

\$MORT: An Immutable, Admin-Free Fee Engine for Community-Directed On-Chain Asset Accumulation

Mort

A dead boomer who buys the dip and never sells.

An EVM chain with USDC as its gas token • heymort.co

September 2026 (v2)

Abstract. \$MORT is a memetic token whose economic substance is a fully on-chain, upgrade-free machine that converts trading fees into productive assets and returns them to holders. Every unit of creator share earned on the launchpad, paid in USDC, is claimed, split by a fixed 80/20 rule, and routed through two independent sinks: a daily community-voted asset purchase distributed pro-rata to holders, and a team payout. A buy-and-burn of the token itself runs alongside, executed by the launchpad on every trade rather than by the engine. The system contains no owner, no upgrade proxy, and no privileged control over the value flow. This paper documents the contract architecture, the time-weighted voting mechanism that directs the daily buy, the oracle-bounded and MEV-resistant execution path, and the trust model that follows from immutability. We argue that the design achieves credible neutrality: once deployed, the rules governing money movement cannot be altered by any party, including the authors.

1 Introduction

Most memetic tokens couple a strong narrative to a weak or absent mechanism: value accrual, if any, depends on discretionary treasury decisions that a team can revise, delay, or reverse. \$MORT inverts this relationship. The narrative (a deceased buy-and-hold investor who accumulates forever and never sells) is expressed literally in code as an autonomous fee engine with no escape hatch for its operators.

The token launches on a bonding-curve launchpad on an EVM-compatible network where USDC is the gas token. Trades pay a fee in USDC; the creator's share of that fee (0.35% of every buy and sell, for the life of the token) is the engine's only fuel. The engine's job is to transform that USDC stream into outcomes that are decided at deploy time and never afterward. On day one the assets it can buy are USDC itself (a "hold" day: Mort keeps cash), EURC and wrapped bitcoin; tokenized stocks are admitted the day they exist on the chain.

The design goals are:

- **Immutability of money flow.** No contract that holds or moves fee value may have an owner, a setter, or an upgrade path.
- **Credible neutrality.** The split, the sinks, and the default behavior are fixed constants readable by anyone.
- **Permissionless operation.** Every step in the daily loop can be triggered by any address; the team runs a keeper bot only for convenience, never as a trusted party.
- **Resistance to extraction.** On-chain execution is bounded by independent price oracles and paced to blunt sandwiching and front-running.

The remainder of this paper describes the components

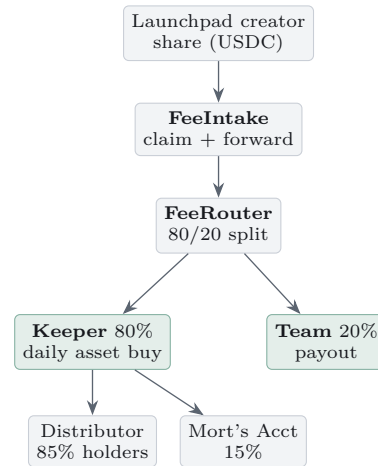


Figure 1: Fee flow. A single intake feeds an immutable split; each sink is an independent, admin-less accumulator. The keeper further routes its daily asset purchase 85/15 to holders and the vault. The token burn (0.20% of every trade) is executed by the launchpad and never passes through the engine.

that realize these goals and the security properties that follow.

2 System Architecture

The engine is a set of small, single-purpose contracts connected in a directed flow from a single intake to two terminal sinks. Contracts that touch value are *admin-less*; the only contracts carrying a narrow privileged action are append-only registries that can never redirect funds.

2.1 Intake

The **FeeIntake** is the front door: it is the token’s *creator* on the launchpad, so the creator share is owed to it and to nothing else. To be the creator the intake must sign the launch itself, which it does through a single ops-only, one-shot **launchVia** that grants the launchpad a bounded USDC allowance for the creation fee, runs the launch calldata, and revokes the allowance; the call is dead forever after. The engine that the intake forwards to needs the token address (the voting contract binds it), so the router is set once, code-checked, immediately after the engine deploys, and is thereafter as immutable as a constructor argument; nothing can leave the intake before that.

Claiming is permissionless: **claimAndForward()** runs an ops-configured call (the launchpad’s creator claim) *from* the intake and forwards the whole balance to the router. Rather than trusting the claim configuration, the intake bounds it by an invariant that holds for any calldata: the intake’s USDC balance may never decrease across the call, and the target may never be the USDC contract (the only place an allowance over the balance could be granted). A wrong or hostile claim therefore fails to claim; it can never spend. What the invariant cannot see, a creator-only launchpad call that redirects *future* payments, is mitigated by time: every claim-configuration change after the first is timelocked for three days in public. A plain **receive()** accepts pushed payments and direct transfers so the forward path never depends on a specific launchpad ABI. On this network, native transfers and ERC-20 transfers of USDC land in the same balance, so nothing is ever wrapped or unwrapped.

2.2 The split

The **FeeRouter** fixes a list of sinks and basis-point shares at deploy (**keeper**= 8000, **team**= 2000 against a denominator of 10,000). There are no setters and no upgrade path. Deposit and split are deliberately separate steps: depositing can never be blocked by a sink, while **split()** is permissionless and works in two moves. It first *books* every sink’s share (pure accounting, no external calls), then pushes each booked amount best-effort under a bounded gas stipend. A sink that cannot take value at that moment keeps its booking, and a permissionless **push(i)** retries it later; the other sinks are paid regardless, and nothing booked is ever re-split. The shape is forced by the network: value sent to an address the stablecoin issuer has blocklisted reverts at the protocol level, and an all-or-nothing split would let one blocklisted sink strand every future fee for everyone. Booked shares confine that failure to the sink itself. Every sink is still an engine contract whose **receive()** is a plain accumulator; all active market operations happen in later keeper steps, never inside a sink.

The engine carries no buyback sink. The launchpad burns 0.20% of every trade of the token, bought on its own market against a 30-minute average price, for the life of the token; a burn leg funded from the 0.35% creator

Share	Sink	Outcome
80%	Keeper → Distributor	voted asset to holders
20%	Team Splitter	fixed per-member shares
<i>outside the engine:</i> 0.20% of every trade burns \$MORT (launchpad)		

Table 1: The immutable 80/20 allocation of the creator share.

share would have been smaller and would have needed an operator or a price bound of its own.

3 The Daily Loop: Time-Weighted Voting

Eighty percent of every fee funds an asset the community selects each day. The selection mechanism, **MortYell**, is the analytic core of the system.

3.1 Voting rule

Between 00:00 and 20:00 UTC, holders name a ticker from an admitted list. The ticker with the greatest weight wins the entire day’s purchase, executed at 21:00. A tie breaks to whichever ticker reached its total first. If nobody votes, the engine holds USDC: a dead boomer defaulting to cash is the joke, but the weighting math is not. The admitted list on day one is USDC, EURC and wrapped bitcoin; tokenized stocks join through the registry’s public timelock the day they land on the chain.

3.2 Voting weight as an integral

Voting power is a true time-weighted balance (TWAB), not a snapshot. To vote, a holder deposits \$MORT into the contract; power accrues as the exact integral of the deposited balance from the moment of the vote to the 20:00 close. For a voter holding balance $b(t)$ over the window $[t_v, T_{\text{close}}]$, weight is

$$w = \int_{t_v}^{T_{\text{close}}} b(t) dt \quad [\text{token-seconds}].$$

Because power is measured in token-seconds, three properties follow directly from the mathematics:

- **Flash-loan immunity.** A deposit and withdrawal in the same block span zero time and integrate to zero weight.
- **Whale-latency penalty.** A large balance arriving seconds before close contributes almost nothing.
- **No double-voting.** Custody, not **balanceOf** sampling, backs the integral, so the same tokens can never vote from two wallets.

Custody is essential: the launchpad token exposes no transfer hooks, so a sampled external balance would be gameable between permissionless finalization pages. Keeping the integral over deposited balances closes that gap. Deposits persist across days (stake once, vote daily) and withdrawal is instant at any time; accrual simply stops. Nothing is ever paid out of the stake pool, so deposits map one-to-one to withdrawals.

3.3 Finalization

After the close, finalization is permissionless and paginated and operates purely on internal accounting with no external calls, so page timing cannot be gamed. The sealed per-voter weights become the day’s TWAB set, which the distributor reuses directly. The voting set and the payout set are therefore identical, by the same weights: a day’s stock is shared exactly among those who held and voted through it, eliminating dividend capture by a last-minute buyer and denying rewards to holders who never voted.

4 Execution and MEV Resistance

The **MortKeeper** holds the 80% asset cut as native USDC and executes the day in a permissionless three-step sequence that any address can drive.

1. **startDay**: after the day is sealed, lock in the winning ticker (or the default on silence) from the append-only registry.
2. **buyChunk**: swap a slice of USDC for the asset. Repeated calls over several minutes form a TWAP execution. On a *hold* day (the winner is USDC itself) there is no swap and no pool to protect: a single call earmarks the balance, in any size, at once.
3. **settleDay**: route the acquired asset 85% to the distributor and 15% to Mort’s Account; on a hold day, the earmarked dollars themselves.

4.1 Oracle-bounded buys

Each chunk independently enforces a minimum output derived from two USD feeds, one for the quote asset (USDC/USD, so a depeg shrinks the bound with the dollar actually spent) and one for the asset, rescaled to the asset’s own decimals, so no single chunk can be sandwiched past the slippage bound. Slippage is capped by a hard constant (`MAX_SLIPPAGE_BPS` = 2000, i.e. 20%) that a deploy cannot exceed. A corporate-action pause on the asset’s oracle blocks the buy rather than trading on a frozen price: a skipped day is strictly better for the engine than a wrong price. Only one day is in flight at a time, keeping USDC accounting unambiguous; the loop runs even on a zero-fee day, buying nothing and settling cleanly. A day whose settlement is wedged (an asset that froze transfers, an address the network refuses value to) can be abandoned after a seven-day long-stop; its purchase is parked, still owed to that day’s voters, and a hold day’s earmark stays reserved so it can never be spent twice.

4.2 Pacing floors

Because **buyChunk** is permissionless, the contract fixes floors that a deploy cannot disable: a minimum interval between chunks (`MIN_CHUNK_INTERVAL` = 15s) and a minimum chunk size (a fraction of the per-day cap). Without a size floor, a one-wei griever could consume one pacing slot per interval and starve an honest keeper’s TWAP. The bot additionally randomizes its start to blunt predictable front-running. The keeper has no admin over

funds: it can only push stock to the immutable distributor and vault, never to an arbitrary address.

4.3 Two views of one dollar

The network exposes USDC twice: as the native asset (18 decimals, used by `msg.value` and account balances) and as an ERC-20 (6 decimals, at a fixed address), over one and the same balance. The engine holds and spends the native view and buys assets that are ordinary ERC-20 balances, including USDC’s own ERC-20 view on a hold day. Every chunk must therefore be a whole micro-USDC (a multiple of 10^{12} wei): a sub-micro remainder would be stranded in one view or truncated in the other, so the keeper and the swap adapter both refuse it. Amounts are lifted to 18 decimals before valuation, so a 6-decimal stablecoin, an 8-decimal bitcoin and an 18-decimal stock are priced by the same formula.

5 Distribution and the Vault

5.1 Distributor

The **StonkDistributor** pays the 85% holder cut of each daily buy pro-rata by the sealed TWAB set. It is pull-claim and gas-bounded per voter: the keeper opens a day with the token and amount deposited, and each voter claims their slice once,

$$\text{payout}_i = \text{amount} \times \frac{w_i}{\sum_j w_j}.$$

Rounding dust (the sum of floors falling below the total) simply rests in the contract. Zero-weight days never reach the distributor: the keeper routes those fully to the vault, and **openDay** refuses a day with no total weight, making division-by-zero and orphaned pots both impossible. The keeper can only add days; it can never claw back or redirect what voters are owed.

5.2 Mort’s Account

The retained 15% accumulates in **MortsAccount**, a one-way vault with no withdraw, no sell, no redeem, no owner, and no upgrade path. The absence of an ordinary exit is the point: with no way to pay anyone, the vault stays clear of redeemable-fund territory. Mort is dead and never sells; his account behaves the same way. A single constrained escape hatch can move holdings only to one fixed, immutable beneficiary Safe and only after a timelock. It cannot redirect, sell, or redeem; regardless of caller or argument, funds reach exactly one pre-committed address. Because the destination is fixed, the off-chain unlock secret (stored only as a hash) is leak-proof and needs no commit-reveal. The one permissionless write is an additive token registration so dashboards can enumerate holdings; valuation happens off-chain from these balances, the stateless **StockValuer** views, and the USD feeds.

6 Extensibility Without Admin Keys

New assets must be admissible after launch without granting anyone power over the value flow. Three append-only structures provide this.

- **Ticker registry.** The single source of truth for the admitted-asset list (stablecoins, blue-chip crypto, and tokenized stocks), recording each asset’s decimals and whether it is the quote asset (a hold ticker). `MortYell` and `MortKeeper` both read it, so they can never drift out of index alignment. Additions only: an index is never mutated or removed, so a listed ticker can never be replaced by a scam. Additions are timelocked, and proposing one cannot touch funds, the split, the sinks, or the default. Tokenized stocks are detected by their ERC-8056 surface and keep the corporate-action pause; plain assets skip it.
- **Swap adapters.** A thin `DispatcherExecutor` holds an append-only token-to-adapter table and forwards each buy to the immutable, single-venue executor for that asset (a Uniswap v3 executor that spends the ERC-20 view of the native USDC it just received, others later). One adapter per token, forever; no admin can redirect a listed token’s venue. A keeper-level executor setter was explicitly rejected as the first redirect lever in the money layer.
- **Team splitter.** The 20% team cut is divided by immutable per-member basis-point shares agreed in writing before launch, pull-based, so one member that cannot receive USDC blocks only their own claim.

Each swap path is validated structurally at registration (byte-length and endpoint checks on the encoded route), so a fat-fingered path cannot be registered at all, and the dispatcher refuses to register an adapter for a token it does not serve, so an admitted-but-unbuyable asset is impossible. An asset admitted before its route exists is still safe: a day it wins settles with nothing bought and the USDC rolls forward.

7 Holder Bootstrapping

To seed a wide, credibly-decentralized holder set, \$MORT uses a custody-free batch distributor (`MortAirdrop`) following the Disperse pattern: one approval plus one `disperseToken()` fans a batch out atomically, so a mid-batch failure can never half-pay a list. The contract never holds a balance and has no owner or state. The engine’s first deployment (August 2026) ran it in escalating waves (an initial visibility push, a capped-grant expansion, and a final wave of 2×10^8 \$MORT across 28,556 unique wallets), with grant sizes assigned by an integer largest-remainder (Hamilton) apportionment of scores so that per-wallet amounts sum exactly to the intended total; the relaunch reuses the same pusher. Because voting weight is time-weighted and custodial, this broad initial distribution translates directly into a broad, sybil-resistant voting base rather than a one-block snapshot that a whale could dominate.

8 Trust Model and Security Properties

The system’s guarantees are structural rather than procedural. They hold because the relevant contracts have no privileged code, not because an operator promises to

behave.

- **No admin over value.** Router, keeper, distributor, team splitter, and vault expose no owner, no setter, and no upgrade proxy touching USDC or holder funds. The intake’s operator can choose the router once, run the launch once, and configure the claim (timelocked after the first); it can never move USDC anywhere but the router.
- **Unstoppable flow.** Every step (claim, split, push, start, buy, settle, claim payout) is permissionless. The team’s keeper is a convenience, not a trust assumption; a stalled bot delays but cannot capture.
- **Bounded execution.** All market buys enforce an independent oracle-derived minimum output under a hard slippage ceiling, with TWAP windows and pacing floors that a deploy cannot weaken.
- **Append-only growth.** The asset list and swap routes can only grow, never mutate, and every addition is timelocked and fund-neutral.
- **Contained failure.** Booked shares confine a rejected push to its own sink; airdrop batches are all-or-nothing; the distributor cannot divide by zero or orphan a pot; an abandoned day’s purchase is parked, never lost.

Residual assumptions are explicit and narrow: the launchpad’s locked liquidity, fee accounting and creator-only surface (a creator-only call that changed the payee would pass the intake’s balance invariant, which is why claim changes are timelocked and the launchpad’s source is read for exactly that before launch), the correctness and liveness of the consumed USD feeds, the network’s blocklist never naming an engine contract, and a launch-time review that every value sink’s `receive()` is a pure accumulator. Each is documented rather than obscured.

9 Conclusion

\$MORT demonstrates that a memetic token can carry a real, self-contained economic mechanism whose rules no party can change after deployment. By fixing the split, the sinks, and the defaults as constants, measuring governance as an integral of custodied stake over time, and bounding every on-chain buy against an independent oracle, the engine turns a narrative about a buyer who never sells into an autonomous, credibly-neutral system. The result is a machine whose most important property is what it *cannot* do: it cannot be paused, redirected, upgraded, or drained by anyone, including its authors.

This document describes a technical system and is not financial advice or an offer of any security. \$MORT is a memetic asset; Mort’s Account holds assets and, by design, cannot pay them out. Deployed contract addresses and source are published at heymort.co.