partial reset — not zeroed). The generation counter increments. A new incubation season begins, with accumulated gene history providing a baseline for the next evolution.

SOLVENCY-AWARE REWARDS

Catch rewards (two-of-a-kind) are funded from the claimable creator fee pool. The protocol never promises a fixed reward size. Instead, each eligible catch receives a reward proportional to what the fee pool can actually cover above the birth reserve threshold. If the pool is empty, the catch is still recorded as a game event and the trainer earns XP — but no token payout occurs until the pool refills. This ensures the reward engine is always solvent.

XIV.E — FairFun Primitive Stack in \$EEVEE

Every mechanic in \$EEVEE maps directly to a FairFun primitive. The game is not a standalone product; it is a composition of the protocol's building blocks, demonstrating their generality:

\$EEVEE mechanic	FairFun primitive	Section
Training credits accumulate toward spins	Buy tracking with fractional credit accumulation	IV, VII
FIFO Trainers Queue with one active ticket	Event-driven processing; deposit-event gravity ordering	IV.C
Catch reward funded from creator fees	Treasury deposit + gravity-weighted claim	VII, IX
Birth trigger deploys new token	LaunchPool bundled launch + coordinated buy	XIII.B, XIII.C
New token supply distributed via gravity	ThresholdVault + sToken distribution	XIII.D, XIII.E
sToken holders earn from new token's fees	Automatic gravity registration via Token-2022 hook	XIII.F
Eight evolutions form a collection	Multi-coin collection with shared gravity namespace	XII.B

\$EEVEE mechanic	FairFun primitive	Section
Raffle for master catch selection	Gravity-weighted random raffle	XII.A
Trainer leaderboard and XP	Per-wallet stats aggregated from gravity events	VII

\$EEVEE AS A PROTOCOL PROOF OF CONCEPT

\$EEVEE demonstrates that FairFun's primitives are composable into a complete economic game without any bespoke infrastructure. The training economy is gravity. The birth mechanic is a bundled launch. The evolution collection is a multi-coin collection pool. The catch reward is a treasury claim. The leaderboard is a gravity-derived stat ledger. Every mechanic that makes \$EEVEE engaging is a reusable FairFun building block available to any project.

XV. The Complete Claim Flow

End-to-end lifecycle for a standard self-claim settling as SOL. Token-claim paths add WSOL wrap, Pump AMM buy, and token transfer after the guards.

01 Treasury deposit

Project earns revenue (AMM fee, product sale, manual deposit). SOL deposited to treasury PDA. `pool.total_deposited` incremented. SOL routed to JitoSOL.

↓

02 Gravity update (event-driven)

Off-chain engine reads all holder balances at deposit time. Each wallet's balance fraction (`Bal/Supply`) is added to its gravity score. Cumulative earned amounts updated for every holder.

↓

03 JitoSOL unwind at claim time

When a holder initiates a claim, the relevant JitoSOL is unwound to native SOL. Accumulated staking yield is included in the claimable amount.

↓

04 Backend signing

Backend computes `cumulative_earned` and `observed_total_deposits` for the claimant, signs the 88-byte message with an expiry timestamp.

↓

05 Transaction construction

Client builds two instructions: (1) Ed25519 verification of backend signature, (2) the claim instruction. Both in the same transaction, claim at index `n`, Ed25519 at index `n-1`.

↓

06 Seven security guards

Guards 1-5 in `calculate_claimable` (monotonicity, deposit cap, truth check, non-zero, pool solvency). Guards 6-7 in `handler` (liveness, vault solvency). Any failure aborts with no state change.

↓

07 Fee split and transfer

10% protocol fee retained in treasury. 90% net transferred via `invoke_signed` from treasury PDA to holder wallet.

↓

08 Ledger update

`UserClaim.claimed_amount` = `cumulative_earned` (gross). `pool.total_claimed` += gross claimable. `pool.total_deposited` reconciled upward if needed. `RewardsClaimed` event emitted.

FairFun is the settlement layer for on-chain community economies — where every holder earns from community growth, every loyal member is rewarded by the protocol, and every project can turn its token into a genuine yield-bearing instrument without asking holders to give up a single moment of liquidity.