# ZAMM: A Minimal Multitoken AMM

Version 1.0 • May 2025

z0r0z

## Abstract

ZAMM is a *singleton* automated market-maker that unifies trading, liquidity, and token factory logic inside one ERC-6909 contract. It generalizes the constant-product formula to natively support ETH, ERC-20s, and ERC-6909 ids—including liquidity-provider (LP) shares and freshly minted "coins". By exploiting transient storage (EIP-1153) for intra-block balance accounting, ZAMM can execute multi-hop swaps in half the gas of contemporary AMMs while preserving the composability and simplicity that made early DEXs successful. This paper describes ZAMM's architecture, core algorithms, security model, and expected performance to guide both developers integrating the protocol and non-technical readers seeking intuition.

## 1 Motivation

**Gas efficiency.** Layer 1 Ethereum remains expensive. Users prefer passive, full-range liquidity positions that do not require continuous management, while swapping through monolithic routers or deploying pair contracts incurs redundant storage writes and external calls.
**Simplicity.** The original Bitcoin and Uniswap whitepapers proved that elegance enables adoption. ZAMM purposely avoids exotic curves, bespoke NFT positions, or multi-contract factories that complicate audits and user education.
**Multitoken future.** ERC-6909 compresses the universe of on-chain assets into a single ledger keyed by *ids*. It offers the convenience of ERC-1155 without its callback hazards and matches the direction of smart-account standards such as ERC-7702. ZAMM embraces this paradigm from genesis.

## 2 Design Goals

1. **One contract, many assets**. All pools, LP shares, and factory-minted coins coexist under the same address.
2. **Constant-product invariance**. Preserve the battle-tested $x\,y = k$ intuition for predictable pricing.
3. **Lowest possible gas**. Remove every extraneous `SSTORE`, external call, and memory allocation; rely on transient storage for flash credit.
4. **Unified interface**. A single, symmetrical API that serves both EOAs and smart contracts; power users can batch these calls via ERC-7702, composing custom strategies in one atomic transaction.
5. **Auditable security**. Keep the surface small enough for human review yet leverage known security patterns (reentrancy guard, safe-transfer libraries, Solidity's overflow checks).

## 3 System Overview

### 3.1 Actors

- **Traders** swap one asset for another at the prevailing pool price.
- **Liquidity Providers** deposit pairs of assets and receive fungible LP shares (ERC-6909 ids) that accrue fees.
- **Coin Creators** call `coin()` to mint a new id and optionally bootstrap its market.
- **Protocol Owner** can direct a portion of fee growth to `feeTo` for long-term sustainability.

### 3.2 Key Primitives

**ERC-6909 Ledger** Each *id* maps to a fungible balance. **Pool IDs are 256-bit hashes of the `PoolKey` tuple, guaranteeing a unique and collision-resistant namespace.** Separately, a simple incrementing counter (`coins` ++) assigns sequential IDs to factory-minted native "zCoins."

**Pool Structure** For each unique tuple ($\text{id}_0$, $\text{id}_1$, $T_0$, $T_1$, $f$)—where $T$ denotes the token contract address (or `0x00` for native ETH), id is the ERC-6909 id (zero for ERC-20/ETH), and $f$ is the swap fee in basis points—ZAMM records:

- current reserves $(R_0, R_1)$
- time-weighted price accumulators for on-chain oracles
- total LP supply $S$
- last invariant $k_{\text{last}}$ for protocol fee calculation

**Transient Balance Table (Flash Credits)** Using EIP-1153's *tstore/tload*, temporary balances are recorded per–`msg.sender` for the duration of a transaction. Multi-hop routes therefore move value through memory instead of the ERC-6909 ledger, eliminating needless events and gas.

## 4 Core Algorithms

### 4.1 Swap

**Fee notation.** Let the swap fee be $f$ basis-points (e.g. $f = 30$ for 0.30 %). Define the net-inflow factor

$$\phi \;=\; 1 - \frac{f}{10\,000},$$

so the pool effectively receives only the fraction $\phi$ of the trader's input.

#### Exact–In

For a user-supplied amount $A_{\text{in}}$ the pool balance increases by

$$A^{\star}_{\text{in}} \;=\; \phi\, A_{\text{in}}.$$

Imposing the constant-product invariant

$$\big(R_{\text{in}} + A^{\star}_{\text{in}}\big)\big(R_{\text{out}} - A_{\text{out}}\big) \;=\; R_{\text{in}}\, R_{\text{out}},$$

and solving for $A_{\text{out}}$ yields

$$A_{\text{out}} \;=\; \frac{\phi\, A_{\text{in}}\, R_{\text{out}}}{R_{\text{in}} + \phi\, A_{\text{in}}}\,.$$

**Exact–Out**

For a target output $A_{\text{out}}$ the pool must receive

$$A_{\text{in}} \;=\; \frac{R_{\text{in}}\, A_{\text{out}}}{(R_{\text{out}} - A_{\text{out}})\, \phi} \;+\; 1,$$

where the extra "+1" wei is always added by the reference implementation to guarantee the integer ceiling, ensuring the pool is never under-paid—even when the preceding fraction is already an integer.

**Invariant check (low-level `swap`)**

After transferring the optimistic outputs and executing any callback, the contract recomputes the balances and verifies

$$\big(10\,000\, R_0' - f\, \Delta_0^{\text{in}}\big)\, \big(10\,000\, R_1' - f\, \Delta_1^{\text{in}}\big) \;\geq\; (10\,000)^2 R_0 R_1,$$

which is algebraically equivalent to the fee-adjusted condition

$$\big(R_0' - (1-\phi)\, \Delta_0^{\text{in}}\big)\, \big(R_1' - (1-\phi)\, \Delta_1^{\text{in}}\big) \;\geq\; R_0 R_1.$$

Equality holds for a vanilla single-sided swap with no fee ($f = 0$).

### 4.2 Liquidity Management

**Add** mints shares proportional to the lesser of ($\Delta R_0$, $\Delta R_1$) against current reserves. The first LP locks a tiny `MINIMUM_LIQUIDITY` to avoid division-by-zero griefing.
**Remove** burns shares and returns underlying assets linearly.
**Fee-on** mints $\frac{1}{6}$ of $\sqrt{k}$ growth to `feeTo` when enabled, mirroring Uniswap V2.

## 5 Factory Mint (`coin()`)

Creates a new incremental id, mints an initial supply to the designated `creator` address, and emits a `URI` event for off-chain metadata—no state bloat required.

## 6 Hook Extension System ("Cooks")

A *cook* is any contract whose address is packed into the `feeOrHook` field of a `PoolKey`. At run-time ZAMM dispatches two optional callbacks:

```
beforeAction(sig, poolId, sender, data)-> feeBps
afterAction(sig, poolId, sender, d0, d1, dLiq, data)
```

- **Dynamic fees** — return a custom basis-point value from `beforeAction`.
- **Range vaults / CL-layers** — rebalance (withdraw-swap-add) the cook's own LP position, or mint/burn a synthetic vault share (an ERC-6909 id issued by the cook) whenever the reserve ratio drifts outside a predefined band.
- **Programmable trades** — implement on-chain limit orders, oracle-gated quotes, or fee rebates.

Because the kernel never delegates or `delegatecalls`, a faulty cook can at worst revert *its own* pool; global safety is preserved.

## 7 Order-Book and Timelock Escrow

Beyond swaps, ZAMM exposes two auxiliary primitives:

**Order-Book** A maker posts an OTC quote by hashing the tuple (maker, tokenIn, idIn, amtIn, . . . ) into `orderHash`, then optionally escrows ETH or ERC-6909 internally. Takers may fill against the hash either partially or fully. *Invariant:* only the signed tuple is honoured; price or size cannot mutate mid-fill.

**Timelock Escrow** Any asset (ETH, ERC-20, ERC-6909) can be locked until a future timestamp:

```
lockup(token, to, id, amount, unlockTime)
```

The recipient later calls `unlock` once the lock expires. Use-cases include cliff-vesting, DAO payroll, or delayed team distributions—without deploying extra contracts.

Both modules reuse the same `lock` reentrancy guard and transient balance map, keeping gas overhead minimal.

## 8 Transient Storage: Flash Credits

When a router calls two consecutive pools, the output of the first hop is written to *transient storage* instead of transferring tokens on-chain. Subsequent hops read and clear this credit, paying only memory gas. At function end the EVM discards the transient map, guaranteeing atomicity and preventing leftover dust. This saves ˜15k gas per internal transfer and over 100k on deep paths.

## 9 Security Analysis

- **Reentrancy**: a single `lock` modifier using transient storage deters nested calls.
- **Overflow**: reserves are bounded to $2^{112} - 1$; arithmetic uses Solidity `unchecked` only where provably safe.
- **Invariant enforcement**: post-swap check $\left(R_0' \times R_1'\right) \geq \left(R_0 \times R_1\right)$, adjusted for fees.
- **Slippage & expiry**: user-supplied min/max bounds and deadlines guard UX.
- **Permissioned actions**: only `feeToSetter` may update fee parameters.

Independent audits are ongoing; bug-bounty details will be announced on-chain at the *security.zamm.eth* domain.

## 10 Performance Benchmarks

| Operation | ZAMM | UniV2 | UniV3 | UniV4 |
|---|---|---|---|---|
| Exact-In swap | 45 507 | 84 293 | 115 522 | 92 090 |
| 2-hop route | 70 529 | 140 377 | 363 231 | 128 802 |
| Add liquidity | 74 112 | 107 235 | 140 012 | 119 501 |

Table 1: Gas used on Ethereum mainnet (Foundry snapshot, Jun 2025)

ZAMM remains ≈2× cheaper than Uniswap V2 and ≈5× cheaper than V3 on multi-hop paths.

## 11 Developer Integration

### 11.1 Getting Balances

Use the standard `balanceOf(owner, id)` view on the ZAMM address for both coins and LP shares.

### 11.2 Approvals and Deposits

- **ERC-6909 blanket (EOA)**: grant ZAMM full pull rights once with `setOperator(zamm, true)`. This single transaction authorizes every *id*, including future coins.
- **ERC-6909 per-id**: for tighter limits, approve just the asset you intend to trade via `approve(zamm, id, amount)`, analogous to an ERC-20 allowance.
- **ERC-20**: call the standard `approve(zamm, type(uint256).max)` (or a custom cap) so ZAMM can `transferFrom` during swaps or liquidity actions.
- **Flash-credit flow (`deposit()`)**: Pre-load assets into ZAMM's transient table to avoid intermediate transfers in a multi-hop route:

  `deposit(token, id, amount)` — ETH uses `token = 0x00`.

  - *Self-custody (ERC-7702)* — a smart-account wallet can embed one or more `deposit()` calls in its user-op, followed by the multi-hop `swap` bundle. The bundler pays gas once; ZAMM reads the credits hop-by-hop.
  - *External router* — a helper contract deposits on behalf of the trader, executes the swap path, and finally refunds any unused credit with `recoverTransientBalance()`.

  No allowances are touched; credits live only for the current transaction and cost ˜200 gas instead of ˜15k for a real on-chain transfer.

### 11.3 Launching a Market

1. Call `coin(msg.sender, supply, URI)` to mint a coin id.
2. Provide seed liquidity through `addLiquidity` using ETH, an ERC-20, or another id.
3. Publish the pair's *poolId* (hash of the `PoolKey`) so wallets can query reserves.

Sample scripts are available in the zamm-examples repository.

## 12 Roadmap

- **Hook-based Liquidity Adapters (CL-ZAMM)**: autonomous cooks that withdraw, rebalance and re-add their own LP to mimic concentrated-range, pseudo-stableswap or volatility-band behaviour. They keep any trading fees their LP earns and simply sit idle (full-range fallback) when outside the target band.

- **General-purpose Cook Hooks** – permissionless contracts referenced via `feeOrHook`. A cook may simply act as a policy layer (e.g. dynamic fees, oracle checks) or expose deposits by minting its own share token. When a share is needed, we *prefer* issuing it as an ERC-6909 id so the entire stack lives on a single multitoken ledger, but ERC-20 (or none) remains equally valid.

- **Layer-2 deployments**: deterministic `CREATE3` roll-outs on Optimism, Base, and Arbitrum so all networks share one canonical ZAMM address, unifying liquidity and tooling across chains.

## 13 Conclusion

ZAMM distills the AMM to its essence: one constant-product equation, one multitoken ledger, and one contract address. The result is a system that ordinary users can grasp, developers can integrate in minutes, and the Ethereum network can execute with minimal gas. We invite the community to review, benchmark, and build on top—ZAMM is designed as a foundation, not a walled garden.

**Simple • Efficient • Complete**

## A Appendix A: PoolId Derivation

```
function _getPoolId(PoolKey calldata key) pure returns (uint256 id) {
    assembly {
        let m := mload(0x40)        // free memory pointer
        calldatacopy(m, key, 0xa0) // copy 5*32 bytes
        id := keccak256(m, 0xa0)   // deterministic
    }
}
```

The hash commits to token addresses, ids, and fee tier, ensuring a single canonical market regardless of parameter order.

Given reserves $(R_0, R_1)$ and an input $\Delta_0$ (after the fee factor $\phi$ has been applied), the post-swap reserves are

$$R_0' = R_0 + \phi\,\Delta_0, \qquad R_1' = \frac{R_0 R_1}{R_0'}.$$

Hence

$$R_0' R_1' \;=\; \bigl(R_0 + \phi\,\Delta_0\bigr)\frac{R_0 R_1}{R_0 + \phi\,\Delta_0} \;=\; R_0 R_1,$$

so the fee-adjusted constant-product invariant holds with equality. (The trader's effective input is reduced by the fee; the pool's reserves remain on the same $xy = k$ curve.)